Fabrizia Scorzoni, Giuseppe Costa

Informatica

Programmazione in C++

Loescher

EDUCAZIONE
TECNICA
SUPERIORE

Fabrizia Scorzoni, Giuseppe Costa

Informatica

Programmazione in C e C++

Seconda Edizione

LOESCHER EDITORE

Ristampe

7	6	5	4	3	2	1	N
2015	2014	2013	2012	2011	2010	2009	

ISBN 9788884330291

Nonostante la passione e la competenza delle persone coinvolte nella realizzazione di quest'opera, è possibile che in essa siano riscontrabili errori o imprecisioni. Ce ne scusiamo fin d'ora con i lettori e ringraziamo coloro che, contribuendo al miglioramento dell'opera stessa, vorranno segnalarceli al seguente indirizzo:

Loescher Editore s.r.l.
Via Vittorio Amedeo II, 18
10121 Torino
Fax 011 5654200
clienti@loescher.it

Loescher Editore S.r.l. opera con sistema qualità
certificato CERMET n. 1679-A
secondo la norma UNI EN ISO 9001-2000

Coordinamento editoriale: Mauro Gargantini
Revisione: Roberto Baldin
Impaginazione: Fabrizia Scorzoni
Disegni: Giorgia Galiazzo

Stampa: Sograte - Città di Castello (PG)

INDICE

INTRODUZIONE

Programmazione in C++ si rivolge a tutti i corsi di Informatica in cui si insegna la programmazione utilizzando il linguaggio C++.

Il libro affronta sia le basi della programmazione, focalizzando l'attenzione sulle strutture fondamentali per la realizzazione degli algoritmi in modo indipendente dal linguaggio, che le caratteristiche del linguaggio di programmazione C++.

La prima parte del libro illustra la **programmazione imperativa** e può essere utilizzata anche da chi desidera programmare in C.

La seconda parte affronta la **programmazione a oggetti** illustrando sia le tecniche di programmazione e di progettazione di applicazioni che la descrizione della libreria standard **STL**.

Sono affrontati anche argomenti complessi come le **eccezioni**, la **genericità** e la gestione dei **flussi**.

Per un approfondimento della programmazione a oggetti è proposta una introduzione all'utilizzo di alcuni **design pattern** (schemi che descrivono soluzioni riutilizzabili di particolari problematiche di progettazione).

Come ambiente di programmazione si è scelto wxDev-C++, un ambiente di sviluppo integrato gratuito, che utilizza il compilatore MingW (GNU C++).

Il software può essere liberamente scaricato da Internet all'indirizzo

```
http://wxdsgn.sourceforge.net/
```

RINGRAZIAMENTI

Ringraziamo Giorgia Galiazzo e Roberto Baldin che hanno collaborato alla stesura di questo libro: Giorgia Galiazzo ha realizzato i disegni e Roberto Baldin ha effettuato la revisione.

Gli autori

Modulo 1

Lo sviluppo del software

1 LINGUAGGI DI PROGRAMMAZIONE E PROGRAMMI DI UTILITÀ

PREREQUISITI

nessuno

OBIETTIVI

CONOSCENZE

- Definizione di linguaggio
- Linguaggi a basso e ad alto livello
- Paradigmi di programmazione
- Linguaggi imperativi
- Editor
- Programmi traduttori (assemblatori, compilatori e interpreti)
- Linker
- Caricamento del programma in memoria per l'esecuzione
- Ambienti IDE e RAD

ABILITÀ/CAPACITÀ

- Comprendere la notazione di Backus Naur e i diagrammi sintattici
- Definire la grammatica di semplici linguaggi
- Distinguere i paradigmi di programmazione
- Comprendere le differenti modalità di lavoro usando linguaggi compilati e linguaggi interpretati

1.1 LINGUE E LINGUAGGI

Come le **lingue** servono agli esseri umani per comunicare tra di loro, i **linguaggi di programmazione** servono per comunicare con il computer.
Perché i messaggi inviati possano essere compresi da chi li riceve, sono necessarie delle **regole** che stabiliscano come comunicare con i propri interlocutori.
Ogni linguaggio deve possedere delle regole note e rispettate sia da chi trasmette i messaggi che da chi li riceve, in modo da escludere ogni incomprensione e ambiguità.
Questo è vero in particolare per quanto riguarda la comunicazione con il computer; infatti, se una persona può completare il significato di una frase ambigua in base al contesto in cui la riceve, il computer non ha assolutamente questa capacità: capisce soltanto ciò che gli viene detto in modo esplicito ed esatto.

Una lingua ed un linguaggio di programmazione possiedono una struttura simile: entrambi sono associazioni logiche di simboli che assumono differenti significati secondo come vengono disposti.
Una lingua però permette di esprimere ad altri esseri dotati di intelligenza un grandissimo numero di cose molto diverse tra loro (compresi sentimenti, esperienze, idee ecc.), mentre un **linguaggio di programmazione** ha degli scopi precisi e limitati (eseguire calcoli, analizzare e conservare dati, guidare delle macchine automatiche, produrre suoni o grafici ecc.) e si rivolge a delle macchine che non sono dotate di intelligenza.
Una lingua quindi è composta da un grandissimo numero di vocaboli che assumono significati anche molto diversi in base alle frasi in cui compaiono, mentre tutti i linguaggi di programmazione sono caratterizzati da un numero limitato di **parole riservate** (il vocabolario del linguaggio) che possono essere utilizzate per comporre le **istruzioni** secondo una sintassi molto rigida.

Definizione formale di linguaggio

Un **linguaggio** è un insieme di **frasi** formate usando le parole di un **vocabolario**, costruite utilizzando i simboli ammessi dall'**alfabeto**.

L'alfabeto è un insieme di simboli con cui si possono comporre le parole (può essere formato da lettere, numeri e simboli vari).
Il vocabolario è l'insieme delle parole ammesse; è un sottoinsieme dell'insieme di tutte le parole componibili con i simboli dell'alfabeto (non tutte le parole che si possono comporre hanno un significato in quel linguaggio).
Una frase è un insieme finito di parole; la lunghezza di una frase è data dal numero di parole che la compongono.

Le **regole** del linguaggio sono descritte dalla **grammatica**.

La grammatica si suddivide in due parti essenziali: la sintassi e la semantica.
La **sintassi** fornisce le regole necessarie per comporre le frasi, mentre la **semantica** fornisce il significato delle frasi.

Nota

Per i linguaggi naturali la semantica risulta molto complessa, tanto che a volte il significato di una frase comune può essere frainteso; per i linguaggi di programmazione la semantica risulta più semplice e può essere definita descrivendo le azioni che la CPU deve compiere durante l'esecuzione di ciascuna istruzione (semantica operazionale).

In pratica un **linguaggio** è definito dal suo **vocabolario** e dall'insieme delle **frasi** che si possono formare.

Se il linguaggio ammette un numero finito di frasi, la descrizione del linguaggio può essere fatta per enumerazione, cioè elencando tutte le frasi possibili.
In generale però i linguaggi sono infiniti, ammettono cioè un numero infinito di frasi anche per un vocabolario molto limitato. La descrizione del linguaggio allora può essere fatta stabilendo le regole che indicano come si possono formare le frasi valide (cioè la grammatica del linguaggio).

La **grammatica** di un linguaggio si può definire descrivendo:

- i **simboli terminali**, cioè le parole del vocabolario del linguaggio;
- i **simboli non terminali**, cioè le parole di un metalinguaggio che servono per descrivere il linguaggio;
- lo **scopo del linguaggio**, un simbolo non terminale che rappresenta la categoria più complessa;
- l'insieme delle regole (chiamate **regole di riscrittura** o **produzioni**) con cui si possono associare simboli terminali e non terminali ai simboli non terminali.

Per esempio "articolo", "nome", "verbo" sono simboli non terminali, parole descrittive; "il", "gatto", "mangia" sono parole del linguaggio, simboli terminali.
La possibilità di sostituire un articolo alla parola articolo è una regola di riscrittura.

Notazioni per la descrizione di linguaggi

La notazione di Backus Naur

La notazione di Backus Naur, chiamata anche notazione BNF (Backus Naur Form), permette di descrivere un linguaggio basandosi sulla sua definizione formale; utilizza i simboli:

< >	racchiudono i simboli non terminali del linguaggio;
{ }	contengono l'elenco dei simboli terminali e di quelli non terminali;
::=	è il simbolo di equivalenza che serve per descrivere le regole di riscrittura che associano a simboli terminali altri simboli non terminali o terminali;
|	indica che più possibili definizioni sono in alternativa;
[]	indicano un elemento opzionale
()	indicano che il contenuto può essere ripetuto un numero arbitrario di volte (anche nessuna); dopo le parentesi può essere indicato il numero n di ripetizioni; il valore di default è l'asterisco * che corrisponde a 0 o più volte; per indicare che il contenuto deve essere presente almeno una volta si può usare il simbolo + (indica 1 o più volte).

Un elenco di simboli indica semplicemente la concatenazione dei simboli stessi.

Grammatica per la rappresentazione di numeri binari.

alfabeto dei simboli terminali VT = {0, 1}
alfabeto dei simboli non terminali VN = {<bit>, <numero>}
regole di riscrittura:
<bit> ::= 0|1
<numero> ::= 0|1(<bit>)

Grammatica per la rappresentazione di numeri decimali con segno.

alfabeto dei simboli terminali VT = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .}
alfabeto dei simboli non terminali VN = {<segno>, <cifra>, <numero>}
regole di riscrittura:
<segno> ::= +|-
<cifra> ::= 0|1|...|9
<numero> ::= [<segno>](<cifra>)$^{+}$ [.(cifra)$^{+}$]

I DIAGRAMMI SINTATTICI

I **diagrammi sintattici** rappresentano un metodo grafico di descrizione dei linguaggi, usato solitamente per la descrizione del linguaggio Pascal.
Questo metodo utilizza dei **grafi** per descrivere le regole di riscrittura di un linguaggio.
Si utilizzano dei simboli **rettangolari** per rappresentare i **simboli non terminali** e dei simboli **circolari** (o **ellittici**) per rappresentare i **simboli terminali**; ognuno di questi è un nodo del grafo; i simboli non terminali devono essere definiti in altri grafi.
I simboli sono collegati da **archi** che indicano dei possibili cammini; un nodo incontrato sul percorso fornisce una parola della frase.
Un'**alternativa** viene rappresentata da più archi che escono da un nodo. Una **ripetizione** viene indicata da un percorso ciclico.
Per **formare una frase** si entra nel grafo da sinistra; se si incontra una biforcazione si sceglie uno dei due percorsi; se si incontra un simbolo terminale lo si inserisce nella frase; se si incontra un simbolo non terminale bisogna passare ad esaminare il grafo che lo definisce.

Diagramma sintattico dei numeri binari

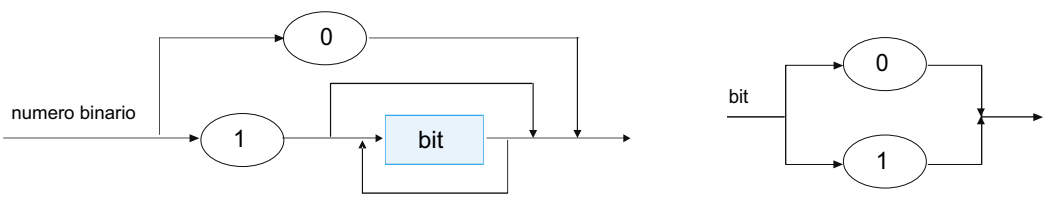
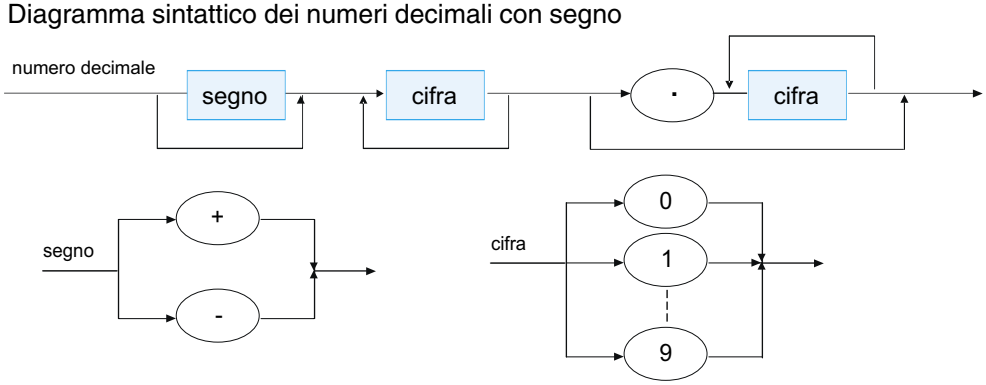


Diagramma sintattico dei numeri decimali con segno



1.2 LINGUAGGI A BASSO E AD ALTO LIVELLO

I linguaggi di programmazione si possono suddividere in due grandi categorie: linguaggi a basso livello (linguaggi macchina e Assembler), detti anche linguaggi orientati alla macchina perché sono specifici di ciascuna macchina, o meglio di ciascun processore, e linguaggi ad alto livello o linguaggi evoluti.

LINGUAGGIO MACCHINA

L'unico linguaggio di programmazione che il computer è in grado di comprendere è il **linguaggio macchina**.

Nota

Tutti gli altri linguaggi devono essere tradotti in linguaggio macchina.

Ogni istruzione è composta da un **codice operativo** che identifica il tipo di operazione e da una parte che individua uno o più **operandi**.

Le istruzioni sono scritte esclusivamente utilizzando **cifre binarie**; per fare riferimento ai dati in memoria e per individuare una istruzione del programma bisogna specificare l'indirizzo della locazione di memoria (che può variare quando si apportano modifiche al programma); la programmazione in linguaggio macchina risulta quindi molto complessa.
Inoltre, poiché il linguaggio macchina dipende dal processore usato, può essere necessario riscrivere completamente il programma per farlo girare su un computer con un processore diverso.

Linguaggio Assembler

Il linguaggio **Assembler** rende un po' più semplice la programmazione perché permette di usare **nomi simbolici** sia per specificare i codici operativi delle istruzioni che per specificare le locazioni di memoria in cui sono memorizzati i dati o che corrispondono a determinate istruzioni; gli indirizzi vengono poi calcolati e sostituiti in modo automatico dall'assemblatore, durante la traduzione del programma.

Il linguaggio Assembler è un linguaggio **a basso livello**, cioè ogni istruzione del linguaggio Assembler corrisponde ad una istruzione del linguaggio macchina; come il linguaggio macchina è quindi strettamente dipendente dal processore.

Le **istruzioni** del linguaggio Assembler sono formate da:

- una **etichetta** che identifica l'istruzione,
- un **codice operativo** simbolico che individua il tipo di operazione,
- uno o più **operandi**.

Le istruzioni si possono classificare in istruzioni dichiarative, esecutive e direttive.

Le istruzioni **dichiarative** permettono di dichiarare i dati utilizzati, variabili o costanti, riservando per ognuno una certa quantità di byte.

Le istruzioni **esecutive** rappresentano le operazioni che si possono eseguire; corrispondono alle istruzioni del linguaggio macchina: l'assemblatore traduce ogni istruzione esecutiva in una istruzione in linguaggio macchina.
Le istruzioni esecutive comprendono istruzioni di **trasferimento** di dati (da memoria a registro e viceversa o da memoria a memoria), istruzioni **aritmetiche** e **logiche**, istruzioni di **confronto** e **diramazione** (più note come istruzioni di salto) e istruzioni per il richiamo di **sottoprogrammi**.

Le istruzioni **direttive**, chiamate anche pseudoistruzioni, danno delle indicazioni che l'assemblatore deve seguire durante la traduzione e non generano istruzioni in linguaggio macchina (si possono considerare tra le direttive anche le istruzioni dichiarative).

Linguaggi ad alto livello

I linguaggi di programmazione ad **alto livello**, oltre a permettere l'uso di nomi simbolici, offrono istruzioni indipendenti dal processore, che corrispondono a più istruzioni in linguaggio macchina, e risultano più comprensibili perché più vicine al linguaggio naturale.

È molto più facile realizzare programmi usando linguaggi ad alto livello piuttosto che il linguaggio Assembler; i programmi ottenuti risultano però in generale meno efficienti, cioè più lunghi e più lenti.

Nota

Può quindi essere opportuno programmare in Assembler quando la velocità del programma è un fattore essenziale.

L'evoluzione dei linguaggi di programmazione

I linguaggi di programmazione si possono classificare in **generazioni** in base al momento in cui si sono evoluti e alle loro caratteristiche.

- prima generazione: linguaggio macchina;
- seconda generazione: linguaggi Assembler;
- terza generazione: linguaggi ad alto livello generici;
- quarta generazione: linguaggi orientati a problemi particolari (rientrano in questa categoria linguaggi nati per risolvere classi particolari di problemi, come la gestione di data base, problemi di simulazione ecc.);
- quinta generazione: linguaggi logici e funzionali.

1.3 I PARADIGMI DI PROGRAMMAZIONE

I linguaggi di programmazione usano molti paradigmi di programmazione diversi.

Un **paradigma di programmazione** è un modello che permette di descrivere astrattamente l'algoritmo (cioè il metodo di soluzione di un problema).

I principali paradigmi di programmazione sono:

- il **paradigma procedurale**,
- il **paradigma dichiarativo**.

Il paradigma procedurale

Il **paradigma procedurale** descrive **come** deve essere risolto un problema.

La programmazione con il paradigma procedurale si può suddividere in:

- **programmazione imperativa**: si basa su esplicite richieste fatte all'esecutore del programma; la soluzione del problema è descritta come una sequenza di azioni che producono dei cambiamenti di stato nell'ambiente (come la modifica del valore delle variabili);

Nota: I linguaggi imperativi sono chiamati anche algoritmici.

Esempi di linguaggi imperativi: linguaggio macchina, Assembler, Pascal, C, Cobol.

- **programmazione orientata agli oggetti**: parte dal concetto di oggetto che descrive proprietà e azioni che un oggetto può compiere; la soluzione del problema è descritta dalle interazioni tra gli oggetti.

Nota: I linguaggi orientati agli oggetti si dicono puri se permettono di usare solo oggetti.

Esempi di linguaggi ad oggetti: Java, C++.

Programmazione basata sugli eventi

Sia nella programmazione imperativa che in quella orientata agli oggetti l'esecuzione può essere guidata dagli eventi, cioè alcune azioni possono essere eseguite quando si verificano determinati eventi, per esempio un'azione dell'utente, il caricamento del programma o il passare del tempo. Di solito la programmazione guidata dagli eventi è legata all'interfaccia grafica.

Le interfacce grafiche sono spesso realizzate tramite oggetti.

> Esempi di linguaggi basati sugli eventi: Visual Basic.

Il paradigma dichiarativo

Il **paradigma dichiarativo** descrive **che cosa** si vuole ottenere come soluzione e non come fare per raggiungerla; in questo caso è l'esecutore che è predisposto per seguire opportune strategie per determinare la soluzione.

Il paradigma dichiarativo comprende:

- **programmazione logica**: descrive i fatti e le relazioni tra i fatti e permette di ricavarne delle conseguenze;

> Esempi di linguaggi logici: Prolog.

- **programmazione funzionale**: il programma è costituito da un insieme di funzioni che elaborano liste di simboli;

> Esempi di linguaggi funzionali: Lisp.

- linguaggi di **markup**: usano dei codici (tag) per stabilire il ruolo degli elementi;

> Esempi di linguaggi di markup: HTML, XHTML, XML.

- linguaggi di **interrogazione di database**.

> Esempi di linguaggi di interrogazione di database: SQL.

1.4 I LINGUAGGI IMPERATIVI

Il vocabolario usato da ciascun linguaggio comprende delle parole riservate e delle parole che possono essere definite dal programmatore seguendo alcune semplici regole.
Le **parole riservate** permettono di definire i dati e di scrivere le istruzioni; le **parole definite dal programmatore** (o **identificatori**) sono in genere i nomi delle variabili e i nomi delle procedure da richiamare.

Le istruzioni normalmente sono semplici da ricordare e si avvicinano al linguaggio naturale (di solito la lingua inglese).

Le **istruzioni** si possono suddividere in:

- dichiarative,
- esecutive,
- di controllo.

Le istruzioni **dichiarative** permettono di definire i dati che si utilizzano; per definire una variabile bisogna stabilire il nome e il tipo di dati; il tipo di dati della variabile determina l'insieme dei valori ammessi e le operazioni eseguibili.

> Nei linguaggi debolmente tipizzati non è necessario definire il tipo di dati delle variabili. **Nota**

Le istruzioni **esecutive** comprendono quelle che si riferiscono ad operazioni effettive come operazioni di input/output che gestiscono lo scambio di dati tra utente e programma, operazioni di assegnazione che consentono di associare un valore ad una variabile, operazioni sui dati in base al loro tipo, per esempio operazioni aritmetiche sui dati numerici ecc.

Le istruzioni **di controllo** governano il funzionamento del programma; permettono di solito la codifica delle strutture fondamentali di programmazione (sequenziale, di selezione e iterativa) o almeno il trasferimento, in base al verificarsi di una condizione, ad una istruzione del programma che non sia quella successiva.
In genere è possibile anche definire procedure da richiamare nei punti desiderati, a volte anche in modo ricorsivo.

1.5 IL SOFTWARE DI UTILITÀ

L'EDITOR

L'**editor** è un programma di utilità che consente di inserire e memorizzare un testo e di modificarlo successivamente per effettuare aggiunte o correzioni.

L'elaborazione del testo viene effettuata in memoria centrale; il testo viene trasferito in un file su memoria di massa soltanto quando viene salvato il testo.
Il testo viene memorizzato carattere per carattere nel codice di rappresentazione specificato (per esempio ASCII).
In genere sono disponibili comandi per effettuare altre operazioni oltre all'inserimento e alla variazione del testo, come per esempio lo spostamento o la copia di parti di testo e la ricerca o sostituzione di stringhe di testo.

Alcuni editor, **orientati alla scrittura di programmi** in un determinato linguaggio di programmazione, possono guidare l'inserimento delle istruzioni presentando un elenco delle parole riservate che si possono utilizzare, evidenziando con colori diversi la funzione dei vari elementi, favorendo l'indentazione e talvolta effettuando un primo controllo formale sull'istruzione inserita.

Le istruzioni del programma vengono memorizzate in un file che contiene il **programma sorgente**, cioè il programma scritto nel linguaggio di programmazione.

Per scrivere il programma sorgente si può usare anche un programma di elaborazione testi, ma bisogna ricordarsi di salvare il file come solo testo. **Nota**

I PROGRAMMI TRADUTTORI

Il computer è in grado di eseguire soltanto istruzioni in linguaggio macchina; il programma codificato in un linguaggio di programmazione viene tradotto in linguaggio macchina da **programmi traduttori** chiamati assemblatori per la traduzione del linguaggio Assembler e compilatori o interpreti, secondo il modo in cui operano, per la traduzione dei linguaggi ad alto livello.

Per ogni linguaggio esiste un programma traduttore specifico; per alcuni esiste un interprete, per altri un compilatore, a volte sia un interprete che un compilatore.

Per esempio per i linguaggi Pascal e C/C++ ci sono dei compilatori, per il linguaggio Visual Basic, sebbene sia più diffuso l'utilizzo con l'interprete, esistono anche compilatori. Java ha un compilatore che traduce il programma sorgente in un linguaggio intermedio, che poi viene interpretato dalla Java Virtual Machine.

Assemblatori e compilatori creano un **programma oggetto**, traduzione del programma sorgente in linguaggio macchina**.**

Il programma oggetto deve essere linkato per produrre un programma eseguibile prima di poter essere eseguito.

L'assemblatore

L'assemblatore è un programma che **traduce** un programma sorgente scritto in linguaggio **Assembler** in un programma oggetto in **linguaggio macchina**.

L'assemblatore in genere opera in due fasi, dette **passate**. Nella prima passata controlla la sintassi delle istruzioni e costruisce la **tabella dei simboli**; la tabella dei simboli contiene tutti i nomi simbolici incontrati (nomi di dati ed etichette di istruzioni) e il corrispondente indirizzo, calcolato automaticamente.
Se durante la prima passata vengono riscontrati degli errori, l'assemblatore interrompe il suo lavoro e segnala gli errori.
Se non ci sono errori, l'assemblatore esegue la seconda passata durante la quale effettua la **traduzione** vera e propria e genera il **codice oggetto**. Per la traduzione utilizza la tabella dei simboli creata nella prima passata e una tabella delle istruzioni in cui ad ogni codice operativo simbolico corrisponde il codice operativo in binario.
La traduzione in linguaggio macchina riguarda solo le **istruzioni esecutive**; per ogni istruzione sostituisce il codice operativo simbolico con il corrispondente codice operativo in binario e sostituisce i nomi simbolici che rappresentano operandi o etichette con il corrispondente indirizzo, memorizzato nella tabella dei simboli nella prima passata (le due passate sono necessarie perché durante la traduzione si potrebbero incontrare nomi non ancora definiti e di cui perciò non è noto l'indirizzo).

L'area di memoria occupata dal programma oggetto è detta **spazio logico** e gli indirizzi del programma in questa fase sono detti indirizzi logici.
Se lo spazio logico è diverso dallo **spazio fisico**, cioè da quello in cui il programma viene eseguito, il programma è formato da codice **rilocabile**.
Se il programma deve già utilizzare indirizzi fisici di memoria allora è in codice **assoluto**.
Se gli indirizzi sono espressi in modo assoluto, per scrivere il programma bisogna conoscere l'indirizzo di memoria in cui verrà caricato per l'esecuzione; al contrario, se gli indirizzi sono espressi in modo rilocabile, non è necessario conoscere l'indirizzo di caricamento del programma che potrà variare da esecuzione a esecuzione o anche durante l'esecuzione stessa (programma rilocabile).

Il compilatore

Il compilatore è un programma che **traduce** un programma **sorgente** scritto in linguaggio ad alto livello in un programma **oggetto** in linguaggio macchina.

Il lavoro del compilatore si divide nelle fasi di **analisi** e di **sintesi**.
Il compilatore può operare in una o più passate; i compilatori a una passata sono più veloci.

L'**analisi** si divide in:

- **analisi lessicale** (eseguita dallo scanner), che individua le parole del vocabolario e costruisce la tabella dei simboli;
- **analisi sintattica** (eseguita dal **parser**), che controlla se le frasi sono scritte rispettando la sintassi del linguaggio cioè le regole della grammatica; durante questo processo viene creato l'albero sintattico (**parse tree**) relativo al programma;
- **analisi semantica**, che controlla la validità del significato degli elementi in base al contesto (per esempio la presenza delle istruzioni dichiarative necessarie, il tipo delle costanti e delle variabili in operazioni di assegnazione, la validità degli operatori in espressioni aritmetiche, la correttezza del numero e tipo dei parametri passati nelle chiamate a sottoprogrammi ecc.).

L'analisi permette di verificare la **correttezza** del programma.

Il compilatore individua e segnala tutti gli **errori formali**.
Se ci sono errori la traduzione non può avvenire.

Se non ci sono errori il compilatore passa alla fase di **sintesi** durante la quale effettua la **traduzione** vera e propria e genera il **codice oggetto**, di solito in formato rilocabile, e produce l'elenco dei simboli rilocabili.

La fase di sintesi può essere preceduta da una traduzione del programma in un **codice intermedio** e da una fase di **ottimizzazione** del codice, che permette di ottenere un programma più efficiente, cioè che utilizza meno risorse di calcolo (memoria e CPU).

La **generazione del codice oggetto** dipende dalla piattaforma utilizzata (processore e sistema operativo).

La fase di analisi di un compilatore è indipendente dalla piattaforma, ma la fase di generazione del codice deve essere adattata per ogni piattaforma. La possibilità di adattamento del compilatore a piattaforme diverse viene detta **portabilità**. Il passaggio della traduzione attraverso un linguaggio intermedio favorisce questa caratteristica.

Il programma oggetto prodotto dal compilatore deve essere **linkato** per produrre un programma eseguibile.

Se il programma oggetto non viene linkato, può essere eseguito soltanto in un ambiente collegato al compilatore, chiamato **modulo run-time**; il modulo run-time comprende il codice oggetto dei sottoprogrammi necessari per lo svolgimento di funzioni di servizio, come per esempio le operazioni di input/output.

Il linker collega al programma il codice oggetto necessario perché il programma possa essere eseguito in modo indipendente.

Il linker

Il **linker** trasforma il programma **oggetto** in programma **eseguibile**.

Il programma oggetto realizzato dall'assemblatore o da un compilatore non può essere eseguito in modo autonomo (anche se può essere eseguito per esempio all'interno di un ambiente integrato). Per creare un programma eseguibile bisogna che al programma oggetto siano collegati dei moduli di sistema (**supporto run-time**) che rendono disponibili delle funzioni necessarie al programma. Il collegamento dei moduli di sistema è effettuato dal linker.

Il linker permette anche di collegare **più moduli oggetto** per creare un unico programma eseguibile. Il programma può essere suddiviso in moduli che si occupano di particolari funzioni. Ogni modulo può dichiarare variabili o funzioni che possono essere usate dagli altri moduli; questi elementi sono detti **pubblici** perché possono essere usati da tutti. Ogni modulo può usare variabili o funzioni definite in altri moduli; questi elementi sono detti simboli **esterni** perché, appunto, sono definiti esternamente.
Tutti i moduli possono essere scritti nello stesso linguaggio, oppure in **linguaggi diversi** (purché vengano rispettate opportune convenzioni per la definizione di elementi pubblici o l'utilizzo di elementi esterni); per ogni modulo deve essere creato il programma oggetto con l'assemblatore o il compilatore opportuno.
Il linker collega i diversi moduli del programma, creando un unico programma eseguibile; il linker provvede a calcolare l'indirizzo degli elementi esterni e a sostituirlo nei punti in cui gli elementi esterni sono utilizzati.
L'elenco dei simboli esterni e dei simboli pubblici definiti nel modulo ed esportati è fornito al linker dall'assemblatore o dal compilatore.

Il linker collega i vari moduli oggetto e crea un programma eseguibile supponendo che il programma sia collocato in un'area di memoria predefinita (**spazio logico**) che di solito non coincide con l'area di memoria in cui il programma viene effettivamente eseguito (spazio fisico).

Se il collegamento è fatto prima dell'esecuzione del programma si parla di collegamento **statico**. Il collegamento può anche essere **dinamico**; in questo caso il programma eseguibile coincide con l'insieme dei programmi oggetto e la risoluzione dei riferimenti esterni avviene a tempo di esecuzione.
In pratica il collegamento dinamico permette di collegare un modulo al programma in fase di esecuzione, solo quando (e se) necessario.

Le DLL

Le DLL (Dynamic Linking Library o Librerie a collegamento dinamico) usano il collegamento dinamico.
Le DLL sono file che esportano una serie di funzioni; un programma può caricare una DLL e usarne le funzioni; una DLL non può mai essere eseguita indipendentemente, ma deve sempre essere attivata da un programma o da un'altra DLL.
L'utilizzo delle DLL offre vari vantaggi:

- le DLL possono essere caricate solo quando occorre, quindi il programma eseguibile occupa meno spazio in memoria, e solo quello che gli è effettivamente necessario;
- le DLL possono essere modificate e ricompilate senza dover ricompilare il programma che le utilizza;
- se una DLL è usata da più programmi viene caricata in memoria una volta sola.

Il caricamento del programma

Il programma **eseguibile** (spazio logico creato dal linker) può essere memorizzato in un file su disco e richiamato quando si vuole per l'esecuzione.

> Al momento dell'esecuzione il programma deve essere portato in memoria centrale, in uno **spazio fisico**.

Lo spazio logico può non coincidere con lo spazio fisico; questa distinzione è possibile se gli indirizzi logici sono in forma **rilocabile** e non assoluta.

Lo spazio fisico viene assegnato dal **gestore della memoria**, un modulo del sistema operativo, che amministra lo spazio disponibile.

Distinguendo spazio logico e spazio fisico è possibile eseguire il programma ogni volta in uno spazio diverso o addirittura modificare più volte lo spazio fisico durante l'esecuzione (**caricamento dinamico**).

> Gli accessi alla memoria del programma in esecuzione possono avvenire solo specificando **indirizzi fisici**.

Gli indirizzi logici, che fanno riferimento allo spazio logico, devono essere trasformati in indirizzi che fanno riferimento alla locazione di memoria occupata (rilocazione).

> La **rilocazione** in pratica consiste nel sostituire ogni indirizzo logico con uno fisico.

La rilocazione può essere **statica** se viene eseguita su tutto il programma prima del caricamento in memoria o **dinamica** se avviene dinamicamente per ogni indirizzo durante l'esecuzione.

L'interprete

L'**interprete** è un programma che traduce le istruzioni del linguaggio ad alto livello in linguaggio macchina, una per una, al momento dell'esecuzione.
L'esecuzione può iniziare subito dopo la scrittura del programma sorgente.
L'interprete permette di lavorare in un ambiente interattivo per la realizzazione dei programmi.

Perché un programma possa essere eseguito, l'interprete e il programma devono essere contemporaneamente in memoria centrale.

> L'**interprete** procede istruzione per istruzione, sul programma sorgente: **controlla** l'istruzione, la **traduce** in linguaggio macchina e la **esegue**.

Le istruzioni di un ciclo vengono tradotte ogni volta che il ciclo viene percorso.

Se un'istruzione (per esempio di un ramo di una istruzione condizionale) non viene eseguita, non viene controllata e potrebbe presentare degli errori che non vengono individuati.

Nota

L'interprete non crea un programma oggetto.

Se l'interprete riscontra un **errore formale** nella scrittura di una istruzione, segnala l'errore e interrompe l'esecuzione del programma.

Confronto tra compilatori e interpreti

Lo **sviluppo del programma** è più veloce con un interprete:

- con un compilatore, ad ogni modifica, prima di eseguire il programma, bisogna compilarlo e linkarlo;
- con un interprete si possono fare modifiche molto velocemente; l'ambiente di lavoro è interattivo.

L'**esecuzione** del programma è più veloce per un programma compilato.

- il compilatore traduce tutto il programma creando il programma oggetto; l'esecuzione avviene sul programma eseguibile già predisposto; il programma oggetto e il programma eseguibile possono essere memorizzati e riutilizzati;
- l'interprete prima di eseguire ogni istruzione la deve tradurre.

L'**occupazione di memoria** è minore con un programma compilato:

- con un compilatore, dopo il linkaggio il programma eseguibile può essere eseguito in modo autonomo;
- l'interprete deve essere in memoria insieme al programma da eseguire.

La garanzia di **correttezza** del programma (per quanto riguarda gli errori formali) è maggiore con un compilatore

- il compilatore controlla tutto il programma sorgente e produce il programma oggetto solo se non ci sono errori formali;
- l'interprete controlla un'istruzione solo mentre la esegue.

La **segretezza** è maggiore con un programma compilato

- con un compilatore si deve distribuire solo il programma eseguibile;
- con un interprete bisogna distribuire il programma sorgente.

La **portabilità** è maggiore con un interprete:

- con un compilatore bisogna produrre un programma eseguibile diverso per ogni piattaforma su cui il programma dovrà essere eseguito;

- con un interprete, basta che sia disponibile un interprete per la piattaforma.

La **comodità per l'utente** finale è maggiore con un compilatore:

- per un programma prodotto con un compilatore per l'esecuzione non è richiesto alcun software
- per un programma interpretato l'utente deve avere a disposizione l'interprete necessario.

Tabella 1.1 *Vantaggi e svantaggi di compilatori e interpreti*

	Vantaggi	Svantaggi
Compilatore	■ maggior velocità di esecuzione, perché la traduzione è già stata fatta in precedenza; ■ risparmio di memoria perché durante l'esecuzione non è necessario che il compilatore sia presente in memoria; ■ rileva tutti gli errori formali; ■ consente la segretezza del programma sorgente; ■ non è necessario codice aggiuntivo per l'esecuzione del programma.	■ tempi di creazione del programma più lunghi (a ogni modifica bisogna compilare e linkare); ■ minore portabilità: bisogna creare un programma eseguibile diverso per ogni piattaforma.
Interprete	■ semplicità di messa a punto dei programmi poiché si lavora in ambiente interattivo; ■ maggior portabilità dei programmi: basta che sia disponibile un interprete per la piattaforma.	■ maggiore occupazione di memoria perché durante l'esecuzione del programma, anche l'interprete deve essere caricato in memoria centrale; ■ l'esecuzione risulta più lenta poiché le istruzioni devono essere tradotte ogni volta che vengono eseguite; ■ non dà garanzia di correttezza sintattica: possono restare degli errori perché vengono controllate soltanto le istruzioni effettivamente eseguite; ■ non consente la segretezza del codice sorgente; ■ è necessario avere l'interprete per eseguire il programma.

Conviene usare un interprete o un compilatore?

L'ideale è avere a disposizione un interprete per lo sviluppo del programma, in modo da poter utilizzare l'ambiente interattivo per effettuare velocemente prove e correzioni, e un compilatore per creare un programma eseguibile, quando si è raggiunta la versione definitiva, per ottenere un programma più efficiente, perché l'utente possa eseguire il programma senza dover installare altro software e per non dover distribuire il codice sorgente.

Ambienti di programmazione IDE e RAD

Gli ambienti di sviluppo **IDE** (Integrated Development Environment, o ambienti di sviluppo intergrati) sono ambienti di programmazione con interfaccia grafica che offrono tutti gli strumenti

necessari all'implementazione di un programma: editor, interpete o compilatore e linker, debugger ecc.

RAD (Rapid Application Development) è un approccio per rendere più rapido lo sviluppo di applicazioni tramite modelli e strumenti per il disegno dell'interfaccia grafica o addirittura la produzione automatica del codice (per esempio tramite Wizard che generano il codice in base a scelte impostate dal programmatore).

VERIFICA

QUESTIONARIO

1. Che cos'è un linguaggio?
2. Che cos'è un linguaggio di programmazione?
3. Quali linguaggi di programmazione il computer è in grado di comprendere?
4. Quali sono le caratteristiche della programmazione in linguaggio macchina?
5. Cosa significa linguaggio a basso livello?
6. Qual è il formato delle istruzioni Assembler?
7. Quali sono le categorie di istruzioni Assembler?
8. Quali sono i vantaggi dei linguaggi ad alto livello?
9. Che cos'è un paradigma di programmazione?
10. Che cos'è la programmazione imperativa?
11. Che cos'è la programmazione ad oggetti?
12. Che cos'è la programmazione guidata dagli eventi?
13. Che cos'è il paradigma dichiarativo?
14. Quali sono le categorie di istruzioni dei linguaggi imperativi?
15. Che cosa fanno i programmi traduttori?
16. Che cosa fa il linker?
17. Qual è la differenza tra collegamento statico e collegamento dinamico?
18. Che cosa sono le DLL?
19. Qual è la differenza tra caricamento statico e caricamento dinamico?
20. Che cos'è la rilocazione?

TEST

1 ..... *Indicare se le seguenti affermazioni sono vere o false.*

V	F	
☐	☐	1) Il linguaggio macchina dispone di istruzioni dichiarative ed esecutive.
☐	☐	2) Il linguaggio Assembler dispone di istruzioni dichiarative ed esecutive.
☐	☐	3) Per realizzare programmi più efficienti conviene scriverli con un linguaggio ad alto livello.
☐	☐	4) Per realizzare programmi in modo più semplice conviene scriverli con un linguaggio ad alto livello.
☐	☐	5) Il linguaggio C è un linguaggio imperativo.
☐	☐	6) Il linguaggio C++ è un linguaggio della quarta generazione.

V	F	
☐	☐	7) Il linguaggio HTML è un linguaggio logico.
☐	☐	8) I linguaggi logici usano il paradigma dichiarativo.
☐	☐	9) I linguaggi imperativi dispongono di istruzioni dichiarative.
☐	☐	10) L'assemblatore traduce in istruzioni del linguaggio macchina tutte le istruzioni del linguaggio Assembler.
☐	☐	11) L'assemblatore assegna un indirizzo ad ogni nome simbolico.
☐	☐	12) L'assemblatore traduce il programma sorgente in programma eseguibile.
☐	☐	13) Lo spazio fisico è l'area di memoria in cui viene effettivamente eseguito il programma.

2 ..... *Dire se le seguenti affermazioni si riferiscono al compilatore, all'interprete, o a entrambi.*

Compilatore	Interprete	
☐	☐	crea un programma oggetto;
☐	☐	crea un programma eseguibile;
☐	☐	deve essere presente in memoria durante l'esecuzione del programma;
☐	☐	lo sviluppo del programma è veloce;
☐	☐	l'esecuzione del programma è veloce;
☐	☐	la traduzione delle istruzioni dei cicli viene ripetuta più volte;
☐	☐	anche dopo l'esecuzione, nel programma possono restare errori formali;
☐	☐	consente la segretezza del programma sorgente.

3 ..... *Segnare tutte le affermazioni esatte riguardo al linker.*

- ☐ il linker crea un programma oggetto;
- ☐ il linker crea un programma eseguibile;
- ☐ il linker può collegare moduli oggetto prodotti con assemblatori e compilatori diversi;
- ☐ il collegamento statico avviene prima dell'esecuzione;
- ☐ il collegamento dinamico avviene prima dell'esecuzione;
- ☐ il linker effettua la rilocazione;
- ☐ il linker suppone che il programma sia collocato in uno spazio logico;
- ☐ il linker si occupa del caricamento del programma in memoria.

Esercizi

1 ..... Dato l'alfabeto {0, 1} formare tutte le possibili parole di 3 caratteri.

2 ..... Definire sia con la notazione di Backus Naur che con i diagrammi sintattici la grammatica per:

- la rappresentazione dei numeri esadecimali;
- la rappresentazione delle targhe automobilistiche (due lettere maiuscole, 3 cifre, altre due lettere maiuscole);
- la rappresentazione dei numeri decimali in notazione esponenziale.

ALGORITMI E PROGRAMMI

2

PREREQUISITI

- Paradigmi e linguaggi di programmazione
- Programmi di utilità (editor, compilatori e interpreti, linker)

OBIETTIVI

CONOSCENZE

- Problemi e loro soluzione
- Algoritmi
- Dati
- Istruzioni
- Pseucodifica e flow-chart
- Programmazione strutturata
- Problem solving
- Computabilità
- Interfaccia programma/utente
- Fasi di realizzazione di un programma

ABILITÀ/CAPACITÀ

- Comprendere l'uso di pseudocodifica e flow-chart per la descrizione di algoritmi
- Implementare un programma
- Controllare la correttezza di un programma

2.1 IL CONCETTO DI PROBLEMA

Per **problema** si può intendere qualsiasi questione o quesito proposto per cui è necessario trovare una soluzione.

La **descrizione di un problema** di solito comprende:

- la situazione iniziale (dati del problema),
- che cosa si desidera ottenere (risultati),
- le risorse a disposizione.

La **soluzione** del problema (o **processo risolutivo**) è la descrizione del procedimento da seguire per ottenere i risultati desiderati.

In base alle risorse a disposizione si possono avere processi risolutivi molto diversi tra loro.

Per ottenere realmente una soluzione ci deve essere un **esecutore** che partendo dalla situazione iniziale e seguendo le istruzioni ottiene il risultato desiderato.

Problema: Somma di due numeri.
Dati due numeri interi si desidera ottenere il risultato della somma dei due numeri.
Le risorse a disposizione potrebbero essere: carta e penna, oppure una calcolatrice, oppure il computer.
Con carta e penna si dovrebbero incolonnare i dati ed eseguire l'operazione cifra per cifra, mentre con la calcolatrice basterebbe digitare le cifre e scegliere il simbolo di operazione. Con il computer si può ottenere una soluzione al problema se si dispone di un programma adeguato.

Problema: Disegno della pianta di una casa.
Conoscendo le misure delle stanze, la disposizione delle stanze, di porte e finestre ecc., si desidera ottenere una rappresentazione grafica della pianta di una casa.
Le risorse a disposizione potrebbero essere: carta, matita e righello, oppure un computer dotato di un programma di disegno (programma di disegno vettoriale o CAD). In entrambi i casi si suppone che l'esecutore sia una persona che disegna la pianta manualmente o col programma di grafica.
Si potrebbe anche desiderare che, date le misure e la relazione tra le stanze, un computer fornisca automaticamente il disegno desiderato; ciò è possibile se si dispone di un programma adeguato.

Problema: Accensione e spegnimento automatico della luce.
Si vuole fare in modo che la luce in una stanza venga accesa quando nella stanza entra qualcuno e venga spenta quando nella stanza non c'è più nessuno, oppure quando c'è luce naturale sufficiente, cioè, sapendo se nella stanza c'è qualcuno oppure no, e conoscendo le condizioni di luminosità, si desidera fare in modo che la luce venga accesa o spenta quando necessario.
Le risorse a disposizione potrebbero essere: una persona che sta accanto all'interruttore e decide quando accendere o spegnere, oppure un computer.
Il computer deve essere dotato di un programma in grado di decidere se accendere o spegnere la luce, ma deve anche essere in grado fisicamente di valutare le condizioni, in modo da poter prendere una decisione, e di azionare l'interruttore: al computer devono essere collegati dei sensori (per esempio una telecamera), per poter stabilire la presenza di persone nella stanza (per esempio in base al fatto che ci siano dei cambiamenti

dell'immagine nel tempo) e le condizioni di luminosità, e un attuatore, cioè un dispositivo in grado di far scattare l'interruttore.
Il problema però non è ancora definito in modo preciso perché non è stabilito chiaramente che cosa significa dire che la luminosità è sufficiente: bisogna stabilire un modo oggettivo per deciderlo.

2.2 IL COMPUTER COME RISORSA PER LA SOLUZIONE DI PROBLEMI

Il computer permette di risolvere in modo **automatico** moltissimi problemi, ma solo se dispone del programma adeguato.

Per poter risolvere con il computer un certo problema bisogna fornire al computer stesso il programma adatto.

Bisogna prima di tutto sapere come si risolve manualmente il problema: bisogna trovare una soluzione al problema e descrivere l'elenco delle istruzioni da eseguire di volta in volta per raggiungere la soluzione.
Il processo risolutivo da seguire deve essere descritto in modo ben preciso, senza lasciare nessuna ambiguità.

Il **computer** fa da **esecutore** del processo risolutivo.

Il computer può soltanto velocizzare le operazioni e padroneggiare una grande quantità di dati, ma esegue soltanto quello che gli viene detto esplicitamente; non è in grado di prendere delle iniziative in modo autonomo, quindi deve essere istruito in modo preciso.

Il **processo risolutivo** di un problema deve essere descritto in modo **formale** e richiede la descrizione dei dati relativi al problema e delle azioni da eseguire per risolvere il problema.

I **dati** che riguardano il problema sono tutte le informazioni disponibili all'inizio e quelle che si desiderano ottenere come soluzione del problema.
Le **azioni** che possono essere eseguite nel processo risolutivo sono le operazioni che il computer è in grado di eseguire.

Il computer è in grado di risolvere soprattutto problemi di **elaborazione delle informazioni**.
I dati però non sono solo numerici e le operazioni non sono solo quelle aritmetiche.

Per esempio è possibile risolvere problemi relativi all'elaborazione di immagini o suoni (per esempio ritocco fotografico) o problemi di simulazione (di traffico, di sistemi naturali ecc.).

Le azioni che il computer è sempre in grado di eseguire sono le operazioni su numeri e caratteri. Con apposito hardware può eseguire anche azioni pratiche (**robot**); servono opportuni **attuatori** per eseguire azioni e **sensori** per rilevare automaticamente dati dall'ambiente.

2.3 ALGORITMO E PROGRAMMA

La descrizione del processo risolutivo (dati necessari e operazioni da eseguire) costituisce l'algoritmo.

L'**algoritmo** è un procedimento che permette di ottenere dei risultati (dati in uscita o di **output**) partendo da alcuni dati iniziali (dati di ingresso o di **input**).

Uno stesso problema può essere risolto con algoritmi diversi.

Due **algoritmi** si dicono **equivalenti** se per ogni combinazione di dati iniziali producono sempre gli stessi risultati, pur eseguendo istruzioni diverse.

Il vantaggio di un algoritmo rispetto all'altro può dipendere per esempio dal risparmio di tempo o di memoria necessari per l'esecuzione o da una maggior comprensibilità.

Definizione di algoritmo

Un algoritmo è la **descrizione del processo risolutivo** di un problema; si compone di **dati** e di **istruzioni**.

L'algoritmo **deve**:

- essere composto da istruzioni che l'esecutore dell'algoritmo è realmente in grado di eseguire;
- terminare sempre in un tempo finito;
- produrre dei risultati che si possano descrivere (cioè deve avere un effetto osservabile);
- essere **deterministico**: i risultati devono essere sempre gli stessi, partendo dagli stessi dati iniziali, cioè non devono dipendere da qualcosa di casuale;
- essere **generale**: non deve risolvere un singolo caso particolare, ma tutta una serie di problemi dello stesso tipo.

È importante distinguere tra descrizione dell'algoritmo ed esecuzione dell'algoritmo.

La **descrizione** dell'algoritmo è la lista delle istruzioni che descrivono come raggiungere la soluzione, che comprende tutti i casi possibili per trattare una classe di problemi.

L'**esecuzione** dell'algoritmo è l'applicazione ad un caso particolare, partendo da dati iniziali noti. L'esecuzione è chiamata anche **processo** e l'esecutore **processore**.

Il **processo di esecuzione** di solito è variabile: le istruzioni da eseguire dipendono dai dati iniziali del particolare caso in esame, e l'algoritmo descrive un insieme di sequenze di esecuzione diverse.

L'algoritmo per poter essere eseguito al computer deve essere implementato in un **linguaggio di programmazione**.

Il **programma** si ottiene codificando l'algoritmo in un linguaggio di programmazione, cioè scrivendo le istruzioni dell'algoritmo secondo la sintassi del linguaggio scelto.

2.4 LA DESCRIZIONE DELL'ALGORITMO

La descrizione del processo risolutivo può risultare molto diversa in base al **paradigma di programmazione** scelto.
La descrizione dei dati e delle azioni per risolvere il problema dipende dal paradigma di programmazione utilizzato e, per linguaggi che usano lo stesso paradigma, almeno parzialmente dal linguaggio scelto, infatti potrebbero essere disponibili per esempio tipi o strutture di dati diverse.

Ci sono però alcuni **concetti comuni** a tutti i paradigmi, per esempio i concetti fondamentali della **descrizione dei dati**.

Nel seguito si fa riferimento al **paradigma imperativo**.
Molti concetti del paradigma imperativo si applicano anche alla programmazione a oggetti per quanto riguarda le interazioni (cioè la scrittura dei metodi degli oggetti).
Per questi paradigmi sono fondamentali le istruzioni di **assegnazione** e le **strutture di controllo**.

2.5 I DATI NELL'ALGORITMO

L'algoritmo non è soltanto un insieme di istruzioni; una componente fondamentale dell'algoritmo è costituita dai **dati**; quindi una fase essenziale della stesura dell'algoritmo è la scelta dei dati da utilizzare.

Per poter stabilire quali dati utilizzare è importante conoscere:

- la differenza tra dati di input, di output e di lavoro;
- la differenza tra costanti e variabili;
- i tipi di dati disponibili.

Dati di input e di output

I **dati** permettono la **comunicazione** tra l'utente del programma e l'esecutore.

L'esecutore applica l'algoritmo ai dati iniziali forniti dall'utente e produce dei risultati da restituire all'utente stesso (il programma viene realizzato proprio allo scopo di ottenere tali risultati).

I dati necessari alla soluzione di un problema si possono suddividere in dati di **input** e dati di **output**.

La suddivisione viene fatta in base al verso di comunicazione dei dati tra utente ed esecutore, in riferimento all'esecutore (o in modo più impreciso al programma).

I dati di **input** sono i dati che l'utente deve fornire al programma per l'esecuzione.

I dati di input devono essere noti al momento dell'esecuzione; non ha invece importanza conoscerli al momento della descrizione dell'algoritmo, anzi l'algoritmo non si deve basare sul caso particolare dei valori noti ma deve essere generale, cioè valido per qualsiasi altro insieme di valori di input.

I dati di **output** sono il risultato dell'elaborazione, cioè i dati che il programma restituisce all'utente.

I dati di output costituiscono lo scopo del programma.

Alcuni dati possono essere sia di input che di output (per esempio un testo modificato in un elaboratore di testi).

I dati **interni** o **di lavoro** sono altri dati, utilizzati all'interno dell'algoritmo, predisposti dal programmatore, ma di nessuna importanza per l'utente del programma.
I dati di lavoro non sono visibili all'esterno ma sono usati solo per l'elaborazione (sono trasparenti per l'utente).

Costanti e variabili

I dati che descrivono un problema si possono suddividere in **costanti** e **variabili**.

I dati possono essere costanti o variabili in base alla possibilità di cambiare o meno valore nel tempo.

I dati **costanti** sono dati che hanno un valore predefinito, che non cambia.
I dati **variabili** possono cambiare valore.

Calcolo del perimetro di un quadrato.
I dati necessari sono la misura del lato e il numero dei lati.
Il numero dei lati è 4 ed è una costante, non cambia mai, per tutti i quadrati.
La misura del lato è variabile e cambia da un quadrato all'altro.

Calcolo del perimetro di un poligono.
Anche in questo caso i dati necessari sono la misura del lato e il numero dei lati.
In questo problema però anche il numero dei lati è variabile, perché si possono considerare poligoni con numero di lati diverso.

Una **variabile** è un dato che viene modificato durante l'elaborazione o da un'elaborazione ad un'altra.

In qualche caso si può considerare costante anche un valore che non cambia durante l'elaborazione, anche se può cambiare per elaborazioni diverse.

Per esempio le aliquote IVA si possono considerare costanti, anche se possono variare da un anno all'altro in base a variazioni di legge.

I valori che possono essere assegnati a costanti e variabili dipendono dal **tipo di dati** a cui appartengono.

Tipi di dati

I tipi di dati utilizzabili dipendono dal **paradigma di programmazione**.

Nella **programmazione orientata agli oggetti** i dati sono sempre rappresentati da **oggetti**.
Nel **paradigma logico** i dati sono **fatti**.
Nel **paradigma funzionale** i dati sono **liste di simboli**.

Nel **paradigma imperativo** i dati possono essere di molti tipi diversi; si possono suddividere in:

- **dati semplici** (come dati numerici, alfanumerici o booleani);
- **strutture di dati**, cioè raggruppamenti di dati con una certa organizzazione che possono essere considerati come un unico oggetto o come insiemi dei singoli dati; è possibile accedere ad ogni dato che compone la struttura mediante modalità che dipendono dal tipo di struttura considerato.

  Esempi di strutture di dati sono array, matrici, record, file ecc.

I tipi di dati **semplici standard** (cioè quelli normalmente disponibili) sono:

- **numerico**: un dato è numerico se è formato soltanto dalle cifre decimali, il segno e la virgola decimale (o il punto decimale secondo la notazione usata); sui dati numerici sono definite tutte le operazioni aritmetiche ed operazioni di confronto; i dati numerici di solito si distinguono in numeri **interi** e numeri **reali**;
- **alfanumerico**: un dato alfanumerico può contenere cifre, caratteri alfabetici o caratteri speciali; i caratteri disponibili dipendono dal tipo di **codifica** usata per la memorizzazione e possono essere confrontati in base al codice corrispondente.

- in particolare spesso sono disponibili le **stringhe** di caratteri alfanumerici; per le stringhe sono solitamente definite operazioni di concatenazione o estrazione di parti e le operazioni di confronto;
- **booleano**: un dato booleano può assumere solamente i valori *vero* o *falso*; sui dati booleani sono definite le operazioni logiche *not*, *and* e *or*.

Nota

Non tutti i linguaggi permettono l'utilizzo di dati booleani.

Normalmente non tutti i valori di un tipo di dati sono utilizzabili in un algoritmo.
La rappresentazione di un tipo di dati è finita, quindi è possibile utilizzare solo un **intervallo limitato** di dati.
Inoltre, in base al problema, potrebbe essere ammissibile solo un certo insieme di valori, **dominio del problema**.

> In un algoritmo per la media dei prezzi il dominio dei prezzi è costituito da numeri reali positivi con due cifre decimali.

> Per la media dei voti presi da uno studente il dominio dei voti è costituito dai voti da 1 a 10.

Il dominio può essere costituito da un intervallo dei valori di un tipo (**subrange**) o da un elenco di valori (**enumerazione** dei valori del dominio).

> Per un algoritmo che lavora su stringhe il dominio potrebbe essere costituito da tutte le stringhe con un certo numero di caratteri o solo da determinate stringhe (per esempio i nomi dei mesi).

In alcuni casi è necessario stabilire dei **vincoli** ulteriori.

> In una data il numero massimo del giorno dipende dal mese considerato.

È importante definire il **dominio** esatto dei dati e i relativi **vincoli** in modo che si possano effettuare dei **controlli** sull'inserimento dei dati o sui valori che i dati assumono durante l'elaborazione.

2.6 LE ISTRUZIONI DELL'ALGORITMO

Le istruzioni di un algoritmo possono richiedere effettivamente l'esecuzione di una operazione (come le istruzioni per l'esecuzione di operazioni aritmetiche o le istruzioni di input/output), o possono servire soltanto per controllare il flusso di esecuzione del programma.
Le istruzioni del primo tipo vengono chiamate **istruzioni effettive**, quelle del secondo tipo **istruzioni di controllo**.

Le **istruzioni** o **strutture di controllo** permettono di governare l'esecuzione delle operazioni effettive.

La forma delle istruzioni dipende dal **paradigma di programmazione**.

Nel **paradigma procedurale** le strutture di controllo fondamentali sono:

- la **struttura sequenziale**: le istruzioni sono eseguite in sequenza una dopo l'altra;
- la **struttura di selezione**: più istruzioni sono eseguite in alternativa;
- la **struttura iterativa**: una serie di istruzioni viene ripetuta più volte.

Anche nella **programmazione a oggetti** sono fondamentali queste strutture: gli oggetti comunicano tra di loro usando istruzioni descritte in questo modo.

Nella **programmazione guidata dagli eventi** le istruzioni sono raggruppate in gestori evento eseguiti al verificarsi dell'evento.

Nella **programmazione imperativa** tradizionale, non guidata dagli eventi, le istruzioni sono eseguite in sequenza, una dopo l'altra, a partire dalla prima.

2.7 REALIZZAZIONE DI ALGORITMI E PROGRAMMI

Domande fondamentali da porsi per realizzare un algoritmo

- Quali sono gli obiettivi che si vogliono raggiungere? Quali sono le funzioni che il programma deve svolgere?
- Il programma viene realizzato per ottenere un certo risultato; quale output si vuole ottenere dal programma?
- Quali dati è necessario conoscere, cioè qual è l'input del programma?
- Il programma, per trattare i dati, usa delle variabili; quali variabili servono? Quali sono invece i valori costanti?
- Per ottenere i dati di output partendo dai dati di input si deve eseguire un procedimento; qual è l'algoritmo da seguire (in base al paradigma di programmazione che si decide di utilizzare)?

Inoltre è importante stabilire il tipo di **interazione** che l'utente deve avere con il programma (**interfaccia** tra programma e utente).

Per la realizzazione del **programma** bisogna anche tenere conto del tipo di **ambiente** in cui il programma deve operare (risorse hardware, software, umane ecc.). Le risorse disponibili possono influenzare la realizzabilità del programma; è necessario quindi operare delle scelte ed eventualmente fissare dei limiti di validità.

La **portabilità** di un programma è la possibilità di eseguire lo stesso programma su piattaforme diverse senza doverlo modificare (o comunque con pochissime modifiche).

I programmi scritti in un linguaggio sono trasportabili da una piattaforma all'altra se in ogni piattaforma è disponibile il compilatore o l'interprete per quel linguaggio.

2.8 METODI DI DESCRIZIONE DEGLI ALGORITMI

I metodi per la descrizione di dati e algoritmi possono variare in base al **paradigma di programmazione** usato.

Una metodologia molto usata soprattutto per la **programmazione a oggetti** è **UML**.

Per il **paradigma imperativo** si usano:

- la **pseudocodifica** (detta anche **linguaggio di progetto** o pseudolinguaggio) in cui le istruzioni vengono descritte in un linguaggio molto simile al linguaggio naturale;

- i **flow-chart** (chiamati anche **diagrammi a blocchi** o diagrammi di flusso del programma) che usano dei simboli grafici.

Nota

I flow-chart sono ormai in disuso perché possono portare a realizzare programmi non strutturati; infatti molti flow-chart per essere codificati così come sono, richiedono l'uso di istruzioni di salto che tendono, in seguito, a rendere illeggibile un programma.

Istruzioni in pseudocodifica

inizio ... fine	inizio e fine di un programma o di una procedura;
variabile ← espressione	assegnazione;
leggi variabile	istruzioni di input;
scrivi stringa o variabile	istruzioni di output;
se condizione **allora** istruzioni1 **altrimenti** istruzioni2 **fine se**	struttura di selezione;
mentre condizione istruzioni **fine mentre**	**ripeti** istruzioni **finché** condizione strutture cicliche;
nomeProcedura	richiamo di una procedura.

L'uso della pseudocodifica permette di descrivere l'algoritmo in modo **generale** senza essere troppo legati a uno specifico linguaggio di programmazione.
Lo stesso algoritmo, descritto tramite la pseudocodifica, può essere implementato in diversi linguaggi di programmazione (per lo stesso paradigma di programmazione).

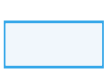
Per rendere più comprensibile la pseudocodifica (e anche il codice) si può usare l'**indentazione**, cioè scrivere le istruzioni con opportuni rientri a sinistra, come si vedrà meglio nei prossimi capitoli.

I flow-chart

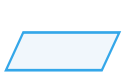
Per disegnare i flow-chart si utilizzano blocchi di varie forme, ognuno con un particolare significato, collegati da frecce che indicano la sequenza di esecuzione.

I blocchi più usati sono:

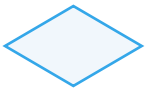
inizio o fine di un programma o di una procedura;



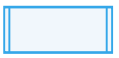
operazione effettiva come una assegnazione o un'operazione aritmetica;



istruzione di input/output;



test su una condizione; ha due uscite che indicano i valori di verità *vero* e *falso*;



richiamo di una procedura;

connettori per interrompere il flow-chart e farlo continuare in un altro punto.

2.9 LA PROGRAMMAZIONE STRUTTURATA

Perché l'algoritmo sia descritto in modo facilmente comprensibile, deve contenere ad ogni livello di dettaglio soltanto strutture di controllo che appartengono alle tre categorie: sequenziale, di selezione, iterativa (si parla in tal caso di **programmazione strutturata**).
Gli schemi che rappresentano queste strutture si chiamano **schemi di composizione fondamentali**.

La programmazione strutturata permette di rendere più semplice la scrittura e soprattutto la manutenzione dei programmi.

In precedenza gli algoritmi venivano descritti facendo largo uso di **istruzioni di salto** (istruzioni *goto*) che indicavano l'istruzione successiva da cui proseguire l'esecuzione; le istruzioni di salto si intrecciavano in modo da rendere molto difficile la comprensione del flusso di esecuzione complessivo.
La programmazione strutturata in pratica bandisce l'utilizzo delle istruzioni di salto e richiede una "struttura" più chiara dell'algoritmo anche nella mente del programmatore.

Il fatto che si possa realmente descrivere ogni algoritmo usando soltanto gli schemi di composizione fondamentale è garantito da un teorema, chiamato **teorema di Bohm-Jacopini** (o **teorema della programmazione strutturata**).

Il **teorema di Bohm-Jacopini** afferma in pratica che qualsiasi algoritmo non strutturato, che utilizza cioè delle istruzioni di salto all'interno del programma, può essere trasformato in un algoritmo strutturato, usando eventualmente delle variabili aggiuntive, chiamate di solito flag o segnalatori.

I linguaggi nati con istruzioni adatte a codificare gli schemi di composizione fondamentale vengono detti **linguaggi strutturati**.

Pascal, C/C++, Visual Basic e Java sono linguaggi strutturati. **Nota**

2.10 LA RICERCA DELLA SOLUZIONE

Per descrivere il processo risolutivo bisogna sapere come risolvere il problema.
Se si sa risolvere manualmente il problema, il programma può aiutare a farlo in modo automatico.

Individuare il processo risolutivo può non essere una cosa semplice.
Alcuni problemi possono addirittura non essere risolvibili, cioè non ammettere soluzioni.
Altri, anche se sono teoricamente risolvibili, possono richiedere un tempo elevatissimo per essere risolti.
Anche se il problema è risolvibile la soluzione potrebbe non essere immediata.
Per uno stesso problema possono invece esserci più soluzioni.

Non esistono regole generali che permettono di trovare automaticamente la soluzione di un problema.

Si dice **problem solving** l'attività di individuare una o più soluzioni al problema.

È importante:

- conoscere l'argomento oggetto del problema: alcune informazioni necessarie per la soluzione possono essere implicite, cioè legate alla natura stessa del problema;
- usare algoritmi noti per problemi analoghi (sfruttare gli elementi comuni per adattare metodi risolutivi già individuati e sperimentati); dato un problema analogo di cui sia nota la soluzione, individuando le caratteristiche simili dei due problemi e le reciproche differenze è possibile tramite una deduzione per analogia ricondurre la soluzione del nuovo problema a quella già nota;
- dividere il problema in sottoproblemi che insieme poi risolvono il problema per ridurre la complessità di ogni singolo sottoproblema.

Metodologia top down

Le strategie che partendo dalle possibili soluzioni cercano di ottenere i dati di partenza sono chiamate **top down**, **goal driven** o **backward chaining**. L'analisi top down (dall'alto in basso) è un metodo di programmazione che permette di descrivere l'algoritmo prima in modo generale e poi sempre più in dettaglio, fino ad ottenere la versione definitiva. In questo modo la stesura dell'algoritmo risulta molto più semplice; un problema complesso può essere scomposto in una serie di sottoproblemi più semplici, affrontando ognuno dei quali separatamente, si raggiunge la soluzione finale.

- combinare più sottoproblemi di cui si conosce la soluzione per risolvere un problema più complesso.

Metodologia bottom up

Le strategie che partendo dai dati iniziali, cercano di arrivare alla soluzione sono chiamate **bottom up**, **data driven** o **forward chaining**.

Nella programmazione imperativa di solito è più usato il metodo top down.
Nella programmazione a oggetti è più usato il metodo bottom up.

Algoritmi euristici

In alcuni casi si può trovare la soluzione di un problema con un procedimento euristico. Un **algoritmo euristico** è un procedimento che cerca la soluzione per **tentativi ed errori**; non garantisce il raggiungimento di una soluzione, ma offre solo un metodo per la ricerca della soluzione stessa.
Per poter definire un algoritmo euristico, si devono realizzare delle tecniche di **backtracking**, cioè la possibilità di ritornare sui propri passi per scegliere una soluzione alternativa nel momento in cui ci si accorge che il cammino percorso non è valido.

> Si può risolvere con un algoritmo euristico per esempio il problema di spostare un cavallo da una casella ad un'altra della scacchiera; si possono provare varie mosse, tornando indietro se ci si accorge che le mosse compiute non portano ad alcuna soluzione.

COMPUTABILITÀ

Un problema si dice **computabile** se la sua soluzione può essere descritta mediante un algoritmo (cioè un procedimento che termini dopo un numero finito di passi, descritto in modo deterministico, effettivamente eseguibile dall'esecutore e generale per la classe di problemi considerata).

Esistono anche dei problemi che non sono risolvibili con un procedimento algoritmico; questi problemi si dicono **non computabili**.

È difficile immaginarsi un problema non computabile eppure i problemi non computabili sono addirittura più numerosi di quelli computabili.

> Un esempio di problema non computabile è quello noto come problema della fermata: non esiste un algoritmo in grado di individuare la presenza di un loop infinito in un programma prima dell'esecuzione.
> Cioè non è possibile realizzare un programma che prenda in input un programma P e un dato D e stabilisca se il programma P termina l'esecuzione elaborando il dato D oppure causa un loop infinito.

Per poter affermare che un problema è non computabile, e quindi insolubile, bisogna "dimostrare" che non esiste una soluzione algoritmica.
Ci sono dei problemi per cui non si riesce a trovare una soluzione algoritmica, ma non si riesce nemmeno a dimostrare che non esiste una soluzione.

I **problemi** si possono quindi dividere in:

- problemi **computabili** se esiste una soluzione algoritmica;
- problemi **non computabili** se è dimostrato che non esiste una soluzione algoritmica;
- problemi che per il momento vengono **considerati non computabili** ma per cui potrebbe essere trovata prima o poi una soluzione.

2.11 L'INTERFACCIA PROGRAMMA UTENTE

Con il termine **interfaccia programma/utente** si intende il modo in cui vengono gestite le comunicazioni tra programma e utente.

L'interfaccia programma utente può essere realizzata in vari modi:

- interfaccia testuale (a **linea di comando**);
- selezione di comandi da **menu**;
- presentazione di **moduli** (o **form**) da compilare (inserimento di dati, scelta di opzioni ecc.);
- **interfaccia grafica** (**GUI** - Graphics User Interface) con strumenti di puntamento per indicare oggetti e azioni.

In ogni caso l'utente ha bisogno di sapere sempre che cosa deve fare e deve essere il programma a spiegarglielo; quando il programma richiede un input, deve specificare quale dato si aspetta di ottenere; quando l'utente inserisce il dato richiesto, il programma deve controllarne l'esattezza, nei limiti del possibile; quando il programma fornisce dei risultati deve specificare di che cosa si tratta esattamente.

Nei programmi a **interfaccia testuale** l'interfaccia programma/utente è costituita dalle **istruzioni di input e di output**; ogni istruzione di input deve essere preceduta da una istruzione di output (**prompt**) che spieghi quale input è atteso; l'output deve essere esplicativo: oltre al risultato effettivo, può comprendere altri dati per maggior chiarezza.

Nota

È bene cercare di produrre sempre un output più completo e comprensibile possibile.

Un programma per la somma di due numeri oltre al risultato può visualizzare anche l'operazione completa: operandi con il simbolo di addizione, il simbolo di uguale e il risultato.

Anche in programmi molto semplici, poche modifiche alle **istruzioni di output** producono effetti molto diversi. Alcune modifiche secondarie e facilmente apportabili, che possono influenzare l'interfaccia, sono per esempio:

- la posizione dei messaggi all'interno del programma;
- inserire o meno un ritorno a capo in un certo punto;
- mandare l'output alla stampante, o a un file, invece che al video.

Bisogna prestare molta attenzione anche a questi particolari perché il programma risulti semplice e gradevole da usare.

In programmi più significativi la scelta di una determinata **interfaccia** può influenzare pesantemente la scelta dei **dati** da utilizzare o l'intera struttura dell'**algoritmo**.

È fondamentale decidere il tipo di **interfaccia** al momento dell'**analisi** del problema e cominciare già a stendere le **istruzioni per l'uso** sulla base dell'interfaccia progettata.

Inserimento di due array
Si può inserire alternativamente un elemento di ciascun array, oppure prima tutti gli elementi di un array e poi quelli dell'altro.

Stampa di matrici
Se non vengono previsti gli opportuni ritorni a capo alla fine di ogni riga, i valori vengono stampati tutti uno di seguito all'altro.
Per facilitare l'inserimento dei dati, anche le operazioni di lettura possono essere combinate con operazioni di posizionamento a capo e spaziature, in modo che anche i dati inseriti assumano un aspetto tabellare.

2.12 IMPLEMENTAZIONE E DOCUMENTAZIONE

Dopo aver determinato l'algoritmo, si passa alla realizzazione del programma eseguibile al computer (**implementazione**).

Con il termine **implementazione** si intende la realizzazione del programma al computer, che comprende le fasi che vanno dalla codifica dell'algoritmo in un linguaggio di programmazione al completamento del programma, perfettamente funzionante (in pratica l'implementazione è la concretizzazione nel programma di una descrizione astratta, l'algoritmo).

È importante che il programma sia corredato della **documentazione** necessaria al suo utilizzo (documentazione esterna) e di quella necessaria a comprendere come il programma è stato realizzato per poter effettuare successive manutenzioni (documentazione interna).

Durante le varie fasi bisogna provvedere al controllo della correttezza (**verifica**) per assicurarsi che il programma funzioni correttamente.

Già durante la stesura dell'algoritmo si devono stabilire i casi di prova per la verifica della correttezza e stendere la documentazione con le istruzioni per l'utilizzo del programma.

L'implementazione

La codifica

La **codifica** dell'algoritmo consiste nella scrittura del programma in un linguaggio di programmazione.

Se il programma è suddiviso in moduli, per la codifica si può procedere con il metodo **top down**, partendo dal programma principale e procedendo via via con i moduli di livello inferiore o, al contrario, con il metodo **bottom up**, partendo dai moduli a livello più basso e risalendo fino al programma principale.

Per scrivere il codice del programma si usa un **editor**; il codice del programma viene memorizzato in un file chiamato programma sorgente.

Il **programma sorgente** è il programma scritto in linguaggio di programmazione.

Ogni linguaggio di programmazione impone proprie regole per la codifica del programma.

Alcuni linguaggi sono **case sensitive**, cioè distinguono tra caratteri maiuscoli e minuscoli mentre altri no.

Ogni linguaggio usa delle parole riservate e degli identificatori.

Le **parole riservate** sono parole predefinite del linguaggio che hanno un significato particolare e possono essere usate solo per lo scopo previsto.

Gli **identificatori** sono tutte le altre parole utilizzate nel programma, tutti i nomi a scelta del programmatore, per esempio per i nomi delle variabili.
Gli identificatori devono sempre essere diversi dalle parole riservate.
Ogni identificatore inoltre deve essere univoco, cioè deve essere utilizzato per un solo scopo.

Suggerimenti per la codifica

Oltre alle regole di codifica imposte dal linguaggio adottato, è bene seguire anche alcune altre semplici regole, che, pur non essendo obbligatorie, permettono di rendere il programma più comprensibile; per esempio:

- scegliere nomi significativi per tutti gli identificatori (variabili, procedure ecc.), in modo che il nome ne ricordi e spieghi l'uso;
- scrivere le istruzioni con il metodo dell'indentazione in modo da evidenziare le strutture di controllo;
- fare ampio uso di commenti all'interno del programma per descrivere il significato delle istruzioni.

Per l'uso di maiuscole e minuscole:

- per quanto riguarda le istruzioni, se il linguaggio è case sensitive bisogna rispettare le regole imposte; se non è case sensitive è conveniente scrivere tutto in minuscolo;

Nota — Alcuni linguaggi adottano convenzioni proprie, pur non essendo case sensitive.

- per quanto riguarda gli identificatori, se il linguaggio è case sensitive un identificatore scritto in minuscolo viene considerato diverso dallo stesso scritto in maiuscolo; anche se il linguaggio non è case sensitive, conviene comunque adottare una regola e seguirla sempre per non confondersi; è tendenza comune scrivere gli identificatori in minuscolo; se l'identificatore è una parola composta, per maggior comprensibilità conviene far iniziare le parole successive con la maiuscola (per esempio *numeroRighe*).

Fasi successive alla codifica

Dopo la codifica si procede in modo diverso se si usa un linguaggio interpretato o un linguaggio compilato.

Se si usa un linguaggio **interpretato** si può provare subito l'esecuzione del programma nell'ambiente di run-time.

Se si usa un linguaggio **compilato** bisogna

- compilare il programma,
- linkare il programma,
- eseguire il programma.

Compilazione

Il **compilatore** traduce il programma sorgente in programma oggetto (in linguaggio macchina); prende in input il programma sorgente e produce come output il programma oggetto.

La produzione del programma oggetto può avvenire soltanto se nel programma sorgente non sono presenti errori formali; se ci sono errori, infatti, la compilazione viene impedita; gli errori vengono segnalati dal compilatore con opportuni messaggi.
Bisogna correggere gli errori di compilazione prima di poter ottenere il programma oggetto.

L'**output del compilatore** può includere:

- informazioni sul compilatore e sulle opzioni usate per la compilazione;
- il listato delle istruzioni del programma sorgente;
- informazioni sugli identificatori usati, con il riferimento alle righe di programma in cui sono stati utilizzati (cross reference);
- messaggi di diagnostica del compilatore (tipi di errore, gravità, linee del programma a cui si sono verificati);
- il programma oggetto.

I **messaggi di diagnostica** del compilatore si suddividono di solito in:

- **errori**: errori veri e propri che impediscono la produzione del programma oggetto;
- **warning** o avvertimenti: segnalazioni di situazioni anomale che potrebbero essere causa di errore, o di istruzioni che non rientrano nel livello di standard richiesto.

Se non ci sono errori veri e propri, ma solo warning, il programma oggetto viene prodotto; è opportuno però controllare ogni warning, per assicurarsi che non ci possano essere cause nascoste di errori di esecuzione.

Nota

Linkaggio

Prima di poter essere eseguito, il programma compilato deve essere linkato.

Il programma oggetto preparato dal compilatore non è ancora un programma effettivamente eseguibile; il programma eseguibile viene preparato dal linker.

Il **linker** prende come input uno o più programmi oggetto e produce come output un programma eseguibile.

Possono esserci degli **errori** nelle relazioni tra i moduli oggetto che impediscono al linker di creare il programma eseguibile.

Esecuzione del programma

Eseguire il programma significa utilizzarlo per risolvere un caso particolare del problema per cui è stato preparato.

L'utente deve fornire i dati iniziali (**input** del programma) e, dopo l'elaborazione, ottiene i risultati che aspettava (**output** del programma).

Per esempio se è stato realizzato un programma per la somma di due numeri, l'utente inserisce i due numeri che desidera sommare e riceve in output il risultato dell'operazione.

L'output dell'esecuzione può comprendere, oltre al vero e proprio output del programma, anche messaggi di errore per **errori a tempo di esecuzione**.

Il modo in cui l'utente e il programma interagiscono tra loro (messaggi di richiesta, modo in cui i dati devono essere inseriti, modo di emissione dei risultati) costituisce l'**interfaccia programma/utente**.

La documentazione

La **documentazione** deve essere prodotta man mano che si procede nel lavoro.

Ogni revisione del lavoro richiede anche la contemporanea revisione della documentazione, in modo che la documentazione sia sempre perfettamente aderente alla situazione del programma.

La documentazione si suddivide in:

- documentazione **esterna**, o manuale utente per l'uso del programma, rivolta agli utenti;
- documentazione **interna**, per la manutenzione del programma, rivolta ai programmatori.

Il manuale dell'utente

Il **manuale utente** o **manuale d'uso** è la documentazione fondamentale per chi usa il programma; deve contenere almeno:

- la **descrizione generale** del prodotto,
- le procedure di **installazione** e le istruzioni per avviare l'esecuzione del programma;
- il **manuale di riferimento** con la spiegazione di tutte le situazioni possibili in modo sistematico, che comprende la documentazione dell'interfaccia programma/utente con la descrizione della sequenza delle istruzioni da seguire durante le varie fasi di esecuzione, la descrizione delle videate nel caso dell'interfaccia grafica e la descrizione dei prospetti di stampa (report) che si ottengono;
- la spiegazione dei **messaggi di errore** presentati dal programma.

L'utente in generale non è un programmatore, non sa come sia fatto veramente il programma, e spesso non ha neppure nozioni di programmazione o di altri aspetti dell'informatica; le istruzioni devono essere dettagliate e non devono presupporre che l'utente abbia conoscenze specifiche di programmazione.

Nota

La documentazione interna

La documentazione interna descrive come il programma è stato effettivamente realizzato; è importante per le fasi di **revisione** e **manutenzione**.

La documentazione interna è importantissima soprattutto in ambiente aziendale, dove la fase di manutenzione del programma (revisione, modifica, adattamento a nuove richieste ecc.) è ancora più lunga della fase di realizzazione e dove spesso non è la stessa persona, o gruppo di persone,

che ha creato il programma, ad occuparsene poi per tutto il resto della vita del programma.

Per la realizzazione della documentazione interna di solito si usano opportune **metodologie**, **strumenti grafici** (diagrammi, tavole ecc.) e moduli appositamente predisposti.

La documentazione interna deve descrivere:

- le **specifiche** del problema: descrizione del problema da risolvere, puntualizzando le funzioni da svolgere, gli obiettivi da raggiungere, le risorse disponibili, i limiti della realizzazione, i compromessi che si attuano per l'implementazione ecc.
- la **progettazione**: architettura generale, descrizione dei dati, descrizione dei singoli componenti) ecc.
- l'**implementazione**, cioè quello che riguarda la **codifica** e le **strategie di test**: informazioni sugli strumenti usati, il listato della codifica del programma, i test da effettuare e l'output delle prove di esecuzione che rappresentano anche degli esempi di uso del programma.

Documentazione di progettazione per un semplice programma

La documentazione di progettazione per un semplice programma deve comprendere almeno:

- la descrizione dei **dati**, precisando per ognuno di essi il nome e il tipo della variabile e il significato che la variabile assume e suddividendoli in dati di input, dati di output e altri dati utilizzati internamente;
- la descrizione dell'**algoritmo** (per esempio con la pseudocodifica o il flow-chart);
- i **casi di prova**.

Se il programma si compone di più moduli, per evidenziare i rapporti e le dipendenze tra le varie procedure si può utilizzare un metodo grafico: una struttura ad albero che, partendo dal programma principale, indica come figli le procedure richiamate, procedendo via via ai livelli successivi.

Le informazioni fondamentali da indicare per ogni modulo sono:

- la **funzione** del modulo, cioè il compito che svolge;
- i **parametri**, cioè i valori che vengono scambiati con il programma chiamante.

Nota

Con queste informazioni il modulo può essere riutilizzato in un altro programma.

- la **documentazione interna** del modulo, come per il programma:
  - dati: evidenziando i dati di input e di output (da non confondere con i parametri) e i dati interni;
  - algoritmo;
  - casi di prova.

2.13 LA VERIFICA DEL PROGRAMMA

Gli errori

Durante le fasi di creazione del programma si possono verificare diversi tipi di errori:

- errori formali;
- errori di esecuzione;
- errori di logica.

Errori formali

Gli errori **formali** sono gli errori che impediscono al compilatore di produrre il codice oggetto e all'interprete di proseguire nell'esecuzione del programma.

Gli errori formali sono segnalati dal programma traduttore (l'interprete o il compilatore) e sono gli errori più semplici da correggere.

Gli errori di questo tipo comprendono gli errori nel **lessico** e gli errori di **sintassi** come:

- banali errori di scrittura (come scrivere una lettera al posto di un'altra);
- errori nella punteggiatura richiesta dal linguaggio;
- errori nella sintassi delle istruzioni (come usare un'istruzione in modo non del tutto corretto, per esempio tralasciando una parola necessaria);
- uso non consentito di parole riservate (come utilizzare una parola riservata come nome per una variabile);
- uso di variabili non dichiarate.

Nota

I programmi traduttori possono riconoscere anche alcuni errori **semantici**, come l'assegnazione di un tipo di dati errato a una variabile.

Errori di esecuzione

Gli errori **di esecuzione** sono errori che si verificano a tempo di run-time (durante l'esecuzione) a causa di una operazione non consentita (di solito per l'uso di valori non permessi in un determinato contesto).

Gli errori di esecuzione sono detti anche errori **semantici** o di significato.

Un esempio di errore di esecuzione è la richiesta di eseguire una divisione per zero.

Alcuni errori di esecuzione possono essere dovuti all'inserimento di dati di input errati da parte dell'utente; per prevenire questi errori sono importanti le istruzioni per l'esecuzione del programma, inoltre il programma dovrebbe sempre, per quanto possibile, controllare l'esattezza dei dati inseriti, segnalando gli errori e permettendo di correggere i dati errati.

Errori di logica

Gli errori **di logica** sono tutti quegli errori che fanno ottenere risultati non corretti; il programma funziona, ma non nel modo che si desidera, non fa il lavoro per cui è stato progettato.

Gli errori di logica possono essere dovuti a una discordanza tra la codifica e l'algoritmo (in questo caso il programma non si comporta secondo le aspettative perché non rispetta fedelmente l'algoritmo progettato) o a errori più o meno gravi nella progettazione che possono richiedere anche la revisione dell'intero lavoro.

Il debug

Viene chiamata **debug** la fase di prova del programma, durante la quale vengono individuati e corretti gli errori che impediscono il corretto funzionamento del programma.

Per avere la certezza assoluta che il programma sia perfettamente esatto bisognerebbe verificare i risultati ottenuti per qualsiasi combinazione di dati di input; questo ovviamente è impossibile anche per programmi non troppo complicati poiché le combinazioni di valori ammessi in genere sono moltissime, se non infinite.

Non potendo escludere del tutto la possibilità di errore, si deve comunque cercare di raggiungere un **ragionevole grado di correttezza**.

Si procede provando il programma su una serie di **dati campione**, caratteristici delle diverse situazioni che si possono verificare durante l'esecuzione.
I **risultati** prodotti in output dal programma devono essere **confrontati** con quelli **attesi**; se ci sono differenze bisogna effettuare altre prove per individuare esattamente l'errore che le provoca.
Dopo aver apportato le correzioni previste, bisogna **ripetere** tutte le prove effettuate, con gli stessi dati, per verificare se l'errore è stato effettivamente corretto.

Nota

Tutte le correzioni devono sempre essere apportate non solo al programma ma anche alla documentazione.
Se il programma è in un linguaggio compilato perché le modifiche diventino effettive il programma deve essere nuovamente compilato e linkato.

È conveniente anche, quando è possibile, far provare il programma a uno degli utenti finali o comunque a una persona che non ha partecipato alla realizzazione del programma; in questo modo chi prova il programma non è influenzato dal modo in cui il programma è stato sviluppato e non rischia di provare soltanto ciò che si aspetta di ottenere.

La scelta dei dati campione

I dati campione per la prova del programma possono essere scelti:

- in base alla **struttura** del programma, scegliendo cioè dati che permettano di esplorare i diversi rami delle strutture condizionali e il corpo delle strutture cicliche;
- secondo le **specifiche** del problema, in base alle funzioni del programma, scegliendo dapprima **valori semplici** e poi **valori realistici**, e provando inoltre i **valori estremi** (che rappresentano situazioni limite, che non rientrano nel generico flusso di esecuzione) e **valori illegali** (che possono cioè causare errori durante l'esecuzione) per verificare se il programma è in grado di gestire situazioni anomale.

Nota

Per verificare un algoritmo ripetitivo si può utilizzare il principio di induzione: si verifica il caso iniziale, si suppone poi che un generico caso sia verificato e in funzione di questo si verifica il caso successivo; la correttezza di queste due situazioni conferma l'esattezza di qualsiasi caso generale.

La scelta dei casi di prova deve essere effettuata **durante la progettazione** e non soltanto dopo l'implementazione.

La traccia del programma (o trace)

Per controllare il programma si possono seguire passo passo le istruzioni, partendo da alcuni valori di input e annotando ad ogni passo il valore che assumono le variabili. Questo procedimento viene chiamato **traccia** o **trace** e può essere eseguito sia sulla descrizione dell'algoritmo che sul programma.
La traccia può essere eseguita anche durante l'esecuzione del programma, eventualmente inserendo delle istruzioni che visualizzino il contenuto delle variabili in punti intermedi dell'elaborazione.

Per esempio un'istruzione di stampa posta all'interno di un ciclo permette di stabilire quante volte il ciclo viene percorso; una stampa all'interno di una struttura condizionale permette di stabilire se la condizione esaminata è vera o falsa e così via.

Gli ambienti di debug

Esistono ambienti che permettono di eseguire in modo **interattivo** il debug del programma, senza dover aggiungere nuove istruzioni al programma; questi ambienti permettono di testare il programma e di individuare gli errori in modo molto più rapido.
È possibile per esempio seguire automaticamente la traccia del programma o di una parte di esso, interrompere l'elaborazione in un punto desiderato, visualizzare il contenuto di una variabile, o addirittura modificarne il contenuto prima di riprendere l'esecuzione.

VERIFICA

Questionario

1. Che cos'è un problema e come si può descrivere?
2. Che cos'è il processo risolutivo?
3. Che cos'è un algoritmo?
4. Che cos'è un processo?
5. Che cos'è un programma?
6. Che cosa si intende per dati di input e dati di output?
7. Qual è la differenza tra costanti e variabili?
8. Quali sono i principali tipi di dati?
9. Che cosa sono le istruzioni di controllo?
10. Che cos'è la pseudocodifica?
11. Che cosa sono i flow-chart?
12. Che cosa si intende per programmazione strutturata?
13. Che cosa dice il teorema di Bohm-Jacopini?
14. Che cos'è il problem solving?
15. Che cos'è l'interfaccia programma/utente?
16. Quali sono le fasi di implementazione di un programma?
17. Qual è la differenza tra documentazione interna e documentazione esterna?
18. Quali sono i tipi di errori che si possono verificare?
19. Che cos'è il debug?
20. Come si possono scegliere i dati per provare il programma?
21. In che cosa consiste la traccia di un programma e come può essere effettuata?

Test

1 ..... *Indicare se le seguenti affermazioni sono vere o false.*

V	F	
☐	☐	1) L'esecutore del processo risolutivo di un problema è sempre il computer.
☐	☐	2) Il computer può risolvere qualsiasi tipo di problemi.
☐	☐	3) L'algoritmo dipende dalle azioni che l'esecutore è in grado di eseguire.
☐	☐	4) Un algoritmo può non essere deterministico.
☐	☐	5) I tipi di dati utilizzabili dipendono dal paradigma di programmazione utilizzato.
☐	☐	6) Un programma si dice portabile se è permesso farne liberamente una copia.

V	F	
☐	☐	7) Un algoritmo si dice strutturato se il programma è realizzato usando un linguaggio strutturato.
☐	☐	8) Ogni algoritmo può essere descritto in modo strutturato.
☐	☐	9) Ogni problema descritto con un flow-chart è sicuramente strutturato.
☐	☐	10) Un algoritmo euristico trova sempre una soluzione.
☐	☐	11) Il backtracking è la possibilità di tornare indietro per scegliere una soluzione alternativa.
☐	☐	12) Un problema che può essere descritto mediante un algoritmo si dice computabile.
☐	☐	13) Tutti i problemi sono computabili.
☐	☐	14) Può non essere possibile dire se un problema è computabile.
☐	☐	15) Nella codifica è sempre fondamentale distinguere tra maiuscole e minuscole.
☐	☐	16) Se il programma è compilato e linkato con successo di sicuro funziona correttamente.

2 ..... *Individuare tutte le attività che possono essere descritte mediante algoritmi.*

- ☐ risolvere l'addizione 3+5;
- ☐ risolvere le addizioni tra numeri interi;
- ☐ cantare una canzone;
- ☐ scrivere un testo sotto dettatura;
- ☐ correggere gli errori ortografici in un testo;
- ☐ correggere gli errori di sintassi in un testo;
- ☐ correggere gli errori di sintassi in un progamma;
- ☐ associare a ogni lettera di un testo il numero corrispondente nell'alfabeto;
- ☐ dipingere un quadro;
- ☐ colorare di rosso tutti i quadrati che compaiono in un disegno;
- ☐ l'attività di problem solving;
- ☐ scattare una fotografia.

Esercizi

1 ..... Individuare i dati necessari, classificandoli in dati di input e output, costanti e variabili per i seguenti problemi:

- sostituire in un testo una lettera con un'altra;
- calcolare il perimetro e l'area di un triangolo;
- copiare un file da una directory ad un'altra;
- associare a ogni lettera di un testo il numero corrispondente nell'alfabeto;
- modificare un colore con un'altro in un'immagine.

2 ..... Si vuole realizzare un programma che permetta di scegliere un colore; descrivere una possibile interfaccia programma/utente:

- a linea di comando;
- a menu;
- mediante moduli;
- grafica (GUI).

3 Il C++ e l'ambiente wxDev-C++

Prerequisiti

- Linguaggi di programmazione
- Programmi di utilità
- Fasi di realizzazione di un programma

Obiettivi

Conoscenze

- Il linguaggio C++
- Compilatori C++
- Struttura di un programma C++
- File di intestazione, librerie e namespace
- Ambiente wxDev-C++

Abilità/capacità

- Capire la struttura di un programma C++ e l'utilizzo di file di intestazione e namespace
- Usare l'ambiente wxDev-C++ per scrivere, compilare ed eseguire programmi in C++
- Usare *gdb* per eseguire il debug di un programma

3.1 I LINGUAGGI C E C++ E I COMPILATORI

Il **C** è un linguaggio di programmazione **imperativa** scritto nei primi anni settanta da **Dennis M. Ritchie** e da **Brian W. Kernighan**, ricercatori presso i Bell Laboratories dell'AT&T.
Per arginare la diffusione di compilatori non compatibili tra loro, l'ANSI (l'ente statunitense per le standardizzazioni) iniziò nel **1983** a lavorare su un'ipotesi di standardizzazione. Questa, pubblicata nel **1989**, prese il nome di **ANSI C**.

Nei primi anni ottanta un altro ricercatore dei Bell Laboratories, **Bjarne Stroustrup** propose il linguaggio "C con classi" ovvero il C con l'aggiunta degli strumenti per la programmazione orientata agli oggetti. Tale linguaggio successivamente prese l'attuale nome: **C++** (si legge C plus plus).

Nel **1999** prima l'ANSI pubblicò le specifiche **C99** per il C e successivamente l'ISO (l'ente internazionale per le standardizzazioni) le specifiche **ISO C++** per il C++.

In questo libro si seguiranno le specifiche ISO C++.

Attualmente esistono moltissimi compilatori che supportano sia il C che il C++. Molto diffusi sono i compilatori professionali della Borland e della Microsoft (Visual C++).

Nota

Ogni compilatore può aggiungere alcune caratteristiche proprietarie all'implementazione del linguaggio. Spesso inoltre vengono offerti file di libreria e strumenti per la creazioni di applicazioni con interfaccia grafica.

wxDev-C++ (o wxDev-Cpp) è un ambiente di programmazione completo, freeware e disponibile sia per *Windows* che per *Linux*, continuazione del più noto progetto Dev-C++.
La versione per *Linux* utilizza il compilatore **GNU C++**, quella per *Windows* utilizza il compilatore **MingW** che consiste nel compilatore GNU C++ con l'aggiunta di strumenti/funzionalità per la piena compatibilità con *Windows*. Entrambe includono il pacchetto **wxWidgets** per la creazione di applicazioni con interfaccia utente grafica.

STRUTTURA DI UN PROGRAMMA C++

C++ 3.1

Il C++ è case sensitive: le parole riservate devono essere scritte in minuscolo.

Gli **identificatori** possono essere lunghi fino a 255 caratteri e possono essere formati solo da lettere dell'alfabeto, cifre e dal carattere di sottolineatura (o underscore _); possono iniziare solo con una lettera o con l'underscore; non possono essere uguali a parole riservate.

Si consiglia di usare per gli identificatori **nomi significativi** che aiutino a ricordare lo scopo per cui vengono utilizzati.
Si suggerisce anche di seguire delle **convenzioni** per la scelta degli identificatori.

Negli esempi verrà seguita la seguente **convenzione**: il nome di un identificatore viene scritto in minuscolo; se è una parola composta le parole successive iniziano con la maiuscola.

Nota

Questa convenzione permette di uniformarsi a convenzioni utilizzate da altri linguaggi di programmazione.

Sono **parole riservate** del C++: *auto*, *bool*, *break*, *case*, *catch*, *char*, *class*, *const*, *const_cast*, *continue*, *default*, *delete*, *do*, *double*, *dynamic_cast*, *else*, *enum*, *explicit*, *extern*, *float*, *for*, *friend*,

goto, *if*, *inline*, *int*, *long*, *main*, *mutable*, *namespace*, *new*, *operator*, *private*, *protected*, *public*, *register*, *reinterpret_cast*, *return*, *short*, *signed*, *sizeof*, *static*, *static_cast*, *struct*, *switch*, *template*, *this*, *throw*, *try*, *typedef*, *typeid*, *typename*, *union*, *unsigned*, *using*, *virtual*, *void*, *volatile*, *while*.

Nota

Sebbene *true*, *false* e *NULL* possano sembrare parole riservate formalmente sono letterali (cioè costanti).

Un programma C++ è costituito da diversi blocchi: direttive di inclusione dei file di intestazione, uso dei namespace, dichiarazione di tipi, costanti e variabili, dichiarazione (e implementazione) di procedure e funzioni.

Nota

Il C++ dispone di molti tipi di dati predefiniti, ma si possono dichiarare anche nuovi tipi.

Il **corpo** del programma principale è costituito dalla funzione ***main()*** contenente una **istruzione composta**, cioè una serie di istruzioni comprese tra le parentesi graffe:

```
int main(void){
   istruzioni
   return 0;
}//main
```

Ogni **istruzione** termina con un **punto e virgola**.

Nota

L'uso errato dei punti e virgola è causa sia di errori di compilazione che di errori logici.

L'istruzione *return* serve a far restituire un valore da una funzione. Il valore 0 indica che il programma termina in modo corretto.

I commenti possono essere scritti con le notazioni:

```
// commento monoriga, cioè fino alla fine della riga
/* commento su
più righe */
```

Nota

Si suggerisce di usare un commento dopo ogni parentesi graffa chiusa, indicante cosa chiude (come mostrato per la funzione *main*), per aumentare la leggibilità del codice.

I FILE DI INTESTAZIONE E LA DIRETTIVA INCLUDE

Il C++ è un linguaggio con pochissime parole chiave; molte istruzioni utili al programmatore (come quelle per la stampa a video o le funzioni matematiche) non sono direttamente disponibili.

Nel programma si possono includere file contenenti definizioni predefinite: **file di intestazione** o **file header.**

I **file di intestazione** contengono le definizioni di tipi, variabili, costanti e funzioni che si possono utilizzare.
Ogni compilatore mette a disposizione vari file di intestazione, alcuni standard, altri propri.
Secondo la sintassi del C, tali file hanno estensione **.h**; nelle specifiche ISO C++ i file header perdono l'estensione.

Per l'inclusione nel programma si usa la direttiva **#include**:

```
#include <nomeFile.h>
#include <nomeFile>
```

Tabella 3.1 *I file header più utilizzati secondo le specifiche ISO C++.*

File header	Significato
<iostream>	gestione dell'input/output
<iomanip>	formattazione dell'input/output
<cstdio>	gestione dell'input/output (secondo le modalità del C) in alternativa a *iostream*
<cmath>	funzioni matematiche
<cstdlib>	funzioni varie di uso comune
<cstring>	funzioni per la gestione di stringhe viste come array di caratteri
<string>	funzioni per la gestione di stringhe viste come oggetti della classe *string*
<fstream>	gestione dell'input/output su file

Il namespace std

Le specifiche ISO C++ introducono il concetto di **namespace** (spazio dei nomi) per definire variabili e funzioni associate ad un certo contesto (o entità) in modo che gli stessi nomi usati in altri ambiti non creino conflitto.

Il namespace che contiene la definizione dei file header e la descrizione delle principali risorse del sistema è **std**.

La clausola **using** rende disponibili le risorse del namespace specificato.
È possibile specificare singole risorse mediante l'operatore di risoluzione di visibilità ::

```
using namespace std;
using std::nomeRisorsa;
```

Righe iniziali di un programma

Se un programma ha bisogno dell'input standard (tastiera) o dell'output standard (video) deve iniziare con le righe:

```
#include <iostream>
using namespace std;
```

L'ambiente del wxDev-C++

C++ 3.2

Scrittura di un programma

Scegliendo ***File***, ***Nuovo, File sorgente*** si apre una finestra di **editor**.
È possibile predisporre il codice da inserire automaticamente quando si crea un nuovo file sorgente nella scheda ***Codice***, sottoscheda *Sorgente predefinito*, della finestra *Opzioni dell'editor* (menu *Strumenti*, *Opzioni dell'editor*).

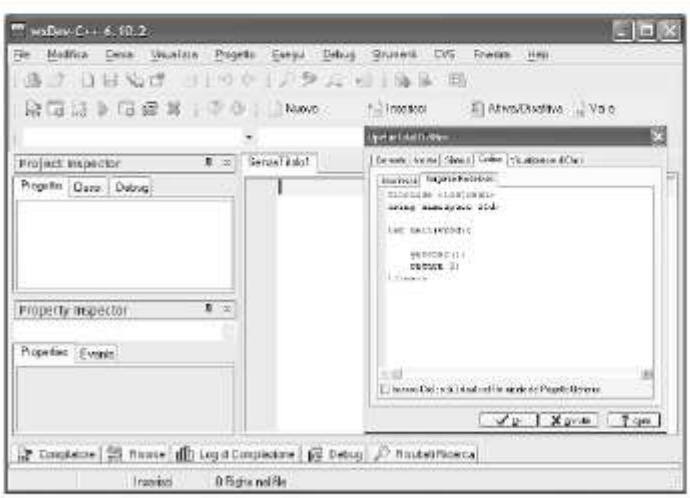


Figura 3.1 *L'editor e la scheda Codice della finestra Opzioni dell'Editor (sottoscheda Sorgente predefinito)*

Nell'editor è possibile scrivere il programma.
Per **salvare** il programma si deve scegliere *File, Salva Come...* indicando nella finestra ***Salva il file*** la directory e il nome del file; per i salvataggi successivi basta scegliere *File, Salva*.

Il comando ***Modifica, Inserisci*** permette di inserire automaticamente un commento di intestazione oppure la data e l'ora.

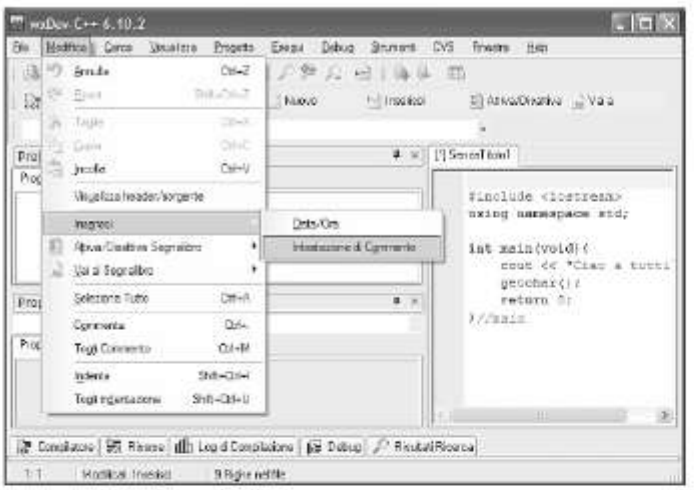


Figura 3.2 *Il comando Modifica, Inserisci*

La **barra di stato** riporta il numero di colonna e riga su cui si è posizionati, l'indicazione se il programma è stato modificato e se si è in modalità inserimento (*Inserisci*) o modifica (*Sovrascrivi*) e il numero di righe del file.

Scegliendo da menu ***Strumenti, Opzioni dell'Ambiente*** si possono modificare alcune impostazioni.

La scheda ***Generale*** della finestra *Opzioni dell'Ambiente* contiene alcune utili caselle di selezione:

Crea file di Backup
: crea una copia di backup quando si modifica un file;

Minimizza durante l'esecuzione
: la finestra dell'editor viene ridotta a icona quando il programma è in esecuzione e ripristinata quando il programma termina;

Osservazione variabile sotto mouse
: durante la fase di debug permette di vedere il contenuto di una variabile quando il mouse vi è sopra.

La scheda ***File & Cartelle*** permette, nella casella *Cartella predefinita dell'utente*, di indicare la directory di default usata per salvare e aprire i file.

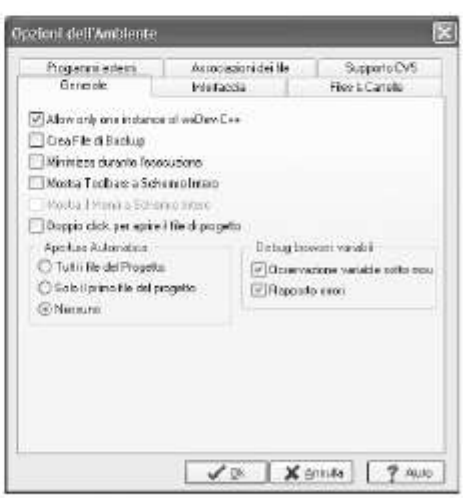


Figura 3.3 *La scheda Generale della finestra Opzioni dell'Ambiente*

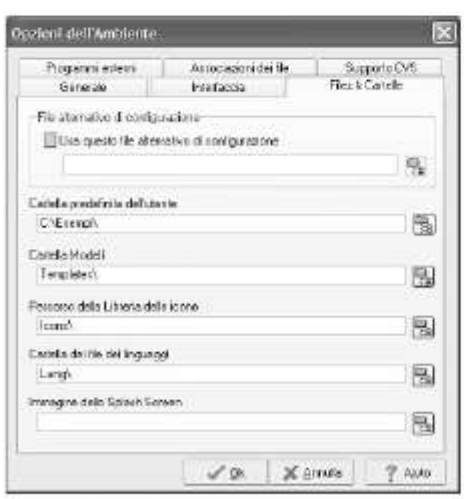


Figura 3.4 *La scheda File & cartelle della finestra Opzioni dell'Ambiente*

La scheda ***Interfaccia*** permette di cambiare l'aspetto grafico dell'ambiente.

Figura 3.5
La scheda Interfaccia della finestra Opzioni dell'Ambiente

Scegliendo da menu ***Strumenti, Opzioni dell'Editor*** si possono modificare alcune impostazioni relative all'aspetto dell'editor.

La scheda ***Generale*** della finestra *Opzioni dell'Editor* contiene alcune utili caselle di selezione:

Auto Indentazione
: mantiene l'indentazione della riga precedente, oppure indenta in corrispondenza di particolari istruzioni come i rami di una selezione o il corpo di un ciclo;

Mostra caratteri di linea nascosti
: mostra i caratteri di controllo dell'editor;

Selezione Linea con doppio click
: seleziona l'intera riga con un doppio click su qualsiasi parola della riga.

Figura 3.6 *La scheda Generale della finestra Opzioni dell'Editor*

Figura 3.7 *La scheda Mostra della finestra Opzioni dell'Editor*

La scheda ***Mostra*** è costituita da due sezioni; nella sezione *Carattere dell'Editor* è possibile scegliere il tipo e la dimensione del carattere usato per scrivere i programmi; nella sezione *Margine* è molto utile la casella di selezione:

Numeri di linea
: mostra i numeri di riga a sinistra nell'editor.

Per **compilare** il programma basta scegliere ***Esegui***, ***Compila***.

Se ci sono **errori di compilazione** vengono visualizzati nel pannello di output del compilatore, in basso. Facendo doppio clic su un errore viene evidenziata la linea che lo contiene.

Per fare in modo che il pannello di output del compilatore sia sempre visibile, basta scegliere da menu il comando *Visualizza*, *Risultati Compilazione*, *Sempre Visibile*.

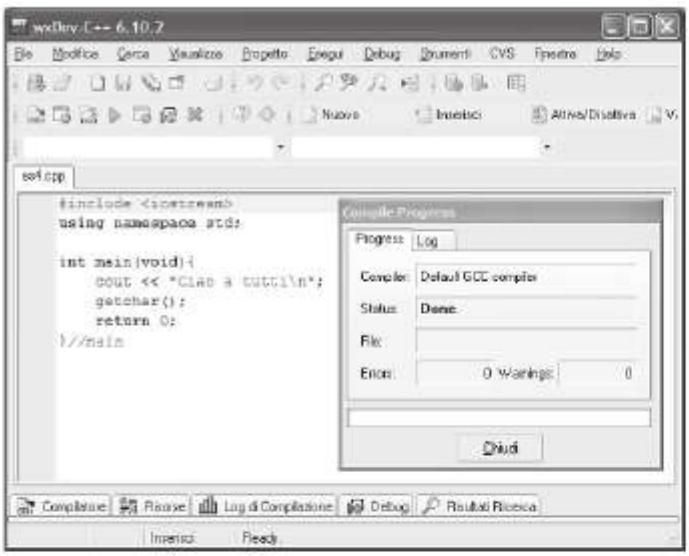


Figura 3.8 *La compilazione di un programma e la finestra Compile Progress*

Se non ci sono errori il programma viene compilato e linkato e viene prodotto un file eseguibile con estensione *.exe*.

Al termine della compilazione compare la finestra di dialogo ***Compile Progress*** (col pulsante *Chiudi*).
Per mandare in **esecuzione** un programma compilato si sceglie da menu ***Esegui, Esegui***.
È possibile compilare e mandare subito in esecuzione un programma richiamando *Esegui, Compila & Esegui*.

Si apre una finestra di **prompt dei comandi** in cui inizia l'esecuzione.

Nota

Al termine dell'esecuzione la finestra viene chiusa; affinché l'output dell'esecuzione rimanga sullo schermo, bisogna far terminare il programma con la chiamata alla funzione *getchar()* che fa sì che il programma attenda la pressione di un tasto da parte dell'utente.

Se si modifica il programma, i comandi per la ricompilazione sono disabilitati e ritornano disponibili solo dopo la chiusura della finestra di prompt.

Se si verifica un **errore di esecuzione** la finestra di prompt dei comandi può chiudersi o rimanere aperta senza segnalare nulla, oppure può comparire una finestra di errore del sistema operativo. Per vedere quale errore si è verificato bisogna eseguire il debug.
Se il programma non si comporta correttamente si può eseguire il debug per controllarne il funzionamento.

Si può fare il **debug** del programma (usando il debug di tipo grafico interno all'ambiente) scegliendo da menu ***Debug***, ***Debug***. La scheda *Debug* viene automaticamente portata in primo piano.

La prima volta che si richiama il comando, il programma potrebbe essere mandato in esecuzione normalmente e potrebbe essere chiesta dopo la chiusura, con la finestra di dialogo *Confirm*, la generazione delle informazioni necessarie al debug.

Per **generare le informazioni di debug** bisogna scegliere da menu *Strumenti, Opzioni di Compilazione* e nella scheda *Generazione di Codice/Ottimizzazioni*, sezione *Linker*, porre a *Yes* la voce *Genera le informazioni per il debug*.

Figura 3.9 *La finestra di dialogo Confirm*

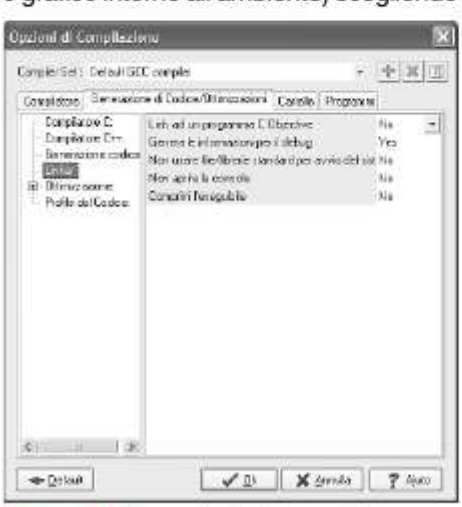


Figura 3.10 *La scheda Generazione di Codice/Ottimizzazioni della finestra Opzioni di compilazione*

Nei capitoli successivi sono presentati alcuni esempi di utilizzo del debug interno. **Nota**

Richiamare il compilatore da linea di comando

Si può richiamare il compilatore per compilare e linkare il programma da **linea di comando** con ***g++***:

```
<directoryDevCpp>\bin\g++ -o nomeEseguibile nomeSorgente.cpp
```

Alcune opzioni utili sono:

-o	specifica il percorso e\o il nome dell'eseguibile;
-g	genera le informazioni per il debug;
-v	aumenta i messaggi sulle operazioni eseguite;
-c	compila senza linkare; produce il file oggetto.

L'invocazione di *g++* senza l'opzione *-o* genera sempre l'eseguibile *a.exe*. **Nota**

La variabile d'ambiente *Path*

Per evitare di scrivere prima di ogni invocazione di g++ la directory di installazione di wxDev-C++, si può inserire il suo percorso nella variabile d'ambiente *Path*.

In **Windows Vista**, **XP** e **2000** si imposta la variabile dal *Pannello di Controllo*, icona *Sistema*, scheda *Avanzate*, pulsante *Variabili d'ambiente*, sezione *Variabili dell'utente*, o *Variabili di sistema* per impostare la variabile per tutti gli utenti; basta selezionare la variabile *Path*, premere il pulsante *Modifica* e aggiungere in coda il percorso; i path sono separati da **;**.

In **Linux** si può modificare il path con il comando

```
PATH=$PATH:nuovaDirectory
```

I path sono separati da **:**.

Per **impostare automaticamente** la variabile all'avvio del sistema, si può impostare la variabile nel file *.profile* della home directory dell'utente, o nel file */etc/profile* per impostare la variabile per tutti gli utenti.

Figura 3.11
La finestra Variabili d'ambiente di Windows

Debug con gdb

È possibile eseguire il debug da riga di comando grazie all'eseguibile ***gdb*** (GNU Debugger) contenuto nella directory *bin* della directory di installazione del *wxDev-Cpp*.

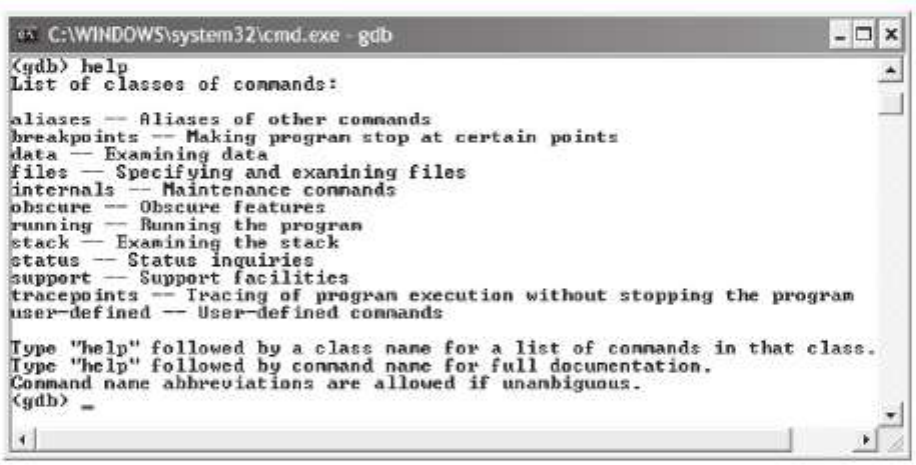
Richiamando ***gdb*** si ottiene il prompt *(gdb)*.

Sono disponibili molti **comandi** per esaminare il funzionamento del programma; i principali sono:

q (*quit*)	esce dal programma;
h (*help*)	help dei comandi;
file *nomeExe*	carica per il debug il file *nomeExe* generato da un'invocazione del compilatore con l'opzione *-g*;

l (*list*) visualizza alcune righe del programma;
l *numerolinea* visualizza alcune righe del programma prima e dopo la linea specificata;
break *numerolinea*
inserisce un breakpoint subito prima della linea con il numero specificato (un breakpoint interrompe l'esecuzione del programma);
break *nomeProcedura* o *nomeFunzione*
inserisce un breakpoint all'inizio della procedura o funzione (i nomi delle procedure e delle funzioni vanno scritti sempre in maiuscolo);
watch *nomeVariabile*
imposta un watchpoint sulla variabile; durante l'esecuzione, ogni volta che la variabile cambia valore, vengono visualizzati sia il vecchio che il nuovo valore;
info break mostra i breakpoint impostati;
r (*run*) esegue il programma fino al primo breakpoint;
s (*step*) esegue l'istruzione successiva, anche se appartiene a una procedura o funzione;
n (*next*) esegue l'istruzione successiva; se è una procedura o funzione la esegue fino alla fine;
c (*continue*) riprende l'esecuzione del programma;
p (*print*) *espressione*
visualizza il valore dell'espressione specificata; come espressione si può usare tra l'altro il nome di una variabile.

Figura 3.12
L'help dei comandi in gdb



In genere si imposta qualche breakpoint o watchpoint e poi si avvia l'esecuzione del programma con *run*.
Quando il programma si arresta si possono usare i comandi di debug, per esempio per far avanzare il programma un'istruzione alla volta o per visualizzare il contenuto delle variabili.

Nota Nei capitoli successivi sono presentati alcuni esempi di utilizzo di *gdb*.

PROPOSTE DI LAVORO

1 ..... Installare e configurare wxDev-C++.
2 ..... Scrivere e compilare uno dei programmi del capitolo 4; poi eseguire il programma.
3 ..... Avviare il debug sul programma compilato ed utilizzare l'help per esaminare i comandi di debug.

Modulo 2

La programmazione imperativa

4 Dati e istruzioni di I/O - I primi programmi

Prerequisiti

- Algoritmi, dati e istruzioni
- Pseucodifica e flow-chart
- Fasi di realizzazione di un programma
- Uso di un ambiente di programmazione

Obiettivi

Conoscenze

- Istruzioni di input/output
- Variabili e costanti
- Tipi di dati
- Assegnazione
- Espressioni e operatori
- Funzioni predefinite

Abilità/capacità

- Descrivere semplici algoritmi
- Realizzare la tabella di traccia di un algoritmo
- Scrivere semplici programmi in C++
- Eseguire il debug di un programma

4.1 ISTRUZIONI DI I/O

Istruzioni di output

Le istruzioni di **output** permettono di comunicare all'utente un dato, costante o variabile.

Il contenuto di una variabile non viene modificato da una istruzione di output.

Nota

Esempio 4.1

Comunicazione all'utente di un dato costante tramite una istruzione di output.

```
inizio
    scrivi "Ciao a tutti"
fine
```

Istruzioni di input

Le istruzioni di **input** permettono di inserire un valore in una variabile.

Quando il programma incontra una istruzione del tipo

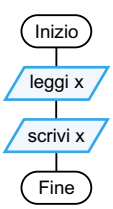
```
leggi x
```

si ferma e aspetta che l'utente inserisca un dato; quando viene premuto il tasto di invio, il dato viene letto e inserito nell'area di memoria riservata alla variabile per poter essere utilizzato successivamente.

Esempio 4.2

Input e output di una variabile.

```
inizio
    leggi x
    scrivi x
fine
```



Per poter scrivere un programma a partire da questo algoritmo bisogna stabilire il tipo della variabile *x*.

Nota

Input e output

Per non fare confusione con i termini **input** e **output** bisogna tenere presente che si parla di lettura (input) e scrittura (output) in riferimento al programma e non all'utente. L'**output** del programma è il **risultato** che si vuole ottenere dal programma stesso. L'**input** è costituito dai **dati noti**, forniti dall'utente al momento dell'esecuzione del programma.

Uso delle istruzioni di input e output

Quando incontra una istruzione di input, il programma si interrompe in attesa dell'inserimento del dato, però non segnala che cosa sta aspettando per proseguire.
Perché l'utente possa sapere di quale tipo di dato è in attesa il programma, è opportuno far precedere ogni istruzione di **input** da una istruzione di output con l'emissione di un **messaggio di spiegazione**.

In alcuni linguaggi l'istruzione di input permette di specificare anche il messaggio. **Nota**

È bene completare anche i dati di **output** con **messaggi esplicativi**.

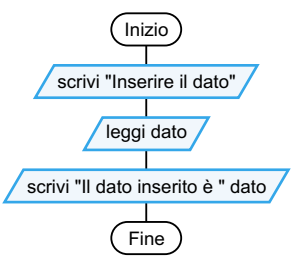
Una stessa istruzione di output può comprendere sia costanti che variabili.

Attenzione a non confondere la stampa del nome delle variabili con la stampa del loro contenuto. La sintassi dipende dal linguaggio di programmazione ma in genere se il nome della variabile compare tra apici o virgolette viene utilizzato il nome (costante) della variabile e non il suo contenuto. **Nota**

Esempio 4.3

Richiesta, input e stampa di un dato variabile.

```
inizio
    scrivi "Inserire il dato"
    leggi dato
    scrivi "Il dato inserito è " dato
fine
```



C++ 4.1 Le istruzioni di scrittura

Il C++ non ha istruzioni specifiche per l'output, ma nella libreria standard *<iostream>* è presente la definizione del dispositivo di output ***cout***; il dispositivo di output standard è il video.
L'operatore << trasferisce il valore al dispositivo:

```
cout << valore;
```

Il valore può essere una costante o una variabile; le **costanti** vanno racchiuse tra virgolette.

Per andare a capo dopo la stampa si può usare:

- la sequenza di **escape** per il ritorno a capo ***\n***;
- il manipolatore ***endl***.

Sono equivalenti e pongono il cursore su una nuova riga. **Nota**

In una stessa istruzione possono essere specificati più valori, costanti o variabili.

```
cout << valore1 << valore2 << ... <<  valoreN;
```

I valori vengono stampati l'uno immediatamente dopo l'altro, senza spaziature.

Per lasciare uno spazio tra due valori basta richiedere la stampa di un carattere spazio:

```
cout << valore1 << " " << valore2;
```

Esempio 4.4

Codifica dell'algoritmo dell'esempio 4.1.

```
#include <iostream>

using std::cout;

int main(void){
    cout << "Ciao a tutti\n";
    getchar();
    return 0;
}//main
```

Figura 4.1 *Esecuzione dell'esempio*

Nota

La funzione *getchar()* attende che l'utente prema un tasto qualsiasi prima di terminare il programma, in modo che l'output rimanga visibile; è contenuta nella libreria standard *<stdio.h>*, ridefinita dalla libreria standard *<cstdio>*, richiamata dalla libreria *<iostream>*.

Le istruzioni di scrittura in C

Per la stampa di valori è possibile utilizzare anche le funzioni ***printf()***, ***putchar()*** e ***puts()*** derivanti dal linguaggio C e presenti nella libreria standard *<stdio.h>* (e quindi in *<cstdio>*). La funzione *printf()* ha le stesse potenzialità di *cout*, *putchar()* permette la stampa di un carattere e *puts()* la stampa di una stringa.

Nota

La sintassi della funzione *printf()* è trattata nel paragrafo C++ 4.10.

Esempio 4.5

Codifica dell'algoritmo dell'esempio 4.1 usando la funzione *printf* e le specifiche del linguaggio ANSI C.

```
#include <stdio.h>

int main(void){
    printf("Ciao a tutti\n");
    getchar();
    return 0;
}//main
```

Figura 4.2 *Esecuzione dell'esempio*

I dispositivi cerr e clog

I dispositivi ***cerr*** e ***clog*** si riferiscono entrambi allo standard error e permettono tramite l'operatore << l'output di messaggi di errore; lo standard error è solitamente il video.
Il dispositivo *cerr* possiede (come *cout*) un buffer dove memorizzare temporaneamente i messaggi inviati allo standard error per offrire al sistema operativo una certa flessibilità di gestione; *clog*, invece, non possiede un buffer, quindi i messaggi sono inviati immediatamente allo standard error.
L'esistenza di *cerr* e *clog* permette in C++ di ridefinire quali sono i dispositivi standard di output e di errore; è possibile, per esempio, impostare la stampante come dispositivo per i normali messaggi e il video per i messaggi di errore.

```
cerr << "Hai sbagliato comando, riprova\n";
```
stampa a video il messaggio di errore specificato.

C++ 4.2

Le istruzioni di lettura

Il C++ non ha istruzioni specifiche per l'input, ma nella libreria standard *<iostream>* è presente la definizione del dispositivo di input ***cin***; il dispositivo di input standard è la tastiera.
L'operatore *>>* trasferisce ciò che viene digitato sulla tastiera alla variabile specificata:

```
cin >> variabile;
```

Dopo la lettura, il cursore viene portato su una nuova riga.

Quando il programma incontra una istruzione di lettura, l'esecuzione si blocca finché l'utente non ha inserito un valore e premuto il tasto di invio; il valore inserito deve essere compatibile con il tipo della variabile, altrimenti ne risente il buon funzionamento del programma (potrebbe interrompersi oppure fornire risultati senza senso).

Nota

In una stessa istruzione possono essere specificate più variabili; in tal caso i valori devono essere inseriti uno dopo l'altro, separati da uno spazio oppure da un invio.

```
cin >> var1 >> var2 >> ... >> varN;
```

Per la lettura del carattere spazio non è possibile utilizzare *cin* perché lo spazio è sempre considerato un separatore e mai un carattere acquisibile. Per leggere uno spazio è possibile usare la funzione *getchar()*.

Nota

La funzione getline

La funzione ***getline()*** permette l'acquisizione di una qualsiasi sequenza di caratteri contenente anche uno o più caratteri spazio. La sintassi è:

```
getline(cin, variabile);
```

dove *cin* è il dispositivo standard di input (tastiera) e *variabile* è il contenitore in cui viene memorizzato il valore letto.

La funzione fflush

La funzione ***fflush()*** permette di eliminare i caratteri letti e non ancora assegnati ad una variabile; è utile dopo una lettura dal dispositivo *cin* mediante l'operatore *>>*.

```
fflush(stdin);
```

dove il parametro ***stdin*** (contrazione di standard input) indica il dispositivo di input standard secondo la notazione dell'ANSI C.

Se vengono introdotti più valori rispetto al numero delle variabili indicate dopo l'operatore >>, questi rimangono in un'area speciale chiamata **buffer** di input; alla lettura successiva viene automaticamente trasferito alla variabile il primo valore contenuto nel buffer; la funzione *fflush()* provoca lo svuotamento del buffer di input.

Esempio 4.6

Uso di *cout e cin*.

```
...
cout << "Inserire un valore\n";
cin >> valore1;
cout << endl;
cout << "Inserire due valori ";
cin >> valore2 >> valore3;
fflush(stdin);
...
```

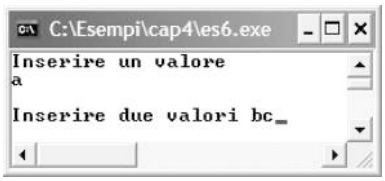


Figura 4.3 *Esecuzione dell'esempio*

`cout << "Inserire un valore\n";`
emissione di un messaggio costante; dopo la scrittura del messaggio il cursore si posiziona su una nuova riga grazie alla sequenza di escape *\n*;

`cin >> valore1;`
lettura della variabile; l'esecuzione si arresta finché l'utente scrive il valore e preme il tasto di invio; dopo la lettura il cursore si posiziona su una nuova riga;

`cout << endl;`
questa istruzione lascia una riga vuota;

`cout << "Inserire due valori ";`
dopo la scrittura del messaggio il cursore rimane posizionato sulla stessa riga, sul carattere immediatamente a destra (per questo alla fine del messaggio è presente uno spazio);

`cin >> valore2 >> valore3;`
lettura delle variabili; i valori devono essere separati da uno spazio o da un invio;

`fflush(stdin);`
svuota il buffer di input.

4.2 VARIABILI

Una **variabile** si può considerare come un contenitore in cui si devono introdurre dei valori. I valori possono **variare** durante l'elaborazione.

Ogni variabile è caratterizzata da un **nome** e dal **tipo di dati** che può contenere.

Un tipo di dati identifica quali sono i valori permessi (dipende da come i dati vengono rappresentati nella memoria), le operazioni elementari possibili e le costanti predefinite.

Dicendo che *x* è una variabile di tipo intero si intende dire che una certa zona della memoria (per esempio di due byte) viene identificata con questo nome.
I valori che possono essere contenuti nella variabile *x* sono solo numeri interni all'intervallo di definizione (per la rappresentazione con due byte l'intervallo è tra -32.768 e 32.767).
Le operazioni possibili su questa variabile sono solo quelle definite per i numeri interi.

Nota Si noti la differenza tra la variabile *x* e la costante "*x*"; la variabile *x* rappresenta un valore del tipo associato alla variabile; la costante "*x*" rappresenta il carattere *x* stesso.

Nomi delle variabili

È consigliabile scegliere sempre **nomi significativi** per le variabili, in modo che sia semplice ricordare a che cosa servono.

Tipo delle variabili

A volte per una variabile la **scelta del tipo** può avvenire tra più alternative.
Per esempio alcuni dati numerici potrebbero essere contenuti tanto in variabili numeriche che in variabili alfanumeriche. Di solito si usano variabili **numeriche** se sui dati devono essere effettuate **operazioni aritmetiche**, altrimenti si usano variabili **alfanumeriche**.

> Un codice postale normalmente viene memorizzato in una variabile alfanumerica.

Linguaggi tipizzati e debolmente tipizzati

In alcuni linguaggi non è necessario **dichiarare il tipo** delle variabili e qualche volta nemmeno **dichiarare le variabili**.
I linguaggi in cui è necessario dichiarare il tipo delle variabili si dicono **tipizzati**; dopo che la variabile è stata dichiarata è possibile assegnarle solo dati di tipo compatibile.
I linguaggi in cui non è necessario dichiarare il tipo delle variabili si dicono **debolmente tipizzati**; in questo caso una variabile può contenere di volta in volta un dato di qualsiasi tipo.
Quando non è necessario nemmeno dichiarare il nome delle variabili, le variabili cominciano ad esistere la prima volta che vengono utilizzate; questo può creare problemi in caso di errori di digitazione, infatti un nome scritto in modo errato non viene segnalato come errore, ma viene interpretato come una nuova variabile.

4.3 VALORIZZAZIONE DI VARIABILI

Tutte le variabili devono essere **inizializzate**.

Con la dichiarazione viene riservato alla variabile uno spazio di memoria, identificato dal nome della variabile; la variabile è predisposta a contenere dati del tipo indicato ma non assume un **valore** definito se non le viene associato un dato in modo esplicito.
Ogni variabile usata nel programma deve quindi essere valorizzata in modo opportuno la prima volta che compare.

> Alcuni interpreti o compilatori possono assegnare alle variabili non inizializzate esplicitamente dei **valori di default**, per esempio 0 per le variabili numeriche o degli spazi per quelle alfanumeriche, ma altre volte il valore della variabile può restare indefinito.
>
> **Nota**

I modi per introdurre valori in una variabile sono essenzialmente due:

- **acquisizione** del valore dall'esterno (operazione di lettura o **input**: è l'utente che inserisce il valore);
- **assegnazione** all'interno del programma (è il programma che assegna un valore).

> La forma più semplice di assegnazione è *variabile* ← *costante*.
>
> **Nota**

> Il valore della variabile non viene modificato se la variabile viene stampata o assegnata a un'altra variabile: rimane inalterato finché non viene cambiato con un'altra operazione di lettura o assegnazione.
>
> **Nota**

Il valore acquisito o assegnato deve essere di **tipo compatibile** con quello della variabile; il valore specificato viene inserito nella variabile.

Errori dovuti al tipo di dati usato

L'inserimento in una variabile di un valore non corrispondente al **tipo** della variabile può essere fonte di **errori**.
L'**assegnazione** di un valore non ammesso viene riconosciuta e segnalata dal compilatore.
L'inserimento (con una istruzione di **input**) di un valore non ammesso può provocare un errore di esecuzione.

Conversioni di tipo

È possibile **assegnare** a una variabile un valore (costante o variabile) di **tipo diverso** in due casi:

- quando il tipo di destinazione "comprende" il tipo di partenza, cioè quando il tipo finale è in grado di rappresentare correttamente i valori del tipo di partenza; in questo caso il compilatore effettua una **conversione implicita** da un tipo all'altro;

  Per esempio è possibile assegnare un numero intero a una variabile reale.

- quando è possibile convertire in qualche modo il tipo di partenza nel tipo di destinazione; in questo caso bisogna effettuare esplicitamente la conversione utilizzando opportune **funzioni di conversione** oppure un **cast** (o **typecast**).

  Per esempio è possibile convertire una stringa in un numero se la stringa contiene effettivamente un valore che può essere considerato un numero del tipo desiderato.

Nota

Le funzioni predefinite sono spiegate nel paragrafo C++ 4.8.

Le variabili

C++ 4.3

Il C++ è un linguaggio **tipizzato**.
Tutte le variabili devono essere dichiarate esplicitamente; se viene usata una variabile non dichiarata il compilatore segnala l'errore.

La **dichiarazione delle variabili** ha la forma:

```
tipo nomeVariabile;
```

In una stessa istruzione si possono dichiarare più variabili dello stesso tipo.

```
tipo nome1, ... , nomeN;
```

Variabili di tipo diverso vanno dichiarate in istruzioni separate.

```
tipo nome1;
...
tipo nomeN;
```

Se ci sono variabili dichiarate ma non utilizzate, alcuni compilatori (non GNU C++) producono un messaggio di avvertimento (warning).

I tipi di dati

C++ 4.4

Il C++ offre molti tipi di dati.

I principali tipi di dati semplici standard (o predefiniti) sono:

- ***short int*** o ***short***: numeri interi rappresentati con 16 bit;
- ***long int*** o ***long***: numeri interi rappresentati con 32 bit;
- ***int***: a seconda della piattaforma è equivalente a *short* (2 byte) oppure a *long* (4 byte);
- ***unsigned short***: numeri interi positivi rappresentati con 16 bit;
- ***unsigned int***: numeri interi positivi rappresentati con 16 o 32 bit (come *int*);
- ***unsigned long***: numeri interi positivi rappresentati con 32 bit;

Tabella 4.1 *Tipi interi*

Tipo	Range	Misura in byte	Costante valore massimo
short	-32.768 .. 32.767	2	SHRT_MAX
unsigned short	0 .. 65.535	2	USHRT_MAX
int	come *short* o *long*	2 o 4	INT_MAX
unsigned int	come *unsigned short* o *unsigned long*	2 o 4	UINT_MAX
long	-2.147.483.648 .. 2.147.483.647	4	LONG_MAX
unsigned long	0 .. 4.294.967.295	4	ULONG_MAX

Nota

Le costanti relative al valore massimo di ogni tipo dipendono dalla piattaforma e sono definite nella libreria standard *<limits.h>*. La libreria *<limits.h>* viene inclusa automaticamente quando si usa il namespace *std*.

- ***float***: numeri reali rappresentati con la notazione IEEE 754 a 4 byte;
- ***double***: numeri reali rappresentati con la notazione IEEE 754 a 8 byte;
- ***long double***: numeri reali rappresentati con la notazione IEEE 754 a 12 byte;

Tabella 4.2 *Tipi reali*

Tipo	Range	Cifre significative	Misura in byte
float	-3.4E38 .. -1.5E-45 per valori negativi 1.5E-45 .. 3.4E38 per valori positivi	7-8	4
double	-1.7E308 .. -5.0E-324 per valori negativi 5.0E-324 .. 1.7E308 per valori positivi	15-16	8
long double	-1.1E4932 .. -3.4E-4932 per valori negativi 3.4E-4932 .. 1.1E4932 per valori positivi		12

ATTENZIONE

Le rappresentazioni dei numeri sono finite e questo può causare degli errori.

- ***bool***: valori booleani (*true* e *false*);

Nota

Una variabile booleana viene rappresentata in memoria con un solo byte e quindi non contiene le parole *true* o *false* ma i valori 0 o 1: *false* corrisponde a 0, *true* a 1 (quindi nell'ordinamento *false* precede *true*).

- ***char***: caratteri del codice ASCII, che occupano un byte ciascuno;

Il tipo *char* è visto come un sottoinsieme del tipo *int* e pertanto può essere usato anche per definire variabili contenenti numeri interi compresi nel range -128 .. 127. Addirittura esiste il tipo **unsigned char** adatto per contenere numeri interi compresi nel range 0 .. 255.

Nota

- ***wchar_t***: caratteri del codice Unicode, che occupano due byte ciascuno;
- ***string***: sequenza di caratteri del codice ASCII (stringa).

In C++ le stringhe sono oggetti: *string* non è un tipo ma una classe; i metodi per la gestione delle stringhe sono spiegati nel paragrafo C++ 4.9.

Nota

Sono inoltre disponibili tipi di **dati strutturati** e i **puntatori**.

I dati strutturati sono spiegati nei capitoli 7 e 9, i puntatori nel capitolo 8.

Nota

È anche possibile definire **nuovi tipi di dati**.

Esempio 4.7

Richiesta e stampa dei dati di un libro.

```
#include <iostream>
using namespace std;

int main(void){
    string titolo, autore, editore;
    int numPagine;
    float prezzo;
    cout << "Inserire il titolo\n";
    getline(cin, titolo);
    cout << "Inserire l'autore\n";
    getline(cin, autore);
    cout << "Inserire l'editore\n";
    getline(cin, editore);
    cout << "Inserire il numero di pagine\n";
    cin >> numPagine;
    cout << "Inserire il prezzo\n";
    cin >> prezzo;
    cout << endl << titolo;
    cout << endl << autore;
    cout << endl << editore;
    cout << endl << numPagine;
    cout << endl << prezzo;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

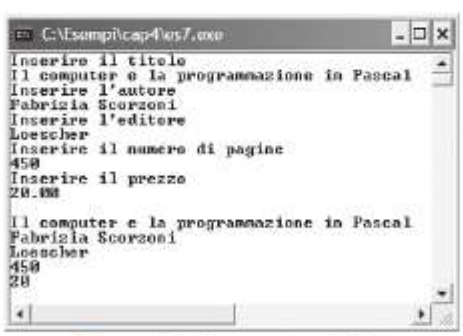


Figura 4.4 *Esecuzione dell'esempio*

Conversioni di tipo

Il C++ è un linguaggio **fortemente tipizzato**; il compilatore esegue molti controlli per verificare la compatibilità di tipo nelle assegnazioni.

Sono consentite assegnazioni fra tipi diversi solo se i valori possono essere **convertiti** da un tipo all'altro.
Alcune conversioni di tipo vengono eseguite **automaticamente** in modo **implicito**.
Gli assegnamenti incompatibili sono vietati, cioè generano errori di compilazione; in caso la compatibilità possa essere verificata solo durante l'esecuzione si può far uso di **conversioni esplicite**.

Conversioni implicite

Alcuni casi di conversione implicita sono:

- assegnazione di un valore numerico a una variabile numerica di range più ampio;

> I numeri interi possono essere assegnati a variabili in virgola mobile (non viceversa perché il range in virgola mobile è più ampio); però si può perdere precisione se si passa da *long* a *float* (da 64 a 32 bit).

> Questa conversione avviene anche quando si passa un numero come parametro a una funzione (cioè se una funzione ha un parametro di tipo *double* si può passargli un numero intero).
>
> **Nota**

- conversione di tipi *char* e *short* in *int* durante la valutazione di espressioni; se l'espressione contiene un *long* tutto viene convertito in *long*;
- conversione dei numeri interi in *double* o *float* durante la valutazione di espressioni che contengono un operando di questo tipo; in tal caso anche le operazioni sono eseguite con l'aritmetica in virgola mobile;
- conversione da un tipo intero a un tipo intero più piccolo; vengono eliminati i bit più significativi; se il valore è troppo grande per il tipo di destinazione il valore viene modificato e può essere di segno diverso;
- conversione da un tipo reale a un tipo reale più piccolo; si può avere perdita di precisione;
- conversione di caratteri in stringhe quando si usano gli operatori + o += nelle espressioni con stringhe.

Conversioni esplicite: cast

Nella sintassi C un **cast** è costituito da un nome di tipo tra parentesi tonde che precede il dato da convertire.

```
(tipo)espressione;
```

Le specifiche ISO C++ introducono per il cast il nuovo formato (del tutto equivalente):

```
static_cast<tipo>(espressione);
```

la conversione avviene in fase di compilazione, quindi per evitare errori di esecuzione bisogna essere certi che l'espressione da convertire assuma un valore lecito (cioè supportato da *Tipo*);

Si può usare un cast per le conversioni da un tipo reale a un tipo intero, ma si può avere perdita di precisione, ottenere zero o, se il valore è troppo grande, infinito.

In pratica la conversione non avviene implicitamente ma bisogna richiederla con un cast se la trasformazione può provocare una perdita di informazione. Spesso i compilatori segnalano questa situazione con un warning (avvertimento) pur consentendo la compilazione.

Per evitare un warning nell'assegnare a una variabile *int* un'espressione con numeri reali, bisogna convertirla a *int* usando un cast:

```
int n = (int)reale;
```
oppure
```
int n = static_cast<int>(reale);
```

Conversioni esplicite: schemi di conversione

Per convertire una **stringa** (oggetto della classe *string*) che contiene un valore numerico in un **numero** e viceversa, non esistono delle funzioni di libreria. Una possibile soluzione è l'utilizzo di un oggetto di appoggio della classe *stringstream* (e/o *ostringstream*).

Nel paragrafo C++ 4.9 sono mostrati due schemi di conversione stringa-numero e numero-stringa attraverso le classi *stringstream* e *ostringstream*.
Le classi *stringstream* e *ostringstream* sono spiegate nel capitolo 16.

Nota

Situazioni da considerare per evitare errori

L'assegnazione a una variabile di un valore di tipo diverso di solito viene segnalata come **errore di compilazione**.
L'inserimento (con una istruzione di input) in una variabile di un valore di tipo diverso può causare un **errore di esecuzione**.

Variabili numeriche intere

Per i numeri interi normalmente si usa come tipo ***int***, a volte *long*.
I valori *byte* e *short* vengono sempre trasformati in *int* prima di essere valutati; servono solo per memorizzare dati ma non vengono mai usati per fare operazioni.

L'aritmetica intera è quella modulare in **complemento a 2**.

Variabili numeriche reali

Per i numeri reali normalmente si usa il tipo ***double***.

Per i numeri in virgola mobile si usa lo standard **IEEE 754** sia per la rappresentazione che per l'aritmetica.
Si possono verificare overflow se i valori sono troppo grandi o underflow se sono troppo piccoli.

Lo zero può essere sia positivo che negativo; confrontandoli si ottiene l'uguaglianza ma nei calcoli possono produrre risultati diversi: 1/0 è più infinito ma 1/-0 è meno infinito.

Nota

Si possono assegnare a variabili reali numeri interi, che vengono convertiti implicitamente in reali.

Nota

Variabili di tipo carattere

Una istruzione di lettura di una variabile di tipo *char* accetta solo il **primo carattere** inserito e lo considera un carattere anche se si tratta di una cifra.

Variabili di tipo stringa

In una istruzione di lettura di una stringa viene accettata qualsiasi **sequenza di caratteri**; se viene inserito un numero viene interpretato come una sequenza di cifre.

Variabili booleane

Una variabile booleana può essere usata in una istruzione di **lettura**, inserendo 0 per *false* e 1 per *true*. L'inserimento di un valore diverso da 0 e 1 non genera un errore ma provoca l'assegnazione alla variabile del valore 255.

Si può assegnare a una variabile booleana uno dei valori predefiniti ***true*** o ***false***, uno dei valori 0 o 1, oppure una espressione relazionale o logica (che dia cioè come risultato un valore vero o falso).

La stampa di una variabile booleana produce uno dei valori 0 o 1.

4.4 COSTANTI

Le **costanti** sono dati che **non variano** da una elaborazione all'altra, o almeno durante una stessa elaborazione.

Le costanti possono essere di vario tipo, per esempio numeriche, alfanumeriche o booleane. Le costanti alfanumeriche di solito vanno tra apici o tra virgolette, in base alla sintassi del linguaggio.

Esempi di costanti:

3, -24, 15.8	costanti numeriche, intere e reali;
's', 'n'	costanti di tipo carattere;
"ciao", "come stai?"	costanti di tipo stringa.

In molti casi è conveniente assegnare un **nome simbolico** a un valore costante:

- per ricordarne meglio il significato;

  Per esempio in matematica si usa abitualmente π al posto di 3,14...

- per rendere in seguito più facilmente modificabile il programma (se la costante viene usata più volte basta modificare il valore solo nella dichiarazione della costante simbolica).

ESEMPI RELATIVI ALL'USO DI COSTANTI E DI VARIABILI

Esempio 4.8

Calcolo del perimetro di un quadrato.

Il numero dei lati è il valore costante 4.

I	lato	reale	lato del quadrato
O	perimetro	reale	perimetro

```
inizio
    scrivi "Inserire la misura del lato"
    leggi lato
    perimetro ← lato x 4
    scrivi "Il perimetro è " perimetro
fine
```

Il numero 4 potrebbe essere sostituito da una costante simbolica:

```
inizio
    numLati ← 4
    scrivi "Inserire la misura del lato"
    leggi lato
    perimetro ← lato x numLati
    scrivi "Il perimetro è " perimetro
fine
```

così si può trasformare questo algoritmo in uno per il calcolo del perimetro di un triangolo equilatero semplicemente modificando il valore della costante:

```
numLati ← 3
```

Esempio 4.9

Calcolo del perimetro di un poligono regolare con numero di lati qualsiasi.

Il numero dei lati è variabile.

I	lato	reale	lato del poligono
I	numLati	intero	numero di lati del poligono
O	perimetro	reale	perimetro del poligono

```
inizio
    scrivi "Inserire la misura del lato"
    leggi lato
    scrivi "Inserire il numero dei lati"
    leggi numLati
    perimetro ← lato x numLati
    scrivi "Il perimetro è " perimetro
fine
```

Le costanti

C++ 4.5

Valori costanti

I valori costanti utilizzabili dipendono dal **tipo** di costante:

- costanti **intere**: numeri con o senza segno compresi nell'intervallo di definizione;

Nota

Le costanti intere possono utilizzare cifre ottali, decimali o esadecimali; se un numero inizia con 0 è ottale, se inizia con 0x o 0X è esadecimale.

- costanti **reali**: numeri espressi nella notazione con il punto decimale o nella notazione esponenziale; se il numero ha più cifre significative di quelle ammesse dalla precisione disponibile, le cifre in eccesso vengono troncate o arrotondate;
- costanti di tipo **carattere**: sono costituite da un solo carattere racchiuso tra apici; si può specificare un carattere anche con il codice ASCII (per esempio *65* per il carattere *'A'*);
- costanti di tipo **stringa**: devono sempre essere racchiuse tra virgolette e possono comprendere qualsiasi carattere; per inserire il simbolo " in una stringa si deve far precedere da una barra rovesciata (simbolo \); la barra rovesciata si usa per definire un carattere di **escape** (o di uscita) e la sequenza \" si chiama sequenza di escape; per inserire una \ basta usare \\; \n indica una riga nuova, \t un carattere di tabulazione. Tutti i caratteri delle stringhe sono in **ASCII**.
- costanti **booleane**: i valori *true* e *false*.

Costanti simboliche

Si possono dichiarare **costanti simboliche** con il qualificatore **const**:

```
const tipo nomeCostante = valore;
```

Per maggiore leggibilità di solito la dichiarazione delle costanti precede la dichiarazione delle variabili.

Alle costanti dichiarate in questo modo è possibile assegnare un valore solo al momento della dichiarazione, poi non è più possibile modificarne il valore: non è possibile assegnare un valore a una costante nel corso del programma.

Più in generale a una costante è possibile assegnare un'**espressione**; l'espressione deve essere valutabile a tempo di compilazione (può contenere per esempio operatori aritmetici o funzioni predefinite).

```
const tipo nomeCostante = espressione;
```

Si possono dichiarare costanti assegnando valori di tipi interi, reali, caratteri e stringhe.

```
const double e = 2.7182818;      costante di tipo reale;
const int n = 10+2;              costante di tipo intero, risultato dell'espressione;
const char c = 'a';              costante di tipo carattere;
const char cc = 65;              costante di tipo carattere;
const string s = "Ciao mondo"; costante di tipo stringa.
```

La direttiva define in C

Si possono dichiarare costanti anche con la modalità del C attraverso la direttiva ***#define***:

```
#define nomeCostante valore
```

La costante è usabile dalla riga seguente a quella in cui è stata definita; affinché sia visibile in tutto il programma la costante va definita di seguito alle direttive di inclusione delle librerie.

Il tipo della costante è automaticamente dedotto dal valore indicato.

```
#define e 2.7182818          costante di tipo reale;
#define n 10+2               costante di tipo intero, risultato dell'espressione;
#define c 'a'                costante di tipo carattere;
#define cc 65                costante di tipo carattere o intero;
#define s "Ciao mondo"       costante di tipo stringa.
```

4.5 ASSEGNAZIONE

Una operazione di **assegnazione** assume la forma:

variabile ← espressione

la punta della freccia è rivolta verso la variabile che deve ricevere un nuovo valore: la variabile a sinistra riceve come contenuto il valore dell'espressione posta a destra.

Nota

Si è scelto di utilizzare questa notazione per maggior chiarezza; a volte come simbolo di assegnazione viene utilizzato =, ma poiché crea ambiguità con l'operatore di confronto di uguaglianza si consiglia di non utilizzarlo.

L'**espressione** può essere:

- una **costante** di tipo compatibile;

  variabile ← costante

  Se *x* è una variabile numerica, l'assegnazione *x* ← *3* associa il valore 3 alla variabile *x*; ogni volta che *x* compare in seguito in una espressione viene sostituita dal valore 3.

- una **variabile** di tipo compatibile;

  variabile ← variabile2

  Nell'assegnazione *y* ← *x* viene assegnato alla variabile *y* il valore della variabile *x*.

Esempio 4.10

Scambio di due variabili.

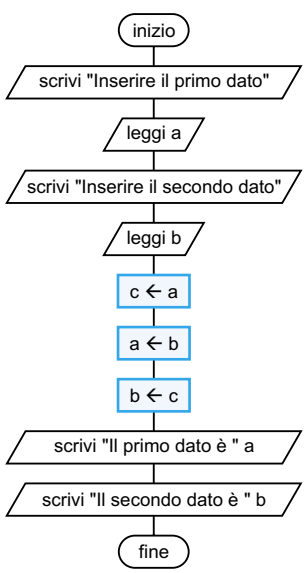
Per scambiare il contenuto di due variabili *a* e *b* è necessaria una variabile di appoggio *c*.

Per scambiare il contenuto di due variabili *a* e *b* non basta fare *a*←*b*, *b*←*a* infatti al termine di tale operazione entrambe le variabili hanno lo stesso valore, quello inizialmente contenuto in *b*, e si perde il valore contenuto in *a*.

Nota

I/O	a, b	qualsiasi tipo	valori da scambiare
	c	dello stesso tipo di a e b	variabile di appoggio per lo scambio

```
inizio
    scrivi "Inserire il primo dato"
    leggi a
    scrivi "Inserire il secondo dato"
    leggi b
    c ← a
    a ← b
    b ← c
    scrivi "Il primo dato è " a
    scrivi "Il secondo dato è " b
fine
```



L'algoritmo è valido indipendentemente dal tipo di dati; ovviamente la variabile di appoggio deve essere dello stesso tipo delle altre due.

Nota

- una **espressione** contenente costanti, variabili e operatori che una volta valutata produca un valore costante di tipo compatibile;

  variabile ← espressione

  le variabili che compaiono nell'espressione devono avere un valore, che viene usato per valutare l'espressione.

  x ← 3 + 2
  viene calcolata l'espressione 3+2 e il risultato 5 viene assegnato alla variabile *x*.

  y ← x + 4
  x viene sostituito dal valore 5, e sommato a 4; il risultato, 9, viene assegnato alla variabile *y*.

Assegnazione

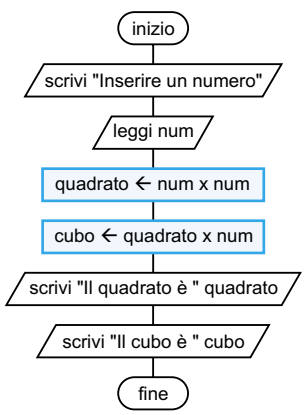
L'**assegnazione** è un'operazione **conservativa a destra** e **distruttiva a sinistra**: le variabili che compaiono a destra dell'operatore di assegnazione mantengono il loro valore, mentre la variabile che compare a sinistra perde il valore che aveva in precedenza per assumere il nuovo valore, risultato dell'espressione che compare a destra.

Esempio 4.11

Calcolare il quadrato e il cubo di un numero.

I	num	intero	numero di cui si vuole il quadrato e il cubo
O	quadrato	intero	quadrato del numero inserito
O	cubo	intero	cubo del numero inserito

```
inizio
    scrivi "Inserire un numero"
    leggi num
    quadrato ← num x num
    cubo ← quadrato x num
    scrivi "Il quadrato è " quadrato
    scrivi "Il cubo è " cubo
fine
```



La **stessa variabile** può comparire sia a sinistra che a destra in una operazione di assegnazione. La variabile deve avere già un valore che viene sostituito alla variabile nell'espressione a destra, in modo che l'espressione possa essere calcolata e il suo risultato assegnato alla variabile che compare a sinistra.

x ← x + 1
viene prima calcolata l'espressione a destra del segno uguale, sostituendo alla variabile *x* che compare in questa espressione il valore che aveva assunto durante l'ultima operazione, supponiamo 5; il risultato dell'espressione così calcolata (5+1) viene assegnato alla stessa variabile *x*; da questo momento in poi la variabile *x* ha il nuovo valore 6.

Incremento del valore di una variabile

Per **sommare** un valore al contenuto di una variabile, tale variabile deve comparire sia a sinistra che a destra del simbolo di assegnazione; infatti il valore precedente della variabile viene utilizzato nel calcolo e il risultato viene poi assegnato alla stessa variabile che assume così un nuovo valore.

x ← x + x
viene prima calcolata l'espressione a destra del segno uguale, sostituendo alla variabile *x* che compare in questa espressione il valore che aveva assunto durante l'ultima operazione, supponiamo 6; il risultato dell'espressione così calcolata (6+6) viene assegnato alla stessa variabile *x*; da questo momento in poi la variabile *x* ha il nuovo valore 12.

L'ASSEGNAZIONE

C++ 4.6

In C++ l'operatore di assegnazione è =.

Una **assegnazione** assume la forma:

```
variabile = espressione;
```

ESEMPIO 4.12

Codifica dell'esempio 4.11

```
#include <iostream>
using namespace std;

int main(void){
    int num, quadrato, cubo;
    cout << "Inserire un numero ";
    cin >> num;
    quadrato = num * num;
    cubo = quadrato * num;
    cout << "Il quadrato e' ";
    cout << quadrato << endl;
    cout << "Il cubo e' " << cubo;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

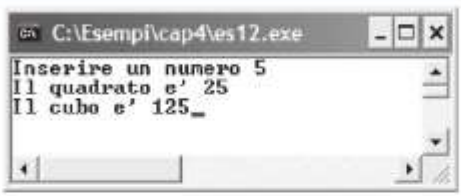


Figura 4.5 *Esecuzione dell'esempio*

4.6 ESPRESSIONI ED OPERATORI

Nelle **espressioni** si possono usare molti **operatori** diversi in base al tipo di operandi che compaiono nelle espressioni:

operandi numerici	operatori per le quattro operazioni aritmetiche e la potenza
stringhe	concatenazione
tutti gli operandi	operatori relazionali ($= \neq > < \geq \leq$)
operandi booleani	operatori logici (not and or)

Gli operatori seguono **regole di precedenza** che dipendono dal linguaggio.
Si possono modificare le regole di precedenza con le **parentesi**.

ESPRESSIONI E OPERATORI

C++ 4.7

Gli operatori sono valutati in base alla **priorità**.

Nota

Attenzione: gli operatori relazionali hanno priorità più alta dei booleani.

Non è garantito che l'ordine in cui sono valutati gli operatori con la stessa precedenza sia sempre da sinistra a destra.

È possibile inserire **parentesi** tonde in una espressione per precisare l'ordine di valutazione.

Tabella 4.3 *Priorità degli operatori, dalla più alta alla più bassa*

di risoluzione di visibilità, di riferimento globale	::
di accesso	[] . () -> *typeid* var++ var--
unari	++var --var +esp -esp ~ ! *punt &var *sizeof new delete*
cast	(tipo)esp static_cast dynamic_cast
moltiplicativi	* / %
additivi	+ -
di shift	<< >>
relazionali	< > <= >=
di uguaglianza	== !=
AND bit a bit	&
OR esclusivo bit a bit	^
OR bit a bit	|
AND logico	&&
OR logico	||
condizionale	?:
assegnamento	= += -= *= /= %= >>= <<= &= ^= |=

Nota

Gli operatori binari sono associativi a sinistra, cioè per esempio *a+b+c* equivale a *(a+b)+c*: le assegnazioni sono associative a destra, cioè *a=b=c* equivale a *a=(b=c)*.

Con gli operatori &&, || e ?: quando è possibile si evita la valutazione del secondo operando. In tutti gli altri casi gli operandi vengono valutati prima che l'operazione sia eseguita.

Nota

Gli operatori relazionali e logici sono spiegati nel capitolo 5.

Operatori aritmetici

Gli operatori utilizzabili nelle **espressioni aritmetiche** sono:

+	addizione	*	moltiplicazione	%	resto della divisione tra numeri interi.
-	sottrazione	/	divisione (sia per gli interi che per i reali)		

```
risultato = dividendo / divisore;

resto = dividendo % divisore;
```
equivale a
```
resto = dividendo - risultato * divisore;
```

La moltiplicazione, la divisione e il modulo hanno la precedenza rispetto ad addizione e sottrazione. L'operatore di divisione / restituisce un valore reale se almeno uno degli operandi è reale.

L'operatore % può essere usato solo con operandi interi, tutti gli altri operatori accettano operandi sia interi che reali.
Il risultato è intero se tutti gli operandi sono interi, altrimenti è reale.

Per gli operatori di divisione e resto se il **divisore** è **0** si verifica un **errore di esecuzione**.

Nota

Il risultato delle operazioni è corretto se risulta compreso tra i limiti di definizione dell'intervallo dei numeri interi, altrimenti si possono verificare due situazioni diverse: il risultato può essere calcolato lo stesso ma non essere valido (poiché viene calcolato con le regole dell'aritmetica modulare), oppure si può verificare un errore di esecuzione (overflow).

ESEMPIO 4.13

Calcolare il risultato e il resto della divisione di due numeri interi.

```
#include <iostream>
using namespace std;

int main(void){
    int a, b, c;
    a = 14;
    b = 5;
    c = a / b;
    cout << "Il risultato di ";
    cout << a << " : " << b << " e' " << c << endl;
    c = a % b;
    cout << "Il resto di ";
    cout << a << " : " << b << " e' " << c << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

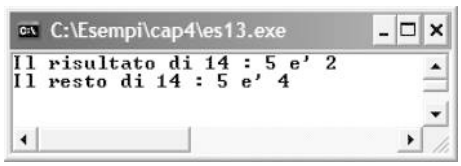


Figura 4.6 *Esecuzione dell'esempio*

+=	-=	*=	/=	%=

Gli operatori += -= *= /= %= sono composti da un operatore di assegnamento e da un operatore numerico.

```
variabile op= espressione
```

corrisponde a

```
variabile = variabile op espressione
```

```
x = 10;
x += 2;
```
x vale 12.

++	incremento
--	decremento

Gli operatori ++ e -- sono operatori unari e possono essere usati in modo prefisso (prima) o postfisso (dopo) rispetto all'operando; il loro comportamento varia in base alla posizione; l'operatore prefisso modifica l'operando prima di usarne il valore, quello postfisso prima usa il valore e poi modifica l'operando.

```
x = 1;
y = x++;
```
y vale 1 e x 2.

```
x = 1;
y = ++x;
```
y vale 2 e x 2.

CONCATENAZIONE DI STRINGHE

+	operatore di concatenazione di stringhe

L'operatore + è usato per l'addizione tra numeri e per concatenare le stringhe; viene interpretato come concatenazione di stringhe se uno degli operandi è una stringa.

Nota

Si possono aggiungere spazi a una stringa concatenandoli con + " ".
Si può concatenare un carattere a una stringa.

```
string s = "stringa";
char ch = 'c';
string s1 = s + " " + ch;
```

s1 vale “stringa c”.

Se uno degli operandi è numerico, l’operatore + di concatenazione genera un errore di compilazione.

4.7 TRACE

Si può controllare l’algoritmo (per individuare eventuali errori) eseguendone la **traccia** (trace) per esaminare il contenuto delle variabili durante il processo esecutivo.
La traccia può essere eseguita manualmente sulla descrizione dell’algoritmo o sul programma, costruendo la **tabella di traccia**, oppure può essere ottenuta in modo automatico dall’esecuzione del programma inserendo opportune **istruzioni di output** o eseguendo il programma in un **ambiente di debug**.

Tabella di traccia

Nella prima colonna della **tabella di traccia** vanno indicate le istruzioni da eseguire; nelle colonne successive vanno riportati i valori delle variabili indicate nell’intestazione delle colonne, dopo l’esecuzione di ciascuna istruzione.

Tabella di traccia dell’algoritmo per lo scambio di due variabili

Controllo dei valori assunti dalle variabili dopo l’esecuzione di ogni istruzione:

	a	b	c
leggi a	1		
leggi b	1	2	
c ← a	1	2	1
a ← b	2	2	1
b ← c	2	1	1

Controllo dei valori delle variabili solo quando subiscono una variazione:

	a	b	c
leggi a	1		
leggi b		2	
c ← a			1
a ← b	2		
b ← c		1	

Trace ottenuta dall’esecuzione del programma

Per ottenere la traccia automaticamente durante l’esecuzione, basta inserire delle **istruzioni di output** per visualizzare il valore delle variabili che interessano.
Si possono controllare ad ogni passo i valori di tutte variabili o solo di alcune.

In genere la trace viene utilizzata per individuare errori di comportamento del programma; **conviene controllare**:

- le variabili che vengono modificate da operazioni di assegnazione per verificare se il risultato delle operazioni è quello atteso;
- le variabili in cui viene inserito un valore con un’operazione di input per verificare se il valore introdotto viene accettato in modo corretto.

ESEMPIO 4.14

Esempio di trace eseguita nel programma.

```
#include <iostream>
using namespace std;

int main(void){
    char a, b, c;
    cout << "Inserire il primo dato ";
    cin >> a;
    cout << "Inserire il secondo dato ";
    cin >> b;
    c = a;
    cout << "c= " << c << endl;
    a = b;
    cout << "a= " << a << endl;
    b = c;
    cout << "b= " << b << endl;
    cout << "Il primo dato e' " << a << endl;
    cout << "Il secondo dato e' " << b << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 4.7 *Esecuzione dell'esempio*

ESECUZIONE IN UN AMBIENTE DI DEBUG

ESECUZIONE CON *GDB*.

Il comando ***l 9*** visualizza il sorgente.
Il comando ***break 12*** imposta un breakpoint all'istruzione 12.
Il comando ***run*** inizia l'esecuzione; al breakpoint l'esecuzione si interrompe e si possono immettere comandi.
Il comando ***p*** visualizza il contenuto di una variabile.
Il comando ***s*** (o in alternativa ***n***) fa avanzare l'esecuzione di una istruzione alla volta.
Il comando ***c*** fa proseguire l'esecuzione fino al termine del programma.

Figura 4.8
Debug (con gdb) del programma per lo scambio di due variabili

ESECUZIONE CON IL DEBUG INTERNO

Impostare un breakpoint all'istruzione 12.
Si possono impostare punti di interruzione facendo clic nel margine grigio a sinistra della finestra di editor o usando i comandi del menu *Debug*, *Attiva/Disattiva Breakpoint*.

Attivare il debug con il comando *Debug*, *Debug*. Se la prima prima volta che si richiama il comando viene richiesto di ricostruire il progetto con le informazioni necessarie al debug, confermare la richiesta.
Posizionare il cursore nella riga del breakpoint e scegliere dalla scheda *Debug* il comando *Esegui dal cursore*.
Inserire i valori richiesti nella finestra di esecuzione e tornare al debug.
Per controllare i valori delle variabili posizionare il mouse su di esse; nella scheda *Debug* della finestra *Project Inspector* compariranno le osservazioni. In alternativa, dalla scheda *Debug* scegliere il comando *Nuova Osservazione* e impostare il nome della variabile.
Per avanzare l'esecuzione scegliere il comando *Step Succ.* controllando i risultati nelle varie finestre.

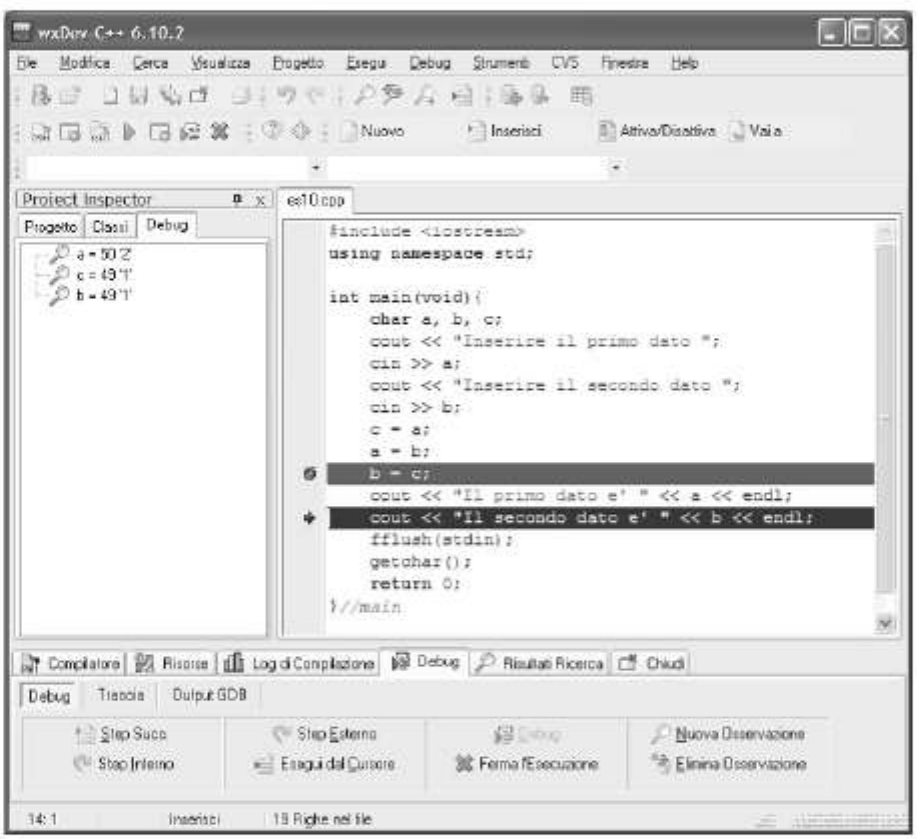


Figura 4.9 *Debug del programma per lo scambio di due variabili*

4.8 LE FUNZIONI PREDEFINITE

Una **funzione** calcola un risultato a partire da uno o più valori, chiamati **parametri** o **argomenti** della funzione.

Tutti i linguaggi di programmazione dispongono di molte **funzioni predefinite**; per utilizzarle basta conoscere il tipo di parametri da passare alla funzione e il tipo di risultato che la funzione restituisce.

Nota Il programmatore può definire delle funzioni personalizzate come si vedrà nel capitolo 6.

Le funzioni possono essere utilizzate quasi negli stessi punti in cui possono essere utilizzate le variabili, quindi per esempio a destra in istruzioni di assegnazione o come argomenti di istruzioni di scrittura.
Non possono essere usate a sinistra in istruzioni di assegnazione, o in istruzioni di lettura.

Le funzioni predefinite

C++

4.8

La libreria cmath

Il file di libreria ***<cmath>*** contiene le definizioni di funzioni matematiche e di costanti.

Costanti

`M_E`	e, la base dei logaritmi naturali;
`M_LOG2E`	il logaritmo in base 2 di e;
`M_LOG10E`	il logaritmo in base 10 di e;
`M_LN2`	il logaritmo naturale (in base *e*) di 2;
`M_LN10`	il logaritmo naturale (in base *e*) di 10;
`M_PI`	pi greco, cioè il rapporto tra la circonferenza e il diametro di un cerchio;
`M_PI_2`	la metà di pi greco;
`M_PI_4`	il quarto di pi greco;
`M_PI_4`	il quarto di pi greco;
`M_1_PI`	il reciproco di pi greco;
`M_2_PI`	il doppio del reciproco di pi greco;
`M_2_SQRTPI`	il doppio del reciproco della radice quadrata di pi greco;
`M_SQRT2`	la radice quadrata di 2;
`M_SQRT1_2`	il reciproco della radice quadrata di 2.

Nota

In realtà sono i valori *double* più vicino a questi valori (che non possono essere rappresentati con un numero finito di cifre).

Le funzioni hanno parametri ***double*** e restituiscono un valore dello stesso tipo; gran parte di esse possono essere richiamate con parametri di tipo diverso (e restituiscono un valore dello stesso tipo).

Funzioni per il calcolo di comuni funzioni aritmetiche

Funzione	Tipo parametri	Tipo risultato	Significato
abs(x)	reale o intero	reale o intero	valore assoluto
exp(x)	reale	reale	esponenziale (*e* elevato alla *x*)
log(x)	reale	reale	logaritmo naturale
log10(x)	reale	reale	logaritmo in base 10
pow(x, y)	reale, reale	reale	primo argomento elevato al secondo
sqrt(x)	intero o reale	dello stesso tipo	radice quadrata.

Funzioni per il calcolo di funzioni trigonometriche

Funzione	Tipo parametri	Tipo risultato	Significato
sin(a)	reale	reale	seno di un angolo *a* in radianti;
cos(a)	reale	reale	coseno di un angolo *a* in radianti;
tan(a)	reale	reale	tangente di un angolo *a* in radianti;
asin(a)	reale	reale	arcoseno di *a* tra -1.0 e 1.0; restituisce un valore in radianti tra -M_PI_2 e M_PI_2;
acos(a)	reale	reale	arcocoseno di *a* tra -1.0 e 1.0; restituisce un valore in radianti tra 0.0 e M_PI;
atan(x)	reale	reale	arcotangente di *a*; restituisce un valore in radianti tra -M_PI_2 e M_PI_2.

Funzioni per arrotondamenti

Funzione	Tipo parametri	Tipo risultato	Significato
ceil(a)	reale	reale	numero intero più piccolo maggiore o uguale ad *a*;
floor(a)	reale	reale	numero intero più grande minore o uguale ad *a*.

Nota

Entrambe le funzioni restituiscono un tipo reale nonostante il valore sia intero; per assegnarlo direttamente a una variabile intera conviene usare un cast evitando così un warning del compilatore.

La libreria cctype

Le funzioni per il riconoscimento e la gestione di un carattere sono definite nel file di libreria ***<cctype>***.

Funzioni per riconoscere un carattere

Funzione	Tipo parametri	Tipo risultato	Significato
isalpha(a)	intero o char	intero o bool	indica se *a* è un carattere alfabetico;
isdigit(a)	intero o char	intero o bool	indica se *a* è una cifra (in base 10);
isxdigit(a)	intero o char	intero o bool	indica se *a* è una cifra (in base 16);
isspace(a)	intero o char	intero o bool	indica se *a* è uno dei caratteri “spazio”, \n, \t, \r, \f;
iscntrl(a)	intero o char	intero o bool	indica se *a* è un carattere di controllo;
isupper(a)	intero o char	intero o bool	indica se *a* è una lettera maiuscola;
islower(a)	intero o char	intero o bool	indica se *a* è una lettera minuscola;
ispunct(x)	intero o char	intero o bool	indica se *a* è un carattere di punteggiatura.

Nota

Sia il parametro che il risultato delle funzioni possono essere di tipo intero perché in C++ i tipi *char* e *bool* sono sottoinsiemi degli interi.

Funzioni di conversione

Funzione	Tipo parametri	Tipo risultato	Significato
tolower(a)	intero o char	intero o char	converte il carattere in minuscolo;
toupper(x)	intero o char	intero o char	converte il carattere in maiuscolo.

La libreria cstdlib

Nel file di libreria ***<cstdlib>*** sono definite funzioni molto diverse tra loro (per l’allocazione dinamica della memoria, per la generazione di numeri casuali, ...).

Costante

`RAND_MAX`	valore massimo ritornato dalla funzione *rand()*; di solito è 32767.

Funzioni per il calcolo di valori casuali

Funzione	Tipo parametri	Tipo risultato	Significato
rand()	nessuno	intero	numero casuale compreso tra 0 e RAND_MAX;
srand(a)	intero	nessuno	imposta un seme.

Due sequenze di numeri casuali generati con l'impostazione dello stesso seme sono identiche. Per essere sicuri di ottenere sequenze diverse ad ogni generazione, bisogna impostare un seme diverso. La funzione ***time()*** definita nel file di libreria ***<ctime>*** restituisce il numero di secondi trascorsi da una certa data. Invocata come parametro della funzione *srand()*, garantisce generazioni di sequenze distinte.

```
srand(time(NULL));
```

Per ottenere un numero intero casuale *n* compreso nell'intervallo *min* e *max* (estremi compresi) si usa la formula:

```
n =(rand() % (max-min+1)) + min;
```

Esempio 4.15

Uso di funzioni matematiche.

```
#include <iostream>
#include <cmath>
using namespace std;

int main(void){
    float a = 100.15;
    cout << abs(-a) << endl;
    cout << sqrt(a) << endl;
    cout << ceil(a) << endl;
    cout << floor(a) << endl;
    cout << log(a) << endl;
    cout << log10(a) << endl;
    getchar();
    return 0;
}//main
```

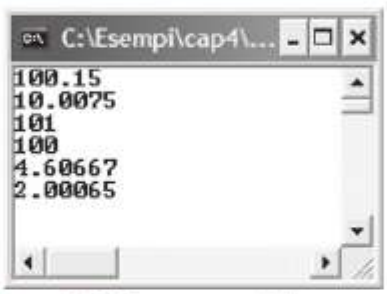


Figura 4.10 *Esecuzione dell'esempio*

Esempio 4.16

Simulazione di un lancio di un dado.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main(void){
    int n;
    srand(time(NULL));
    cout << "Lancio di un dado\n";
    n =(rand() % 6) + 1;
    cout << "Si e' ottenuto " << n << endl;
    getchar();
    return 0;
}//main
```

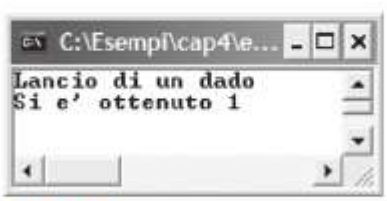


Figura 4.11 *Esecuzione dell'esempio*

Le stringhe

C++ 4.9

Le stringhe definite tramite *string* sono oggetti della classe ***string***.
La classe *string* offre molti metodi per la manipolazione di stringhe.

Classi, oggetti e tutti i termini relativi sono spiegati nel capitolo 10. **Nota**

Per usare i costruttori e i metodi della classe *string* bisogna includere il file di libreria ***<string>***; alcuni ambienti (come wxDev-C++) lo includono automaticamente con *<iostream>*. **Nota**

Costruttori

string ha diversi costruttori; i più utili sono:

Costruttore	Tipo parametri	Significato
string()	nessuno	crea un oggetto string con valore la stringa vuota "";
string(s)	string	crea un oggetto string con i caratteri della stringa *s*;
string(v)	char[]	crea un oggetto string con i caratteri contenuti nell'array *v*;
string(v, pos, n)	char[], intero, intero	crea un oggetto string con una parte dei caratteri contenuti nell'array *v* (*n* caratteri a partire da *pos*).

Il modo più comune di creare una stringa, però, è assegnare una costante stringa a una variabile di tipo *string*.

```
string stringa = "stringa";
```

Questo è un modo conciso che corrisponde all'uso di un costruttore.

```
string stringa = string("stringa");
```

Questo comportamento dipende dall'overloading dell'operatore di assegnazione =. **Nota**

In pratica per ogni sequenza di caratteri tra virgolette viene creato un oggetto *string* inizializzato con il valore tra virgolette; questo consente di impiegare una costante stringa ovunque sia richiesto un oggetto della classe *string*.

Dopo che un oggetto *string* è stato creato viene trattato come costante; nessun metodo di questa classe modifica l'oggetto stringa su cui opera: le stringhe sono oggetti **immutabili**.
Per ogni operazione che sembra modificare un oggetto *string*, in realtà viene creato un nuovo oggetto *string*, mentre l'oggetto originale resta inalterato.

```
string str = "uno";
str = "due";
```
la seconda assegnazione crea un nuovo oggetto stringa; modifica il valore del riferimento all'oggetto ma non il contenuto della prima stringa.

Metodi

Metodi principali

I metodi principali sono:

Metodo	Tipo parametri	Tipo risultato	Significato
length()	nessuno	intero	restituisce il numero di caratteri della stringa;
size()	nessuno	intero	restituisce il numero di caratteri della stringa;
at*(indice)*	intero	intero o char	restituisce il carattere alla posizione specificata (le posizioni partono da 0); il tentativo di accedere a una posizione minore di 0 o maggiore della lunghezza -1 genera una eccezione *out_of_range*.

```
int n = stringa.length();
```

```
stringa.at(0);
```
restituisce il primo carattere della stringa.

Si possono usare i metodi della classe *string* anche su costanti stringa, dopo averle convertite in un oggetto *string* attraverso un cast.

```
static_cast<string>("Ciao a tutti").length();
```

```
((string)"prova").at(0);
```
restituisce il carattere 'p';

METODO PER ESTRARRE SOTTOSTRINGHE

Metodo	Tipo parametri	Tipo risultato	Significato
substr(pos, n)	intero, intero	string	restituisce la sottostringa di *n* caratteri che inizia dalla posizione *pos*;

```
string saluto = "ciao a tutti";
string s = saluto.substr(0, 4);
```
s contiene "ciao".

METODI PER DETERMINARE LA POSIZIONE DI UN CARATTERE O DI UNA SOTTOSTRINGA NELLA STRINGA

Metodo	Tipo parametri	Tipo risultato	Significato
find(str, daPos)	string, intero	intero	restituisce la prima posizione di *str* >= di *daPos*;
find(c, daPos)	char o intero, intero	intero	restituisce la prima posizione di *c* >= di *daPos*;
rfind(str, daPosInd)	string, intero	intero	restituisce l'ultima posizione di *str* <= di *daPosInd*;
rfind(c, daPosInd)	char o intero, intero	intero	restituisce l'ultima posizione di *c* <= di *daPosInd*;
find_first_of(str, daPos)	string, intero	intero	restituisce la prima posizione di uno dei caratteri di *str* >= di *daPos*;
find_first_of(c, daPos)	char o intero, intero	intero	restituisce la prima posizione di *c* >= di *daPos*;
find_last_of(str, daPosInd)	string, intero	intero	restituisce l'ultima posizione di uno dei caratteri di *str* <= di *daPosInd*;
find_last_of(c, daPosInd)	char o intero, intero	intero	restituisce l'ultima posizione di *c* <= di *daPosInd*;
find_first_not_of(str, daPos)	string, intero	intero	restituisce la prima posizione >= di *daPos* del carattere della stringa diverso da uno dei caratteri di *str*;
find_first_not_of(c, daPos)	char o intero, intero	intero	restituisce la prima posizione >= di *daPos* del carattere della stringa diverso da *c* ;

find_last_not_of(str, daPosInd)	string, intero	intero	restituisce l'ultima posizione <= di *daPosInd* del carattere della stringa diverso da uno dei caratteri di *str*;
find_last_not_of(c, daPosInd)	char o intero, intero	intero	restituisce l'ultima posizione <= di *daPosInd* del carattere della stringa diverso da *c*;

Questi metodi restituiscono -1 se il carattere o la sottostringa non vengono trovati.

Metodi che sembrano modificare una stringa

Metodo	Tipo parametri	Significato
assign(str)	string	assegna alla stringa il contenuto di *str*;
assign(n, c)	intero, char o intero	assegna la stringa formata da *n* caratteri *c*;
append(str)	string	concatena *str* alla fine;
insert(pos, str)	intero, string	inserisce *str* a partire dalla posizione *pos*;
erase(pos, n)	intero, intero	rimuove *n* caratteri a partire da *pos*;
replace(pos, n, str)	intero, intero, string	sostituisce *n* caratteri a partire da *pos* con *str*;
swap(str)	string	scambia i contenuti delle due stringhe;

Questi metodi in realtà non modificano la stringa su cui operano ma restituiscono una nuova stringa.

Il metodo *append()* esegue la stessa operazione dell'operatore + (overloading dello stesso).

```
s = s.append("aggiungi");
```
equivale a
```
s = s + "aggiungi";
```

Metodi per il confronto tra stringhe

Metodo	Tipo parametri	Tipo restituito	Significato
compare(str)	string	intero	confronta la stringa con *str* e restituisce un numero <, = o > di 0;
compare(pos, n, str)	intero, intero, string	intero	confronta i primi *n* caratteri da *pos* con la stringa *str* e restituisce un numero <, = o > di 0;
compare(pos, ns, a, na)	intero, intero, char[], intero	intero	confronta i primi *n* caratteri da *pos* con *na* caratteri dell'array *a* e restituisce un numero <, = o > di 0;
empty()	nessuno	intero o bool	controlla se la stringa è vuota e restituisce *true* (1) o *false* (0).

Il confronto tra stringhe può avvenire anche con gli operatori ==, !=, <, >, <=, >= che la classe *string* sovraccarica.

Conversioni

Conversioni da stringhe ad array di caratteri

Metodo	Tipo parametri	Tipo restituito	Significato
c_str()	nessuno	char* (puntatore a char)	restituisce un array con i caratteri della stringa; può essere utilizzato in tutte le funzioni che gestiscono le stringhe come array di caratteri;

copy(a, n , pos)	char[], intero, intero	intero	copia *n* caratteri a partire da *pos* nell'array di caratteri *a* ritornando l'effettivo numero di caratteri copiati;

La classe *string* sovraccarica l'operatore **[]** per poter individuare il singolo carattere della stringa; perciò si può usare la sintassi degli array per scorrere la stringa o accedere ad un suo elemento. L'operatore [] diventa quindi un'alternativa al metodo *at()*.

Conversioni da interi ad array di caratteri a stringhe

Per convertire un intero in una stringa si può effettuare una doppia conversione (ed eventualmente un cast): prima si converte il numero in un array di caratteri, poi l'array in una stringa.

Le funzioni ***itoa()***, ***ltoa()*** definite nel file di libreria ***<cstdlib>*** permettono la conversione rispettivamente di un *int* e di un *long* in un array di caratteri.

Funzione	Tipo parametri	Tipo restituito	Significato
itoa(intero, a, valBase)	int, char[], intero	char* (puntatore a char)	inserisce nell'array *a* le cifre del numero *intero*; la conversione avviene secondo la base *valBase*; restituisce un puntatore alla stringa;
ltoa(lungo, a, valBase)	long, char[], intero	char* (puntatore a char)	inserisce nell'array *a* le cifre del numero *lungo*; la conversione avviene secondo la base *valBase*; restituisce un puntatore alla stringa;

```
char ac1[20], ac2[20];
int n = 100;
long gn = 1234567;
string s = "stringa: "+ (string)itoa(n, ac1, 10) + " e " +
(string)ltoa(gn, ac2, 10);
```
s contiene "stringa: 100 e 1234567"

Conversioni da numeri interi o reali a stringhe

Per convertire un numero reale o intero in una stringa della classe *string* è possibile utilizzare un oggetto della classe ***ostringstream***.

Nota

Per utilizzare la classe *ostringstream* bisogna includere il file di libreria ***<sstream>***.

La classe *ostringstream* sovraccarica l'operatore << e possiede il metodo ***str()*** che restituisce il contenuto dall'oggetto sotto forma di stringa.

```
ostringstream appoggio;
appoggio << realeOIntero;
string finale = appoggio.str();
```

Conversioni da stringhe a numeri interi o reali

Per convertire una stringa in un numero del tipo voluto è possibile utilizzare un oggetto della classe ***stringstream***. Tale classe sovraccarica gli operatori << e >> in modo da inserire una stringa ed estrarre il valore come tipo primitivo voluto.

Per utilizzare la classe *stringstream* bisogna includere il file di libreria **<sstream>**. **Nota**

```
stringstream appoggio;
tipo numero;
appoggio << stringa;
appoggio >> numero;
```

Questo schema funziona anche come convertitore numero-stringa in alternativa all'uso di *ostringstream*. **Nota**

Le classi della gerarchia stream sono trattate nel capitolo 16. **Nota**

Esempio 4.17

Elaborazione di stringhe.

```
#include <iostream>
using namespace std;

int main(void){
    string s, s1;
    s = "Non funziona";
    cout << s << endl;
    cout << s.find_first_of(' ',0) << endl;
    s1 = s.substr(4, 8);
    cout << s1 << endl;
    s = s.erase(0, 4);
    s = s.insert(s.size(), "!");
    cout << s << endl;
    getchar();
    return 0;
}//main
```

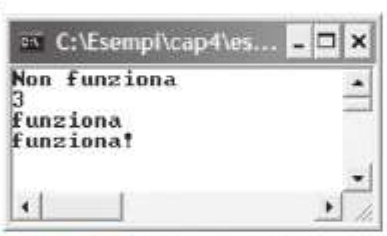


Figura 4.12 *Esecuzione dell'esempio*

Il fatto che non sia garantito che l'ordine in cui sono valutati gli operatori con la stessa precedenza sia sempre da sinistra a destra è particolarmente importante con le funzioni; per esempio in *a := f(x) + g(y)*, *g(y)* può essere eseguita prima di *f(x)*.

Se una espressione deve essere eseguita tassativamente prima di un'altra, bisogna spezzare l'espressione usando variabili per memorizzare i risultati intermedi:

```
temp := f(x);
a := temp + g(y);
```

C++ 4.10 La formattazione dei numeri in output

Con l'istruzione

```
cout << numero;
```

- se *numero* è **intero**: in output occupa tante posizioni quante sono le sue cifre;

- se *numero* è **reale** e la sua parte intera è composta da meno di 7 cifre: appare in forma decimale con lunghezza massima di 6 cifre;
- se *numero* è **reale** e la sua parte intera è composta da almeno 7 cifre: appare in **notazione esponenziale**:

segno	1 cifra per la parte intera	il . decimale	da 0 a 5 cifre decimali con l'ultima eventualmente arrotondata	e	segno dell'esponente	3 cifre di esponente

Il file di libreria ***<iomanip>*** contiene delle specifiche per poter formattare sia ciò che viene inviato al dispositivo *cout* sia ciò che viene prelevato dal dispositivo *cin*.

I manipolatori principali, oltre al già visto *endl*, sono quelli che permettono la stampa di un numero secondo i sistemi ottale ed esadecimale e l'impostazione della lunghezza di stampa.

PRINCIPALI MANIPOLATORI

Manipolatore	Dispositivo	Significato
oct	cin, cout	imposta il sistema ottale (base 8);
dec	cin, cout	imposta il sistema decimale (base 10);
hex	cin, cout	imposta il sistema esadecimale (base 16);
ends	cout	identifica la sequenza di escape *\0*, fine della stringa;
setw(n)	cout	lunghezza di stampa pari a *n*;
setprecision(nd)	cout	imposta il numero di cifre (compresi i decimali) pari a *nd* a cui sarà arrotondato il numero stampato;
setfill(car)	cout	imposta il carattere car come carattere di riempimento; lo spazio è il carattere di riempimento di default.

Nota Una volta impostato il manipolatore, questo rimane attivo per tutto il codice. Fa eccezione il manipolatore *setw* che agisce solo sull'istruzione di stampa successiva.

Sempre in *<iomanip>* sono definite le funzioni ***setiosflags*** e ***resetiosflags*** che permettono rispettivamente l'impostazione o la rimozione di particolari formattazioni attraverso il valore del loro parametro.
Il parametro è una delle costanti della classe ***ios*** (classe per gestire l'input/output) perciò (come si vedrà in seguito) sarà nella forma ***ios::nomeCostante***.

PRINCIPALI COSTANTI DI IOS

Costante di ios	Significato
`ios::fixed`	imposta i numeri reali (virgola mobile) con un numero di decimali definito da *setprecision*;
`ios::showbase`	stampa prima i caratteri 0 e 0x per i numeri stampati rispettivamente con *oct* (ottali) e *hex* (esadecimali);
`ios::scientific`	stampa i numeri reali in formato scientifico;
`ios::showpos`	stampa i numeri positivi preceduti dal carattere +;
`ios::showpoint`	stampa i numeri reali con il . decimale e cifre decimali (0) anche se sono valori interi;
`ios::uppercase`	stampa in maiuscolo le cifre esadecimali letterali e il carattere (e) del formato scientifico;

`ios::right`	impostazione di default; nell'ambito della dimensione di stampa impostata con *setw*, allinea a destra, riempendo lo spazio rimanente con il carattere impostato con *setfill*;
`ios::left`	nell'ambito della dimensione di stampa impostata con *setw*, allinea a sinistra, riempendo lo spazio rimanente con il carattere impostato con *setfill*.

Nota

Una volta impostato il manipolatore con la funzione *setiosflags(valoreCostante)*, questo rimane attivo per tutto il codice seguente finché non è richiamata la funzione *resetiosflags(valoreCostante)*.

Nota

È possibile invocare le funzioni *setiosflags* e *resetiosflags* indicando come parametro più costanti unite dall'operatore | (or bit a bit).

ESEMPIO 4.18

Esempi di formattazione mediante manipolatori di *iomanip*.

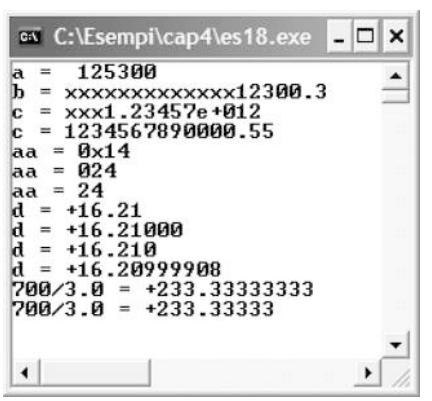


Figura 4.13 *Esecuzione dell'esempio*

```
#include <iostream>
#include <iomanip>
using namespace std;

int main(void){
    int a = 125300, aa = 20;
    float b = 12300.2691, d = 16.21;
    double c = 1234567890000.55;
    cout << "a = " << setw(7) << a << endl;
    cout << "b = " << setw(20) << setfill('x') << b << endl;
    cout << "c = " << setw(15) << c << endl;
    cout << "c = " << setprecision(15) << setfill(' ')
         << c << endl;
    cout << "aa = " << setiosflags(ios::showbase | ios::showpos)
         << hex << aa << endl;
    cout << "aa = "  << oct << aa
         << resetiosflags(ios::showbase) << endl;
    cout << "aa = "  << oct << aa << endl;
    cout << "d = " << setprecision(7) << d << endl;
    cout << "d = " << setiosflags(ios::showpoint) << d << endl;
    cout << "d = " << setprecision(5) << d
         << resetiosflags(ios::showpoint) << endl;
    cout << "d = " << setprecision(8)
         << setiosflags(ios::fixed) << d << endl;
    cout << "700/3.0 = " << 700/3.0 << endl;
    cout << "700/3.0 = " << resetiosflags(ios::fixed)
         << 700/3.0 << endl;
    getchar();
    return 0;
}//main
```

La funzione printf

Si può controllare il formato di stampa precisando quante posizioni deve occupare ciascun dato anche mediante la funzione ***printf()*** contenuta nel file di libreria ***<stdio.h>***.

```
printf(stringaDiControllo, variabili)
```

stringaDiControllo è una stringa contenente anche dei caratteri di controllo specificanti i tipi delle variabili da stampare ed eventualmente lo spazio occupato da esse.

Un carattere di controllo è formato dal simbolo **%** seguito da uno dei seguenti caratteri:

Carattere	Tipo applicabile	Esito
c	char	carattere (secondo il codice ASCII)
s	char[]	stringa (come array di caratteri)
d, i	int	intero con segno
u	int	intero senza segno
o	int	intero senza segno in forma ottale
x, X	int	intero senza segno in forma esadecimale
f	float, double	numero reale sempre con 6 cifre decimali
e, E	float, double	numero reale in notazione esponenziale
g, G	float, double	numero reale in notazione esponenziale o decimale a seconda della composizione dello stesso

Si possono riservare *n* posizioni per la stampa del numero scrivendo il numero *n* tra il simbolo % e il relativo specificatore; il numero viene allineato a destra, preceduto da spazi per completare le posizioni in eccesso; se il numero di posizioni richieste è inferiore al necessario, il valore viene comunque stampato in modo completo utilizzando ulteriori posizioni.

Per ottenere la stampa di un numero **reale** in **notazione decimale** con *m* cifre decimali basta scrivere *.m*. È possibile specificare il numero complessivo *n* di posizioni da riservare alla stampa del valore, compreso il segno, il punto decimale e le *m* cifre decimali, scrivendo *n.m*; se sono indicate meno posizioni di quelle necessarie, vengono occupate ulteriori posizioni in modo che non vengano perse cifre significative della parte intera; se le cifre decimali del valore da stampare sono più delle posizioni indicate, il numero viene arrotondato per difetto o per eccesso in base al valore della prima cifra aggiuntiva (se è minore di 5 il numero viene troncato, altrimenti arrotondato alla cifra superiore).

%7d riserva 7 posizioni, %15.3f riserva in totale 15 posizioni di cui 3 per i decimali (una posizione è sempre occupata dal punto decimale.

Nota

Per usare la funzione *printf()* non è necessario includere il file *<stdio.h>* se è già stato incluso *<iostream>*, poiché *<iostream>* include implicitamente anche *<stdio.h>*.

Esempio 4.19

Esempi di scrittura di dati mediante la funzione *printf()*.

```
#include <iostream>
using namespace std;

int main(void){
    int i = 3977;
    float rf = 457.7893;
    char car = 'D';
    printf("intero\n\tdecimale: %5i\n", i);
```

Figura 4.14 *Esecuzione dell'esempio*

```
    printf("\tesadecimale: %X\n\tottale: %o\n", i, i);
    printf("reale\n\tdecimale: %10.5f\n", rf);
    printf("\tscientifica: %10e\n", rf);
    printf("carattere\n\tvalore: %c\n\tASCII: %i\n", car, car);
    getchar();
    return 0;
}//main
```

LA FUNZIONE SCANF

Per acquisire valori dallo standard input è possibile utilizzare anche la funzione ***scanf()*** contenuta nel file di libreria ***<stdio.h>***.

```
scanf(stringaDiControllo, &variabili)
```

stringaDiControllo è una stringa contenente solo dei caratteri di controllo specificanti i tipi delle variabili da acquisire; i caratteri di controllo ammessi sono %d, %i, %f, %c, %s, %u il cui significato è lo stesso che per la funzione *printf()*.

Il significato del simbolo & (chiamato operatore di indirizzo) è spiegato nel paragrafo C++ 6.3.

Nota

Se con un'unica istruzione *scanf()* si leggono più valori, questi vanno separati da uno spazio o da un invio.

ESEMPIO 4.20

Esempi di lettura di dati mediante la funzione *scanf()*.

```
#include <iostream>
using namespace std;

int main(void){
    int ics, ics2;
    unsigned int iss;
    float rf;
    char car;
   printf("Inserisci un intero, un intero positivo e un
reale:\n");
    scanf("%i%u%f", &ics, &iss, &rf);
    printf("Inserisci un carattere:\n");
    fflush(stdin);
    scanf("%c", &car);
    printf("I dati sono:\n%5i\t\%5i\t%5u\n" , ics, ics2, iss);
    printf("%f\t%c\n", rf, car);
    ...
}//main
```

Figura 4.15 *Esecuzione dell'esempio*

C++ 4.11

DICHIARAZIONE DI NUOVI TIPI SEMPLICI

DICHIARAZIONE DI TIPO

In C++ si possono definire nuovi tipi di dati a partire dai tipi predefiniti.

La sezione di dichiarazione di nuovi tipi di dati comincia con la parola riservata ***typedef***.

La definizione di un nuovo tipo ha il formato:

```
typedef definizioneDiTipo nuovoTipo;
```

Il nome del nuovo tipo è preceduto dalla definizione del tipo e termina con un punto e virgola.

Dopo aver definito un nuovo tipo di dati si possono dichiarare variabili di quel tipo:

```
nuovoTipo nomeVariabile;
```

```
typedef unsigned short dueByte;
dueByte numero = 3000;
```
definizione del tipo *dueByte* e successiva dichiarazione (e assegnazione) di una variabile di quel tipo.

Tipi enumerativi

I tipi **enumerativi** permettono di definire tipi che richiedono un numero limitato di valori identificati da opportune costanti mnemoniche.
Nella definizione di tipo le costanti mnemoniche vanno elencate tra parentesi graffe e costituiscono un insieme ordinato di valori interi. Il numero di valori è detto **cardinalità** del tipo.

Per creare tipi enumerativi in C++ si usa l'istruzione ***enum***:

```
enum nomeTipo { elencoCostantiMnemoniche }
```

Le costanti assumono implicitamente i valori interi 0,1, ..., *cardinalità-1*. È possibile specificare valori diversi.

```
enum coloreSemaforo {rosso, giallo, verde};
```
crea una enumerazione di nome *coloreSemaforo* le cui componenti valgono 0 (*rosso*), 1 (*giallo*), 2 (*verde*).

```
enum coloreSemC {rosso=5, giallo, verde, lampeggiante=10, spento};
```
crea una enumerazione di nome *coloreSemC* le cui componenti valgono 5 (*rosso*), 6 (*giallo*), 7 (*verde*), 10 (*lampeggiante*), 11 (*spento*).

È possibile dichiarare variabili enumerative e assegnare loro valori:

```
coloreSemC semaforo1, semaforo2;

semaforo1 = rosso;
semaforo2 = spento;
```

Sulle variabili di tipo enumerativo **non** è possibile fare operazioni di **input** attraverso il dispositivo ***cin***.
L'output, attraverso il dispositivo *cout* è invece possibile e produce il valore intero corrispondente alla costante mnemonica.

Nota

Per la lettura di valori di tipo enumerativo si può ricorrere a istruzioni *if* e all'istruzione *switch*, spiegate nel capitolo 5.

Nota

Le operazioni di input/output di variabili enumerative attraverso i dispositivi *cin* e *cout* possono essere possibili sovraccaricando gli operatori << e >>.

Nota

In passato era necessario usare ***typedef*** affinché il nome dell'enumerazione creata con *enum* fosse un effettivo tipo con cui dichiarare variabili.

```
typedef enum coloreSemaforo {rosso, giallo, verde};
coloreSemaforo semaforo1;
```

creazione di un'enumerazione e dichiarazione di una variabile enumerativa.

In alternativa:

```
enum coloreSemaforo {rosso, giallo, verde};
enum coloreSemaforo semaforo1;
```

ESEMPIO 4.21

Esempi di enumerazioni.

```
#include <iostream>
using namespace std;

enum coloreSemC {rosso=5, giallo, verde, lampeggiante=10, spento};

int main(void){
    coloreSemC semaforo1, semaforo2, semaforo3;
    semaforo1 = rosso;
    semaforo2 = spento;
    semaforo3 = (coloreSemC) 5; //conversione di tipo necessaria
    cout << "semaforo1=" << semaforo1 << endl;
    cout << "semaforo2=" << semaforo2 << endl;
    cout << "semaforo3="
         << semaforo3 << endl;
    getchar();
    return 0;
}//main
```

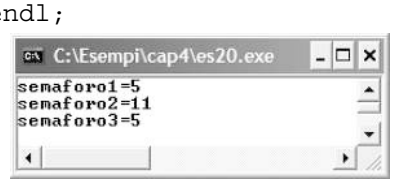


Figura 4.16 *Esecuzione dell'esempio*

VERIFICA

QUESTIONARIO

1. A che cosa servono le istruzioni di output?
2. A che cosa servono le istruzioni di input?
3. Da che cosa è caratterizzata una variabile?
4. Che cosa identifica un tipo di dati?
5. Come può avvenire una conversione di tipo?

TEST

1 ..... *Individuare la risposta esatta.*

1) Quale istruzione permette di ottenere un valore casuale rappresentante il risultato dell'estrazione di un numero della tombola (da 1 e 90)?

- ☐ `rand(90);`
- ☐ `rand() % 90 + 1;`

- ☐ `rand() % 90;`
- ☐ `rand() * 90 + 1;`
- ☐ `rand(90) + 1;`

2) Dato il seguente frammento di programma:

```
string u, t, v;
float r;
int x, y;
x = static_cast<int>(floor(r/4));
y = static_cast<int>(ceil(r/3));
v = t;
t = t.erase(x, y);
u = v.substr(x, y);
v = t.insert(y, u);
cout << v << endl;
```

Cosa contiene v se r è 12.13 e t contiene *ABCDefghIJKL*?

- ☐ ABCDefghIJKL
- ☐ ABCIJDefghKL
- ☐ ABCDIJKLefgh
- ☐ ABCDghefghIJKL
- ☐ una stringa diversa da tutte le precedenti.

3) Dato il seguente frammento di programma:

```
int x = 1, y = 3;
string r, s, t, u, v, w;
t = "ABcdEFgh";
u = t.substr(x, x+y);
v = t.substr(x+y, y);
r = t.substr(x, y-x);
s = t.substr(y-x, y);
w = u.append(v).append(r).append(s);
cout << w << endl;
```

Cosa stampa il programma se *t* contiene *ABcdEFgh*?

- ☐ `ABcdEFFgABch`
- ☐ `BcdEEFgBccdE`
- ☐ `BcdEEFghBcdE`
- ☐ `ABcdEFgBccdE`
- ☐ una stringa diversa da tutte le precedenti.

2 ..... Se *n* e *m* sono variabili intere e *x* e *y* sono variabili reali, indicare quali di queste istruzioni producono un errore di compilazione oppure un avvertimento (warning):

- ☐ `n = m + x;`
- ☐ `n = m * x;`
- ☐ `n = m / 2;`
- ☐ `y = m + x;`
- ☐ `y = m * x;`
- ☐ `y = m / 2;`
- ☐ `y = m / 2.0;`
- ☐ `y = x / 2;`
- ☐ `m = n % y;`
- ☐ `y = m % n;`
- ☐ `x = n % m;`
- ☐ `y = x % m;`
- ☐ `n = x % y;`

Esercizi

1 ..... Dire che cosa succede eseguendo le seguenti istruzioni:

```
cout << (48/7) << endl;
cout << (4.8/7) << endl;
cout << (48/7.0) << endl;
cout << (48.0/7.0) << endl;
```

2 ..... Se *x* e *y* sono due variabili reali dire che cosa stampano le istruzioni:

```
cout << floor(x)*floor(y) << " : " << floor(x*y) << endl;
cout << ceil(x)*ceil(y) << " : " << ceil(x*y) << endl;
```

dopo ciascuna coppia di assegnazioni:

```
x := 3.44;  y := 2.18;
x := 3.44;  y := 2.79;
x := 3.84;  y := 2.18;
x := 3.84;  y := 2.79;
```

3 ..... Dire che cosa viene stampato eseguendo le seguenti istruzioni:

```
float f = 12.30;
cout << f << " euro" << endl;
cout << setprecision(2) << f << " euro" << endl;
cout << setiosflags(ios::fixed)
     << setprecision(2) << f << " euro" << endl;
```

4 ..... Definire un tipo enumerativo per:

- i nomi dei giorni della settimana;
- le stagioni;
- i mezzi di trasporto.

PROPOSTE DI LAVORO

1 ..... Chiedere all'utente di inserire il proprio nome e la data e stampare il messaggio:

```
Ciao ...
Oggi è il ...
Come stai?
```

2 ..... Chiedere all'utente di inserire titolo e autore di un libro e stampare:

```
Il mio libro preferito è "...." di .....
```

3 ..... Chiedere all'utente di inserire i nomi di tre colori e stamparli uno dopo l'altro; poi stampare:

```
I miei colori preferiti sono: .., ... e ...
```

4 ..... Stampare il seguente prospetto, dopo aver chiesto all'utente i dati necessari:

```
Cognome e nome ...
Nato a ... il ...
Residente a ... prov. (...) CAP ... in via... n. ...
```

5 ..... Stampare il seguente prospetto, dopo aver chiesto all'utente i dati necessari:

```
Nome ... età ... sesso ...
sport preferito ...
hobby preferito ...
```

6 Chiedere all'utente di inserire il testo di una domanda e di tre possibili risposte, di cui una sola esatta, e di specificare il numero della risposta esatta. Stampare il seguente prospetto:

```
(testo della domanda)
1 - (testo della prima risposta)
2 - (testo della seconda risposta)
3 - (testo della terza risposta)
La risposta esatta è la ...
```

7 Chiedere all'utente di inserire due numeri, calcolarne la somma e il prodotto e stampare:

```
... + ... = ...
... x ... = ...
```

8 Date le misure dei cateti di un triangolo rettangolo, calcolare l'area e il perimetro.

9 Dati altezza e peso di tre persone, calcolare altezza e peso medi.

10 Calcolare la percentuale all'*n*% di un numero reale.

11 Di un prodotto viene fornito il prezzo all'inizio e alla fine dell'anno; calcolare l'incremento o decremento in valore e in percentuale.

12 Dati i risultati di un referendum: numero iscritti, numero sì, numero no, numero schede bianche o nulle, calcolare la percentuale dei votanti sul numero degli iscritti e le percentuali dei sì e dei no sul numero degli iscritti e sul numero dei votanti.

13 Calcolare il valore di un polinomio di secondo grado, dopo aver chiesto i valori dei coefficienti di secondo e primo grado e del termine noto e il valore *x* per cui si desidera il calcolo. Stampare

```
... X*X + ... X + ... = ...
```

14 Data la misura di un angolo espressa in gradi, primi e secondi, trasformarla in secondi.

15 Data la misura di un angolo espressa in secondi, trasformarla in gradi, primi e secondi.

16 Dato un numero intero minore di 1000, dire di quante centinaia, decine e unità è composto.

17 Avendo a disposizione banconote da 100, 50, 20, 10 e 5 euro, determinare il minor numero di banconote per formare un valore *n* espresso in migliaia.

18 Date tre variabili *a*, *b*, *c*, ruotarne il valore (*a* → *b* → *c* → *a*).

19 Dati due numeri, calcolare la radice quadrata del valore assoluto della differenza.

20 Arrotondare un numero positivo *n* all'*m*-esima cifra decimale.
La formula per l'arrotondamento è $parteInteraDi(n*10^m + 0{,}5)/10^m$.

21 Generare un numero casuale compreso tra 15 e 25.

22 Data una stringa, calcolarne la lunghezza, fare una copia del primo carattere e inserirlo in fondo, poi cancellare il primo carattere. Stampare la stringa risultante.

23 Data la data di nascita di una persona (espressa come una stringa formata da anno di 4 cifre, mese e giorno) e l'anno corrente (espresso come una stringa di 4 cifre) calcolare l'età della persona.

24 Chiedere all'utente nome ed età e creare una stringa che funga da password concatenando le prime due lettere del nome, la prima cifra dell'età, un numero casuale di 2 cifre (quindi tra 10 e 99), l'ultima cifra dell'età e le ultime due lettere del nome in maiuscolo.

5 Strutture di controllo

Prerequisiti

Saper descrivere algoritmi e realizzare programmi in C++ che usano:

- istruzioni di input/output
- variabili, costanti e tipi di dati
- assegnazioni, espressioni ed operatori

Obiettivi

Conoscenze

- Struttura condizionale
- Operatori logici ed espressioni booleane
- Strutture cicliche
- Controllo degli errori

Abilità/capacità

- Descrivere algoritmi che utilizzano le strutture di controllo
- Scrivere programmi in C++ che usano istruzioni condizionali e cicliche
- Eseguire il debug di programmi

5.1 LE STRUTTURE DI CONTROLLO

Per descrivere qualsiasi algoritmo basta usare una combinazione delle tre strutture di controllo, chiamate **schemi di composizione fondamentali**:

- la **struttura sequenziale** in cui le istruzioni vengono indicate una dopo l'altra nell'ordine in cui devono essere eseguite;
- la **struttura di selezione o condizionale** in cui le istruzioni vengono separate in due gruppi che vengono eseguiti in alternativa in base al verificarsi di una condizione;
- la **struttura iterativa o ciclica** in cui viene indicato un gruppo di istruzioni da ripetere più volte, terminando in base al verificarsi di una condizione.

Ogni gruppo di istruzioni presente in una struttura condizionale o iterativa può a sua volta comprendere istruzioni di qualsiasi tipo, comprese altre strutture condizionali o cicliche; si dice in questo caso che le strutture sono **nidificate** (cioè una interna all'altra).

L'INDENTAZIONE

L'**indentazione** è un metodo di scrittura, utilizzato sia per la pseudocodifica che per la scrittura dei programmi in linguaggio di programmazione, che permette di **evidenziare** gli schemi di composizione utilizzati per la stesura dell'algoritmo, e quindi di far comprendere più facilmente qual è il flusso di esecuzione dell'algoritmo stesso.
Il metodo consiste nello scrivere le istruzioni dei cicli, o dei rami delle strutture condizionali, spostandosi verso destra di alcune colonne ad ogni livello di nidificazione.
Molti editor offrono funzioni che facilitano la scrittura dei programmi con il metodo dell'indentazione; infatti il ritorno a capo può essere fatto posizionandosi sotto il primo carattere della riga precedente anziché sul primo carattere della nuova riga.

Nota Tutti gli esempi proposti verranno scritti indentando opportunamente le istruzioni.

5.2 LA STRUTTURA SEQUENZIALE

La **struttura sequenziale** è la prima delle strutture di controllo utilizzabili per la descrizione dell'algoritmo e consiste semplicemente nella concatenazione di istruzioni.

Le istruzioni vengono eseguite una dopo l'altra nell'**ordine** in cui si presentano, come si è visto nel capitolo precedente.

```
istruzione1
...
istruzioneN
```

L'ISTRUZIONE COMPOSTA

C++ 5.1

Un'**istruzione composta** è formata da una serie di istruzioni racchiuse tra **parentesi graffe** (i simboli **{** e **}**).
All'interno delle parentesi graffe ogni istruzione termina con il **punto e virgola**.
Le istruzioni di un'istruzione composta sono eseguite **sequenzialmente**.

Le istruzioni composte sono trattate come una **singola istruzione** e sono usate in tutte le istruzioni C++ che come sintassi richiedono una sola istruzione.

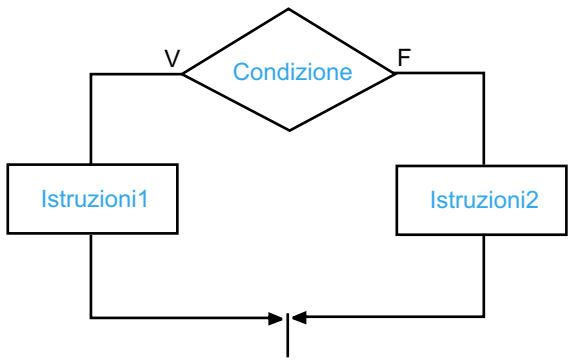
5.3 LA STRUTTURA CONDIZIONALE

La **struttura condizionale** permette di eseguire due diverse sequenze di istruzioni, **in alternativa**, secondo il valore di una determinata **condizione**.

Le due sequenze di istruzioni in alternativa compongono i due **rami** della struttura condizionale.

Lo schema generale della struttura condizionale è:

se condizione **allora**
    istruzioni1
altrimenti
    istruzioni2
fine se



La ***condizione*** (o *espressione condizionale*) è una **proposizione logica** che dà un risultato booleano, cioè che può assumere uno dei valori di verità *vero* o *falso*; è una espressione che può contenere variabili e costanti, operatori relazionali ed operatori logici.

La condizione può essere costituita per esempio dal confronto tra due variabili numeriche, o tra una variabile e una costante, per determinare se sono uguali o diverse, o se una è maggiore dell'altra, o tra due stringhe, per determinare se una precede l'altra nell'ordine alfabetico ecc.

Se la proposizione logica che costituisce la condizione ha valore ***vero***, vengono eseguite le istruzioni che seguono la parola ***allora***; se ha valore ***falso***, vengono eseguite le istruzioni che seguono la parola ***altrimenti***.

In generale i rami delle strutture condizionali possono contenere una sequenza di istruzioni (di qualsiasi tipo).

OPERATORI RELAZIONALI

Gli **operatori relazionali** permettono di confrontare due valori:

$=$ $\neq$ $>$ $<$ $\geq$ $\leq$

Oltre che tra valori numerici gli operatori relazionali si possono utilizzare anche tra valori di tipo carattere; in tal caso l'esito del confronto si basa sulla posizione dei caratteri nella tabella del codice usato per la rappresentazione (ASCII o Unicode, in base al linguaggio); un carattere è minore dell'altro se lo precede nella tabella del codice e viceversa.

Il **confronto** tra variabili **alfanumeriche** nei linguaggi di programmazione avviene in base al codice corrispondente a ciascun carattere.

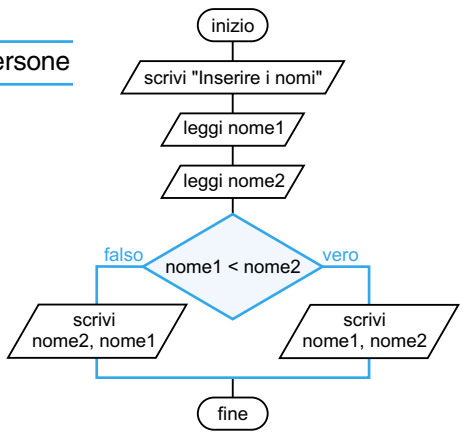
Esempio 5.1

Scrivere in ordine alfabetico i nomi di due persone.

Per nome si intende anche cognome e nome separati da spazi.
Per determinare l'ordine alfabetico viene usato l'operatore < che in questo caso significa "precede nell'ordine alfabetico".

I/O	nome1, nome2	stringhe	nomi delle due persone

```
inizio
    scrivi "Inserire i nomi"
    leggi nome1
    leggi nome2
    se nome1 < nome2 allora
        scrivi nome1, nome2
    altrimenti
        scrivi nome2, nome1
    fine se
fine
```



Struttura condizionale con un solo ramo

La struttura condizionale può essere costituita da **un solo ramo** se per uno dei due valori della condizione non deve essere eseguita nessuna istruzione.

se condizione **allora**
  istruzioni1
fine se
	oppure	**se non** condizione **allora**
		  istruzioni2
		fine se

L'istruzione if .. else

C++ 5.2

La struttura condizionale in C++ si codifica con l'istruzione ***if***.

Se in ogni ramo c'è una sola istruzione il formato è

```
if(condizione)
    istruzione;
else
    istruzione;
```

Nota — Il punto e virgola non deve mai essere inserito dopo la condizione o dopo la parola *else*.

Se la struttura condizionale è formata da **un solo ramo** si può omettere l'*else:*

```
if(condizione)
    istruzione;
```

L'istruzione in ogni ramo può essere un'istruzione composta.

```
if(condizione) {
    istruzione1;
    ...
    istruzioneN;
}//if
else {
    istruzione1;
    ...
    istruzioneN;
}//else
```

La parentesi graffa aperta indicante l'inizio dell'istruzione composta può essere scritta anche in una nuova riga; la parentesi graffa chiusa può essere scritta anche di seguito al punto e virgola dell'ultima istruzione.

Nota

Si suggerisce di usare questa convenzione per avere listati più brevi e più leggibili e di inserire un opportuno commento dopo ogni parentesi graffa chiusa per evidenziare quale istruzione composta termina.

La **condizione** è un'espressione che restituisce un valore booleano.

La valutazione dell'espressione booleana si ferma quando il valore è noto con certezza (**short-cut evaluation**).

OPERATORI RELAZIONALI IN C++

==	uguale	>	maggiore	>=	maggiore o uguale
!=	diverso	<	minore	<=	minore o uguale

Il confronto tra caratteri e stringhe è fatto in base al codice ASCII.

L'operatore == può essere usato per controllare l'uguaglianza di oggetti solo se è stato sovraccaricato nella classe che definisce l'oggetto, altrimenti viene segnalato un errore di compilazione.

ESEMPIO 5.2

Codifica in C++ dell'esempio 5.1.

```
#include <iostream>
using namespace std;

int main(void){
    string nome1, nome2;
    cout << "Inserire i nomi\n";
    getline(cin, nome1);
    getline(cin, nome2);
    if(nome1 < nome2)
        cout << nome1 << " " << nome2;
    else
        cout << nome2 << " " << nome1;
    getchar();
    return 0;
}//main
```

Figura 5.1 *Esecuzione dell'esempio*

L'operatore condizionale ?:

?: operatore condizionale

L'operatore condizionale **?:** ha il significato *condizione ? valore per vero : valore per falso;*

```
max = k>j ? k : j;
```
assegna a *max* il maggiore tra *k* e *j*

Con l'operatore ?: quando è possibile si evita di proseguire la valutazione (**short-cut evaluation**).

Strutture condizionali in sequenza e nidificate

Le strutture condizionali possono essere combinate in vario modo tra loro o con altre strutture di controllo per la soluzione di un problema.

Se una struttura condizionale inizia dopo la conclusione della precedente si parla di strutture **in sequenza**.

Si hanno invece strutture condizionali **nidificate** se un ramo di una struttura condizionale contiene a sua volta un'altra struttura condizionale.
Possono esserci anche più livelli di nidificazione.

If nidificati

C++ 5.3

Usando gli ***if nidificati*** può esserci qualche ambiguità nel capire quale *else* corrisponde a quale *if*. L'***else*** si riferisce sempre, tornando indietro, al primo *if* a cui non corrisponde ancora nessun *else*.

Per esempio

```
if(cond1)
    if(cond2)
        istruzioni1;
else
    istruzioni2;
```

corrisponde a

```
if(cond1){
    if(cond2)
        istruzioni1;
    else
        istruzioni2;
}//if
```

e non, come si potrebbe pensare a

```
if(cond1){
    if(cond2)
        istruzioni1;
}//if
else
    istruzioni2;
```

Per ottenere quest'ultimo comportamento è **indispensabile** utilizzare le parentesi graffe.

Quando non si è sicuri è bene utilizzare le parentesi graffe, anche se non strettamente necessarie.

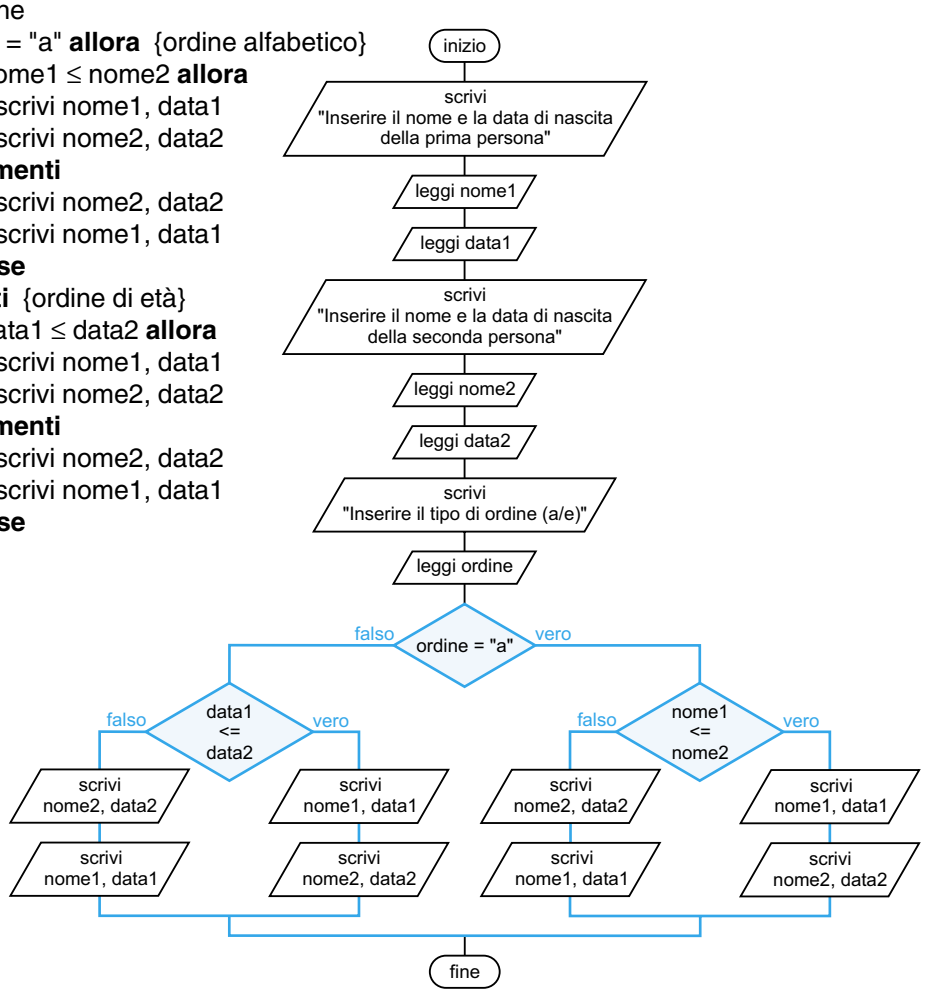
Esempio 5.3

Richiedere i nomi e le date di nascita di due persone; permettere poi, a richiesta, di ottenere i dati delle due persone in ordine alfabetico in base al nome, o in ordine di età.

Per fare direttamente il confronto tra le date, si devono inserire come stringhe nel formato *annoMeseGiorno*. L'anno può essere inserito con quattro cifre o con due cifre a scelta, ma nello stesso formato per le due date.

I/O	nome1, nome2	stringhe	nomi delle due persone
I/O	data1, data2	stringhe	date di nascita delle due persone
I	ordine	carattere	opzione per la scelta del tipo di ordinamento: "a" alfabetico, "e" per età

```
inizio
    scrivi "Inserire il nome e la data di nascita della prima persona"
    leggi nome1
    leggi data1
    scrivi "Inserire il nome e la data di nascita della seconda persona"
    leggi nome2
    leggi data2
    scrivi "Inserire il tipo di ordine (a/e)"
    leggi ordine
    se ordine = "a" allora  {ordine alfabetico}
        se nome1 ≤ nome2 allora
            scrivi nome1, data1
            scrivi nome2, data2
        altrimenti
            scrivi nome2, data2
            scrivi nome1, data1
        fine se
    altrimenti  {ordine di età}
        se data1 ≤ data2 allora
            scrivi nome1, data1
            scrivi nome2, data2
        altrimenti
            scrivi nome2, data2
            scrivi nome1, data1
        fine se
    fine se
fine
```



Per la scelta tra ordine alfabetico e ordine di età bisogna inserire uno dei caratteri "a" (per alfabetico) o "e" (per età). Ogni carattere diverso fa ottenere l'ordinamento in base all'età poiché la condizione *se ordine = "a" allora* risulta falsa e vengono eseguite le istruzioni del ramo falso.

Nota

```
#include <iostream>
using namespace std;

int main(void){
    string nome1, nome2, data1, data2;
    char ordine;
    cout << "Inserire il nome e la data di nascita della prima
persona\n";
    getline(cin, nome1);
    getline(cin, data1);
    cout << "Inserire il nome e la data di nascita della seconda
persona\n";
    getline(cin, nome2);
    getline(cin, data2);
    cout << "Inserire il tipo di ordine (a/e)" << endl;
    cin >> ordine;
    if(ordine == 'a')
        if(nome1 <= nome2){
            cout << nome1 << " " << data1 << endl;
            cout << nome2 << " " << data2 << endl;
        }//if
        else {
            cout << nome2 << " " << data2 << endl;
            cout << nome1 << " " << data1 << endl;
        }//else
    else
        if(data1 <= data2){
            cout << nome1 << " " << data1 << endl;
            cout << nome2 << " " << data2 << endl;
        }//if
        else {
            cout << nome2 << " " << data2 << endl;
            cout << nome1 << " " << data1 << endl;
        }//else
    fflush(stdin);
    getchar();
    return 0;
}//main
```

La variabile *ordine* è una variabile opzione.

Figura 5.2 *Esecuzione dell'esempio*

Le **opzioni** sono variabili usate per immettere scelte.

Casi limite

Nello stabilire il test della struttura condizionale bisogna fare particolarmente attenzione ai **casi limite**; per esempio nel confronto di due valori si distinguono i casi:

- primo valore maggiore del secondo;
- secondo valore maggiore del primo;
- valori uguali (caso limite).

Nell'esempio 5.3 se due valori (nomi o date) sono uguali, si vogliono stampare i dati nell'ordine in cui sono stati inseriti, quindi il test va fatto per minore o uguale e non solo per minore. Nell'esempio 5.1 invece se i nomi erano uguali era indifferente stampare prima il primo o il secondo.

Nota

Condizioni composte con operatori logici

I test delle strutture condizionali possono essere **condizioni composte** con **operatori logici**. La condizione composta viene valutata per stabilirne il valore di verità (*vero* o *falso*) in base ai valori delle condizioni semplici e agli operatori da applicare.

Operatori logici

Gli operatori logici più comuni sono ***and***, ***or*** e ***not***; il loro comportamento è definito da delle tabelle, dette **tavole di verità**, in cui ad ogni possibile coppia di valori (*vero* o *falso*) degli operandi viene associato il valore (*vero* o *falso*) del risultato delle operazioni.

Tabella 5.1 *Tabella di verità degli operatori booleani*

X	Y	XandY	XorY	notX
falso	falso	falso	falso	vero
falso	vero	falso	vero	
vero	falso	falso	vero	falso
vero	vero	vero	vero	

Gli operatori logici permettono di combinare espressioni relazionali.

Quando in una espressione compaiono operatori booleani bisogna controllare la precedenza degli operatori ed eventualmente correggerla con l'uso di parentesi.

Nota

C++ 5.4

Operatori logici

! (not) && (and) || (or)

L'operatore *and* ha la precedenza sull'operatore *or* e l'operatore *not* ha la precedenza su entrambi. Inoltre gli operatori logici hanno la **precedenza** sugli operatori relazionali.
Bisogna quindi precisare l'ordine di valutazione desiderato, inserendo opportunamente le parentesi tonde.

Gli operandi sono **booleani** e il risultato è ancora un valore booleano.

Attenzione a non confondere gli operatori &&, || e ! usati su operandi booleani con gli operatori **&**, **|** e **!** usati su operandi interi in grado di agire bit per bit.

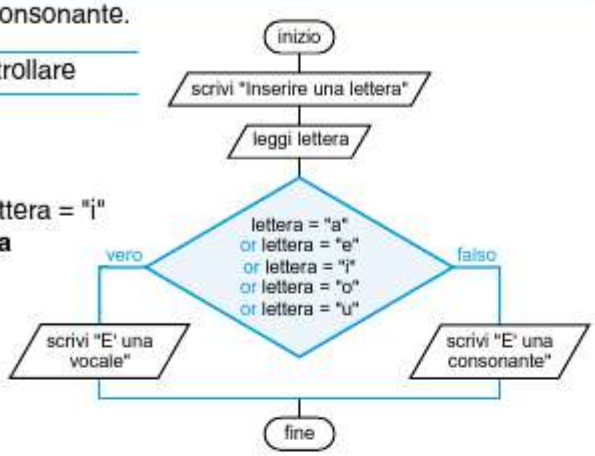
Nota

ESEMPIO 5.4

Dire se una lettera è una vocale o una consonante.

I	lettera	carattere	lettera da controllare

```
inizio
    scrivi "Inserire una lettera"
    leggi lettera
    se lettera = "a" or lettera = "e" or lettera = "i"
    or lettera = "o" or lettera = "u" allora
        scrivi "E' una vocale"
    altrimenti
        scrivi "E' una consonante"
    fine se
fine
```




```
#include <iostream>
using namespace std;

int main(void){
    char lettera;
    cout << "Inserire una lettera ";
    cin >> lettera;
    if((lettera == 'a') ||
    (lettera == 'e') || (lettera == 'i') ||
    (lettera == 'o') || (lettera == 'u'))
        cout << "E' una vocale" << endl;
    else
        cout << "E' una consonante " << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 5.3 *Esecuzione dell'esempio*

ALGEBRA BOOLEANA DELLA LOGICA PROPOSIZIONALE

Una **proposizione** è una frase che può essere ***vera*** o ***falsa***.
Le proposizioni elementari possono essere combinate tra di loro in proposizioni composte mediante gli **operatori logici**.

Per l'**algebra delle proposizioni** valgono le seguenti **proprietà**:
(*A* e *B* siano proposizioni aventi valori di verità *vero* o *falso*)

and e ***or*** sono operazioni **commutative**:
A and B = B and A
A or B = B or A

vero è un **elemento neutro** rispetto all'***and***, cioè non muta il valore di una proposizione; ***falso*** è l'elemento neutro rispetto all'***or***:
A and vero = A
A or falso = A

e **reciprocamente distributive**:
A or (B and C) = (A or B) and (A or C)
A and (B or C) = (A and B) or (A and C)

not è un'operazione **unaria** (ha un solo operando) detta di negazione o complemento: il complemento di *A* è *not(A)* ed è tale che:
A or not(A) = vero
A and not(A) = falso

Da queste proprietà si possono ricavare alcuni **teoremi**.

Teorema della proprietà di **associatività di *and* e or**
(A and B) and C = A and (B and C)
(A or B) or C = A or (B or C)

Teorema dell'**esistenza di un elemento annullatore**
A and falso = falso
A or vero = vero

Teorema della **proprietà di assorbimento**
A or (A and B) = A
A and (A or B) = A

Teorema della **proprietà di idempotenza**
A and A = A
A or A = A

Teorema dell'**involuzione**
not(not(A)) = A

Teorema delle **leggi di De Morgan**
not(A and B) = not(A) or not(B)
not(A or B) = not(A) and not(B)

I postulati e i teoremi dell'algebra booleana permettono di **trasformare** un'espressione booleana in un'altra forma **equivalente**.

Uso delle variabili booleane come espressioni condizionali

Al posto di una espressione condizionale si può usare una variabile booleana.

se variabileBooleana allora

equivale a

se variabileBooleana = vero allora

(il test di uguaglianza al valore vero è sottinteso);

per indicare

se variabileBooleana = falso allora

si scrive invece

se not variabileBooleana allora

Esempio 5.5

Dire se un numero è pari o dispari.

I	numero	intero	numero da controllare
	resto	intero	resto della divisione per 2
	pari	booleano	indica se il numero è pari o dispari

Impostazione della variabile *pari* in base al resto della divisione.

```
resto ← numero mod 2
se resto = 0 allora
    pari ← vero
altrimenti
    pari ← falso
fine se
```

oppure, in modo più sintetico, ricordando che un confronto dà come risultato un valore booleano:

```
resto ← numero mod 2
pari ← resto = 0
```

Istruzione condizionale in cui la condizione è costituita da una variabile booleana.

```
se pari allora
    scrivi 'E'' pari'
altrimenti
    scrivi 'E'' dispari'
fine se
```

Flag o switch

Si chiamano **flag** o **switch** (o anche deviatori) le variabili che permettono di impostare un valore (di solito tra due possibilità, variabili booleane) che segnala il verificarsi di una condizione.

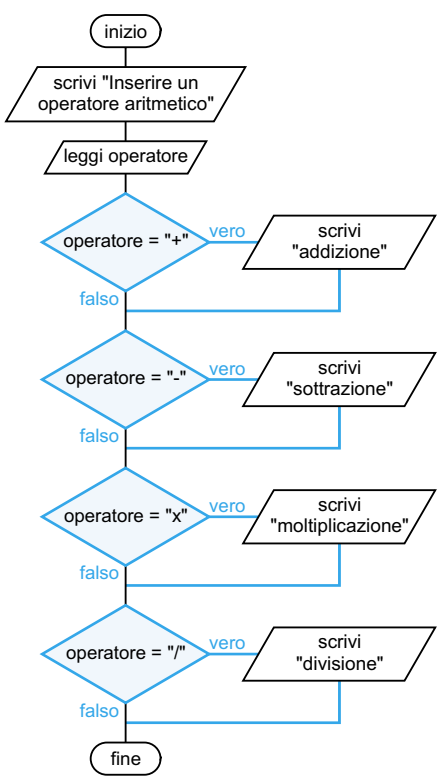
Strutture condizionali in sequenza, nidificate o condizioni composte

Quando per descrivere un algoritmo si possono usare strutture condizionali in sequenza e quando invece è necessario ricorrere a strutture condizionali nidificate o a condizioni composte?
Per poter usare soltanto strutture condizionali **in sequenza**, le condizioni devono far riferimento a **casi indipendenti** l'uno dall'altro.
Se ciò non si verifica si devono usare **strutture nidificate**, o si devono rendere indipendenti le condizioni, individuando le **condizioni composte** che risolvono il problema.

Esempio 5.6

Scrivere l'operazione dato l'operatore aritmetico.

```
I   operatore   carattere   operatore (+ - x /)
inizio
      scrivi "Inserire un operatore aritmetico"
      leggi operatore
      se operatore = "+" allora
          scrivi "addizione"
      fine se
      se operatore = "-" allora
          scrivi "sottrazione"
      fine se
      se operatore = "x" allora
          scrivi "moltiplicazione"
      fine se
      se operatore = "/" allora
          scrivi "divisione"
      fine se
 fine
```



Nell'esempio 5.6 tutte le condizioni sono indipendenti e quindi si possono usare strutture condizionali in sequenza.

Nota

Si possono usare strutture nidificate per maggior efficienza; con le strutture in sequenza infatti vengono eseguiti sempre tutti i controlli.

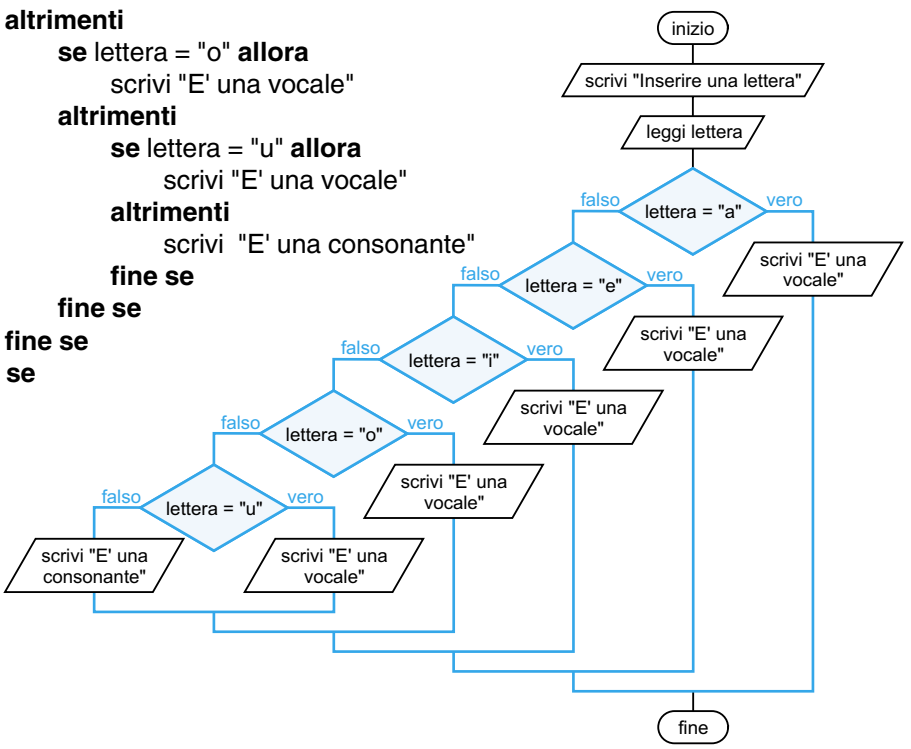
Se si volesse risolvere l'esempio 5.4 usando solo strutture condizionali in sequenza ne servirebbero 26, una per ogni lettera.

La soluzione vista che usa le condizioni composte è la più semplice. Non volendo usare condizioni composte si possono usare strutture nidificate, anche se la soluzione risulta un po' più complessa.

Esempio 5.7

Soluzione dell'esempio 5.4 non usando condizioni composte ma strutture condizionali nidificate.

```
I   lettera     carattere     lettera da controllare
inizio
    scrivi "Inserire una lettera"
    leggi lettera
    se lettera = "a" allora
        scrivi "E' una vocale"
    altrimenti
        se lettera = "e" allora
            scrivi "E' una vocale"
        altrimenti
            se lettera = "i" allora
                scrivi "E' una vocale"
            altrimenti
                se lettera = "o" allora
                    scrivi "E' una vocale"
                altrimenti
                    se lettera = "u" allora
                        scrivi "E' una vocale"
                    altrimenti
                        scrivi  "E' una consonante"
                    fine se
                fine se
            fine se
        fine se
    fine se
fine
```



La selezione multipla

La **selezione multipla** permette di scegliere fra più alternative.

Nota La selezione multipla può sempre essere sostituita da istruzioni *if*.

La selezione multipla permette di indicare quali istruzioni eseguire per diversi valori di una espressione chiamata **selettore**, specificando vari casi.

```
nel caso che espressione sia
    caso1: istruzione1
    ...
    casoN: istruzioneN
altrimenti
    istruzione
fine caso
```

Ogni caso individua una o più **costanti** o **intervalli** di valori. Il valore dell'espressione selettore viene confrontato di volta in volta con ciascuno dei valori o intervalli indicati nei vari casi; se il valore è compreso nel caso considerato, vengono eseguite le istruzioni relative; poi l'istruzione termina e l'esecuzione prosegue con la prima istruzione successiva.

Può esserci una parte opzionale che viene eseguita quando il valore dell'espressione selettore non rientra in nessuno dei casi indicati.

Esempio 5.8

Scrivere l'operazione dato l'operatore aritmetico.

```
I   operatore    carattere    operatore (+ - x /)
inizio
    scrivi "Inserire un operatore aritmetico "
    leggi operatore
    nel caso che operatore sia
        "+": scrivi "addizione"
        "-": scrivi "sottrazione"
        "x": scrivi "moltiplicazione"
        "/": scrivi "divisione"
    fine caso
fine
```

Nota Il flowchart è uguale a quello dell'esempio 5.6.

Esempio 5.9

Dire se una lettera è una vocale o una consonante.

```
I   lettera    carattere    lettera da controllare
inizio
    scrivi "Inserire una lettera"
    leggi lettera
    nel caso che lettera sia
        "a","e","i","o","u": scrivi "E' una vocale"
    altrimenti
        scrivi "E' una consonante"
    fine caso
fine
```

L'istruzione switch

C++

5.5

L'istruzione ***switch*** termina sempre con } (parentesi graffa chiusa).

Ogni caso consiste di una costante e termina con l'istruzione ***break*** per evitare che vengano eseguiti i casi successivi e/o il caso di default.

```
switch(espressione){
case caso1:
    istruzioni1;
    break;
    ...
case casoN:
    istruzioniN;
    break;
default:
    istruzioni;
}//switch
```

Se a casi distinti si vuole far eseguire le stesse istruzioni basta scrivere in sequenza i casi (sempre nella forma *case valoreCaso:*) e nell'ultimo le istruzioni volute.

La variabile **selettore** può essere di tipo intero, carattere, booleano ma anche un tipo enumerativo definito dall'utente.

Esempio 5.10

Scrivere l'operazione dato l'operatore aritmetico.

```
#include <iostream>
using namespace std;

int main(void){
    char operatore;
    cout << "Inserire un operatore aritmetico ";
    cin >> operatore;
    switch(operatore){
    case '+':
        cout << "addizione" << endl;
        break;
    case '-':
        cout << "sottrazione" << endl;
        break;
    case 'x':
        cout << "moltiplicazione" << endl;
        break;
    case '/':
        cout << "divisione" << endl;
    }//switch
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Esempio 5.11

Dire se una lettera è una vocale o una consonante.

```
#include <iostream>
using namespace std;

int main(void){
```

```
    char lettera;
    cout << "Inserire una lettera ";
    cin >> lettera;
    switch(lettera){
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        cout << "E' una vocale" << endl;
        break;
    default:
        cout << "E' una consonante " << endl;
    }//switch
    fflush(stdin);
    getchar();
    return 0;
}//main
```

ESEMPIO 5.12

Stampare il valore di un semaforo definito con un tipo enumerativo.

```
#include <iostream>
using namespace std;

enum coloreSemC {rosso=5, giallo, verde, lampeggiante=10, spento};

int main(void){
    coloreSemC semaforo = verde;
    switch(semaforo){
    case rosso:
        cout << "Il semaforo e' rosso\n";
        break;
    case giallo:
        cout << "Il semaforo e' giallo\n";
        break;
    case verde:
        cout << "Il semaforo e' verde\n";
        break;
    default:
        cout << "Il semaforo e' lampeggiante o spento\n";
    }//switch
    getchar();
    return 0;
}//main
```

TRACE CON STRUTTURE CONDIZIONALI

Il comportamento errato di un programma con strutture condizionali può essere dovuto all'esecuzione di un ramo non corretto di una delle strutture.

Con la **tabella di traccia** si può controllare il valore delle espressioni condizionali.

Tabella di traccia dell'esempio 5.3:

	nome1	nome2	data1	data2	ordine
leggi nome1	tommaso				
leggi data1			19910511		
leggi nome2		rachele			
leggi data2				20021027	
leggi ordine					a

Nel programma può essere utile stampare prima dell'inizio della struttura condizionale il valore delle variabili che intervengono nella **condizione**, in modo da valutare la condizione che si verifica, o porre in ogni ramo della struttura condizionale un messaggio che identifichi il ramo stesso (e quindi il tipo di condizione verificata).

```
cin >> ordine;
cout << "ordine = " << ordine << endl;
cout << "nome1 = " << nome1 << endl;
cout << "nome2 = " << nome2 << endl;
cout << "data1 = " << data1 << endl;
cout << "data2 = " << data2 << endl;
if(ordine == 'a')
   if(nome1 <= nome2){
```

5.4 STRUTTURE CICLICHE

Spesso è necessario ripetere più volte uno stesso gruppo di istruzioni; le istruzioni di controllo che permettono di controllare la ripetizione vengono dette **strutture cicliche o ripetitive**.

Le **strutture cicliche** permettono di **ripetere** una sequenza di operazioni, terminando quando si verifica una particolare **condizione**.

Le istruzioni cicliche possono essere usate in molti modi diversi.

La realizzazione di algoritmi ripetitivi

Per realizzare un **algoritmo ciclico** si devono determinare:

- le inizializzazioni da effettuare prima del ciclo;
- le operazioni che devono essere ripetute all'interno del ciclo;
- la condizione di terminazione del ciclo;
- le istruzioni da eseguire alla fine del ciclo.

Il ciclo può **terminare**:

- quando è stato eseguito un certo numero di ripetizioni noto;
- quando l'utente lo richiede esplicitamente;
- quando viene inserito un valore particolare;
- più in generale quando si verifica una condizione particolare dipendente dal problema.

Nota

Ci sono molti tipi di problemi in cui è l'algoritmo che deve stabilire quando terminare la ripetizione del procedimento, in base al verificarsi di una condizione dipendente dal problema.

Una scelta più tecnica riguarda il fatto di utilizzare:

- la ripetizione con controllo della condizione **in testa**, cioè all'inizio del ciclo (precondizionale);
- la ripetizione con controllo della condizione **in coda**, cioè alla fine del ciclo (postcondizionale).

La differenza sostanzialmente riguarda il fatto di valutare la condizione di terminazione prima di eseguire il ciclo o dopo averlo eseguito almeno una volta.

Nota: In realtà la ripetizione con controllo in testa e quella con controllo in coda sono equivalenti ed è sempre possibile utilizzare indifferentemente l'una o l'altra.

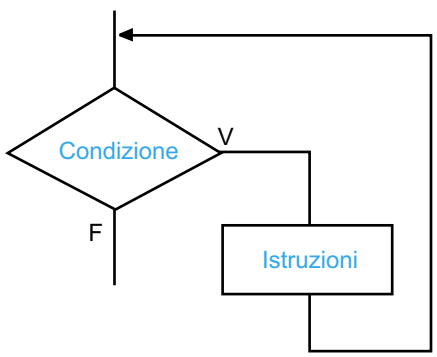
Un'altra differenza più marginale riguarda il fatto se si deve uscire dal ciclo quando la **condizione è vera** o quando la **condizione è falsa**.

Nota: Linguaggi diversi offrono costrutti leggermente diversi.

Per poter iniziare a realizzare qualche soluzione bisogna studiare i costrutti per la ripetizione.

Ripetizione con controllo in testa

```
mentre condizione
    istruzioni
fine mentre
```



Per **prima** cosa viene valutata la **condizione** (cioè stabilito se la condizione è *vera* o *falsa*); se la condizione risulta ***falsa* si esce** subito dal ciclo, altrimenti si eseguono le istruzioni del ciclo e poi si prova nuovamente la condizione; se è ancora ***vera* si ripetono** le istruzioni e si torna a provare la condizione, e così via finché la condizione risulta *falsa*. La condizione deve essere **inizializzata prima** di entrare nel ciclo.

Nota: All'interno del ciclo deve esserci un'istruzione che prima o poi fa diventare *falsa* la condizione (per evitare un loop infinito).

Forme alternative:

```
mentre not condizione
    istruzioni
fine mentre
```

```
finché condizione
    istruzioni
fine finché
```

Loop infinito

Se la condizione di terminazione di un ciclo non viene mai verificata, ci si trova in una situazione di **loop infinito** (un ciclo che non termina mai); bisogna quindi sempre accertarsi che sia possibile raggiungere la condizione di terminazione.

ESEMPIO 5.13

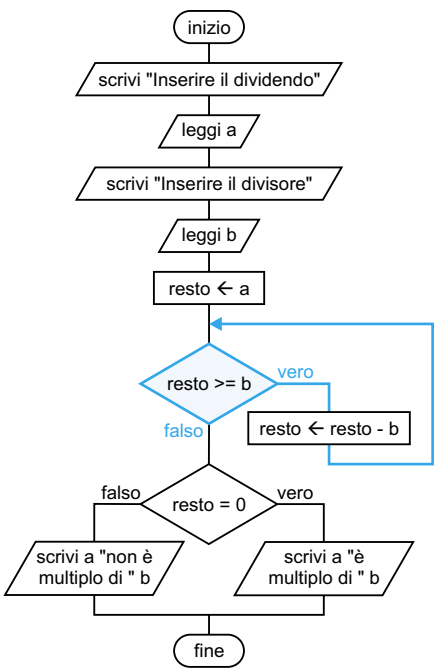
Determinare se un numero è multiplo di un altro procedendo per sottrazioni successive.

Considerando per semplicità solo numeri interi e positivi, si può procedere calcolando *a - b - b - ... - b* finché il risultato della sottrazione diventa minore di *b*.

La condizione di arresto del ciclo è che il risultato della sottrazione diventi minore del secondo numero; se il risultato è 0 il primo numero è multiplo del secondo, altrimenti no.

I	a	intero	dividendo
I	b	intero	divisore
	resto	intero	resto della divisione

```
inizio
    scrivi "Inserire il dividendo"
    leggi a
    scrivi "Inserire il divisore"
    leggi b
    resto ← a
    mentre resto ≥ b
        resto ← resto - b
    fine mentre
    se resto = 0 allora
        scrivi a "è multiplo di " b
    altrimenti
        scrivi a "non è multiplo di " b
    fine se
fine
```



C++ 5.6

L'ISTRUZIONE WHILE

L'istruzione ***while*** permette di codificare una ripetizione con controllo in testa.
Il formato dell'istruzione è

```
while(condizione)
    istruzione;
```

L'istruzione viene eseguita se la condizione è vera; la condizione viene valutata all'inizio del ciclo e dopo ogni esecuzione dell'istruzione del corpo del ciclo. Il ciclo termina appena l'espressione condizionale assume il valore *false*.

L'istruzione del corpo del ciclo può essere una istruzione composta (più istruzioni comprese tra parentesi graffe).

```
while(condizione){
    istruzione1;
    ...
    istruzioneN;
}//while
```

La valutazione dell'espressione booleana si ferma quando il valore è noto con certezza (short-cut evaluation).

Esempio 5.14

Codifica dell'esempio 5.13.

```
#include <iostream>
using namespace std;

int main(void){
    int a, b, resto;
    cout << "Inserire il dividendo ";
    cin >> a;
    cout << "Inserire il divisore ";
    cin >> b;
    resto = a;
    while(resto >= b)
        resto -= b;
    if(resto == 0)
        cout << a << " e' multiplo di " << b << endl;
    else
        cout << a << " non e' multiplo di " << b << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

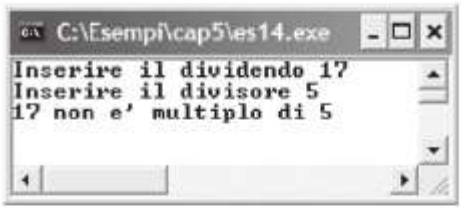
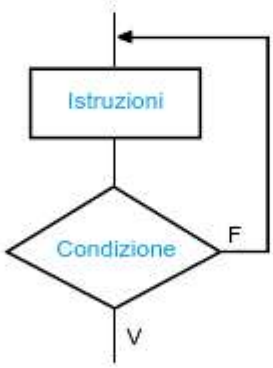


Figura 5.4 *Esecuzione dell'esempio*

Ripetizione con controllo in coda

ripeti
istruzioni
finché condizione

Per **prima** cosa vengono eseguite le **istruzioni**, senza considerare se la condizione sia *vera* o *falsa*; quindi le istruzioni del ciclo vengono sempre eseguite **almeno una volta**; poi si valuta se la **condizione** è *vera* o *falsa*; se la condizione risulta ***vera* il ciclo termina**, altrimenti se risulta ***falsa* si ripetono** le istruzioni e si valuta nuovamente la condizione.
La condizione può essere **inizializzata all'interno** del ciclo.



Nota

All'interno del ciclo deve esserci un'istruzione che prima o poi fa diventare *vera* la condizione (per evitare un loop infinito).

Forme alternative:

ripeti
istruzioni
finché not condizione

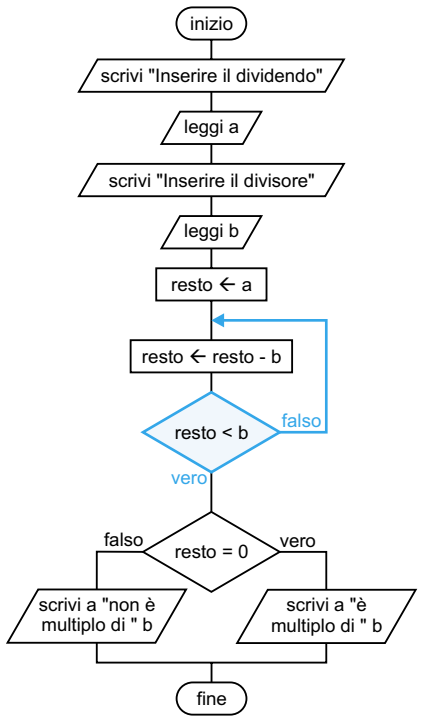
ripeti
istruzioni
mentre condizione

ESEMPIO 5.15

Determinare se un numero è multiplo di un altro procedendo per sottrazioni successive.

I	a	intero	dividendo
I	b	intero	divisore
	resto	intero	resto della divisione

```
inizio
    scrivi "Inserire il dividendo"
    leggi a
    scrivi "Inserire il divisore"
    leggi b
    resto ← a
    ripeti
        resto ← resto - b
    finché resto < b
    se resto = 0 allora
        scrivi a "è multiplo di " b
    altrimenti
        scrivi a "non è multiplo di " b
    fine se
fine
```



C++ 5.7 L'ISTRUZIONE DO .. WHILE

L'istruzione ***do .. while*** permette di codificare un ciclo con controllo in coda.
Il formato dell'istruzione è:

```
do
    istruzione
while(condizione);
```

L'istruzione *do .. while* termina con il punto e virgola alla fine della riga del *while*.
L'istruzione viene eseguita sempre almeno una volta; poi viene valutata la condizione; se la condizione è *falsa* il ciclo termina; se è *vera* viene eseguita nuovamente l'istruzione del corpo del ciclo e poi rivalutata la condizione.

L'istruzione del corpo del ciclo può essere una istruzione composta (più istruzioni comprese tra parentesi graffe).

```
do{
    istruzione1;
    ...
    istruzioneN;
}while(condizione);
```

La valutazione dell'espressione booleana si ferma quando il valore è noto con certezza (short-cut evaluation).

Esempio 5.16

Codifica dell'esempio 5.15.

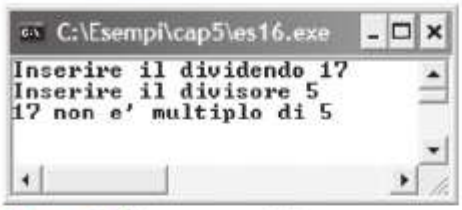


Figura 5.5 *Esecuzione dell'esempio*

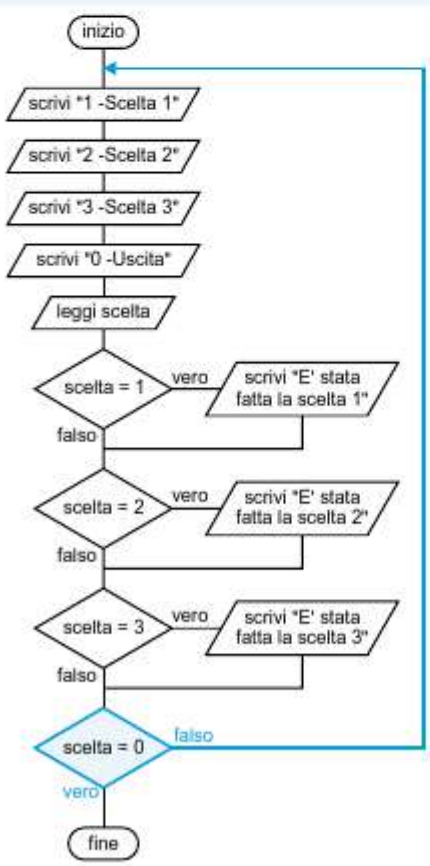
```
#include <iostream>
using namespace std;

int main(void){
    int a, b, resto;
    cout << "Inserire il dividendo ";
    cin >> a;
    cout << "Inserire il divisore ";
    cin >> b;
    resto = a;
    do
        resto -= b;
    while(!(resto < b));
    if(resto == 0)
        cout << a << " e' multiplo di " << b << endl;
    else
        cout << a << " non e' multiplo di " << b << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Esempio 5.17

Realizzare un menu di scelte.

I	scelta	intero	scelta voce del menu

```
inizio
    ripeti
        scrivi "1 -Scelta 1"
        scrivi "2 -Scelta 2"
        scrivi "3 -Scelta 3"
        scrivi "0 -Uscita"
        leggi scelta
        se scelta = 1 allora
            scrivi "E' stata fatta la scelta 1"
        fine se
        se scelta = 2 allora
            scrivi "E' stata fatta la scelta 2"
        fine se
        se scelta = 3 allora
            scrivi "E' stata fatta la scelta 3"
        fine se
    finché scelta = 0
fine
```




```
#include <iostream>
using namespace std;
```

```
int main(void){
    int scelta;
    do{
        cout << "1 - Scelta 1\n";
        cout << "2 - Scelta 2\n";
        cout << "3 - Scelta 3\n";
        cout << "0 - Uscita\n";
        cin >> scelta;
        if(scelta == 1)
            cout << "E' stata fatta la scelta 1\n";
        if(scelta == 2) cout << "E' stata fatta la scelta 2\n";
        if(scelta == 3) cout << "E' stata fatta la scelta 3\n";
    }while(!(scelta == 0));
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 5.6 *Esecuzione dell'esempio*

Contatori e totalizzatori

Le variabili **totalizzatori** servono per calcolare totali progressivi, come somme o prodotti; devono essere inizializzate con un valore neutro, cioè con un valore che non alteri i calcoli successivi (i valori neutri sono 0 per la somma e 1 per il prodotto).

Le variabili **contatori** servono per contare quante volte viene effettuata una certa operazione; i contatori vanno azzerati.

Nota

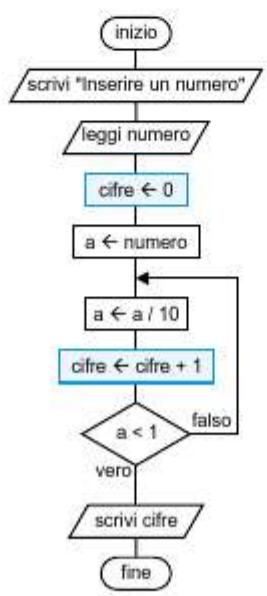
Contatori e totalizzatori devono essere sempre opportunamente inizializzati prima di entrare in un ciclo.

Esempio 5.18

Calcolare quante cifre compongono un numero intero.

I	numero	intero	numero da controllare
O	cifre	intero	contatore del numero di cifre
	a	intero	risultato delle divisioni del numero per 10

```
inizio
    scrivi "Inserire un numero"
    leggi numero
    cifre ← 0
    a ← numero
    ripeti
        a ← a / 10
        cifre ← cifre + 1
    finché a < 1
    scrivi cifre
fine
```



Per risolvere il problema bisogna far riferimento alla conoscenza specifica dell'argomento da trattare.

Bisogna ricordare che nel sistema decimale ogni cifra da destra a sinistra aumenta di valore di un fattore 10. Si possono contare le cifre dividendo più volte il numero per 10, finché il risultato della divisione diventa minore di 1, contando quante divisioni sono state effettuate: il numero delle divisioni necessarie fornisce il numero di cifre.

Viene sempre effettuata almeno una divisione poiché ogni numero (anche lo 0) ha almeno una cifra; si può usare la ripetizione con controllo in coda.

Ripetizione di un procedimento

Per capire i vari modi in cui può essere ripetuto un procedimento proviamo a risolvere lo **stesso problema** in **modi diversi**.

Esempio
Calcolo della media di alcuni numeri.

Anche un problema così semplice può essere risolto in molti modi.

Nota

Se il numero dei valori da sommare fosse già stabilito (e non troppo alto), per esempio 4 valori, si potrebbe realizzare l'algoritmo senza usare strutture cicliche.

Il programma prevede l'inserimento di vari numeri e il calcolo della somma.
Ad ogni ciclo bisogna leggere un numero e **calcolare la somma progressiva**.
Il **numero dei valori** da inserire potrebbe essere noto in anticipo o no.

Se **è noto** in anticipo, si deve chiedere all'utente all'inizio e **contare i valori** inseriti in modo da fermarsi quando è stato raggiunto il numero richiesto.

Se il numero di valori **non è noto** bisogna stabilire quando fermarsi: ci si potrebbe fermare quando si inserisce un **valore prestabilito** oppure **chiedere esplicitamente** all'utente cosa vuole fare. Mentre si inseriscono i valori bisogna **contare** quanti sono.

Numero di ripetizioni noto

Se il numero di ripetizioni è noto bisogna usare un **contatore** per stabilire quando si è raggiunto il numero di ripetizioni richiesto; basta contare le ripetizioni effettuate e terminare quando si raggiunge il valore desiderato.
il numero di ripetizioni, può essere un valore costante o una variabile, in cui però al momento dell'esecuzione del ciclo sia stato inserito un valore particolare.

Se il ciclo deve essere ripetuto ***n* volte** utilizzando un ciclo con **controllo in testa** si ha:

indice ← 1
mentre **indice** ≤ **n** esegui
    ...
    indice ← indice + 1
fine mentre

Nota

Come si vedrà più avanti, in alternativa si può usare un ciclo enumerativo:

per **indice da 1 a n**
  ...
fine per

Se il ciclo deve essere ripetuto ***n* volte** utilizzando un ciclo con **controllo in coda** si ha:

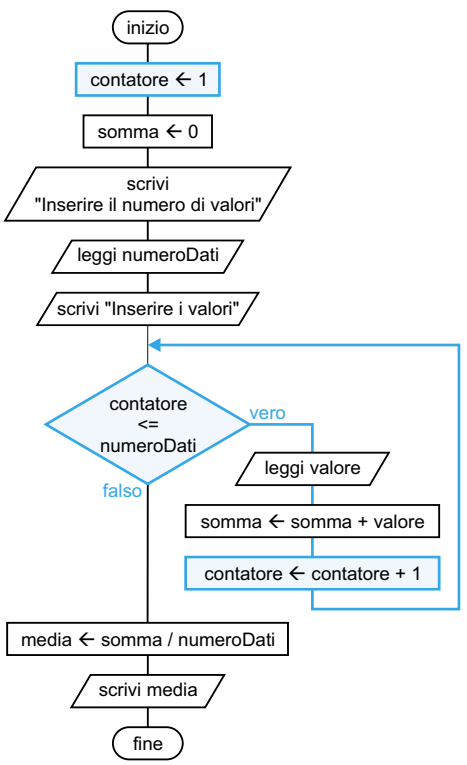
indice ← 0
ripeti
    indice ← indice + 1
    ...
finché **indice ≥ n**

Esempio 5.19

Calcolo della media usando la ripetizione con controllo in testa e considerando noto il numero di valori.

I	numeroDati	intero	numero dei valori
I	valore	numerico	ciascuno dei valori
O	media	numerico	media dei valori
	somma	numerico	somma dei valori
	contatore	intero	contatore dei valori

```
inizio
    contatore ← 1
    somma ← 0
    scrivi "Inserire il numero di valori"
    leggi numeroDati
    scrivi "Inserire i valori"
    mentre contatore ≤ numeroDati
        leggi valore
        somma ← somma + valore
        contatore ← contatore + 1
    fine mentre
    media ← somma / numeroDati
    scrivi media
fine
```



Nota

somma è una variabile totalizzatore, *contatore* è una variabile contatore.

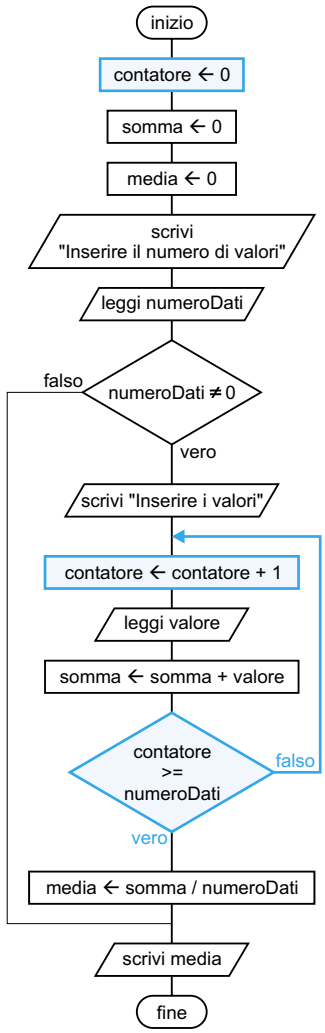
Esempio 5.20

Calcolo della media usando la ripetizione con controllo in coda e considerando noto il numero di valori.

Perché il funzionamento sia corretto bisogna controllare se *numeroDati* è 0.

I	numeroDati	intero	numero dei valori
I	valore	numerico	ciascuno dei valori
O	media	numerico	media dei valori
	somma	numerico	somma dei valori
	contatore	intero	contatore dei valori

```
inizio
    contatore ← 0
    somma ← 0
    media ← 0
    scrivi "Inserire il numero di valori"
    leggi numeroDati
    se numeroDati ≠ 0 allora
        scrivi "Inserire i valori"
        ripeti
            contatore ← contatore + 1
            leggi valore
            somma ← somma + valore
        finché contatore ≥ numeroDati
        media ← somma / numeroDati
    fine se
    scrivi media
fine
```



Nota

La differenza tra la soluzione con controllo in testa e quella con controllo in coda è se si vuole che il programma possa gestire anche il caso in cui non ci sia nessun numero disponibile o che si debba sempre inserire almeno un numero.

Richiesta esplicita di terminazione all'utente

Si può terminare il procedimento in base alla risposta dell'utente ad una **domanda di continuazione**. Il programma chiede all'utente che cosa vuole fare, se continuare o no, ed utilizza la sua risposta come condizione di terminazione.

Utilizzando il ciclo con controllo in testa

```
scrivi "Vuoi continuare?"
leggi risposta
mentre risposta = "s"
    ...
    scrivi "Vuoi continuare?"
    leggi risposta
fine mentre
```

Utilizzando il ciclo con controllo in coda

```
ripeti
    ...
    scrivi "Vuoi continuare?"
    leggi risposta
finché risposta = "n"
```

Con il ciclo con il controllo in testa il procedimento termina quando l'utente risponde con un carattere diverso da "s" alla domanda di continuazione.
Se l'utente risponde con un carattere diverso da "s" alla prima domanda, il programma termina senza eseguire neppure una volta il procedimento.

Con il ciclo con il controllo in coda il procedimento termina quando l'utente risponde "n" alla domanda di continuazione. Qualsiasi altro carattere fa proseguire il programma.

Nota

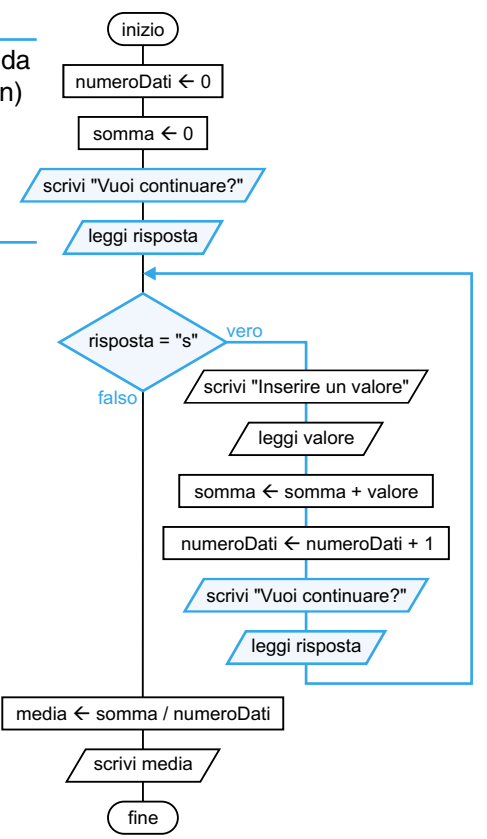
Il numero di ripetizioni non è noto in anticipo ma può essere conosciuto alla fine, usando un contatore per contare il numero di ripetizioni effettuate.

Esempio 5.21

Calcolo della media usando la ripetizione con controllo in testa e una richiesta all'utente per terminare il ciclo

I	risposta	carattere	risposta alla domanda di continuazione (s/n)
I	valore	numerico	ciascuno dei valori
O	media	numerico	media dei valori
	somma	numerico	somma dei valori
	numeroDati	intero	contatore dei valori

```
inizio
    numeroDati ← 0
    somma ← 0
    scrivi "Vuoi continuare?"
    leggi risposta
    mentre risposta = "s"
        scrivi "Inserire un valore"
        leggi valore
        somma ← somma + valore
        numeroDati ← numeroDati + 1
        scrivi "Vuoi continuare?"
        leggi risposta
    fine mentre
    media ← somma / numeroDati
    scrivi media
fine
```

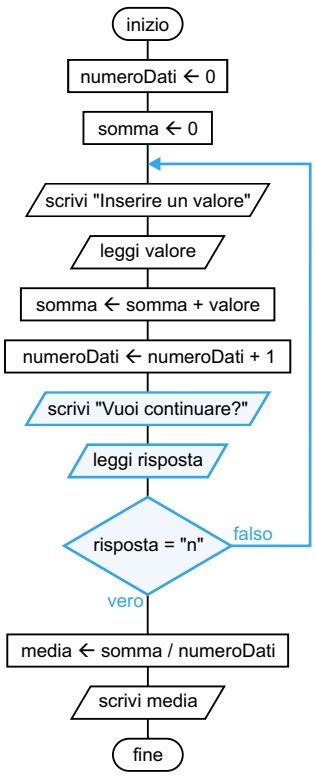


Esempio 5.22

Calcolo della media usando la ripetizione con controllo in coda e una richiesta all'utente per terminare il ciclo

I	risposta	carattere	risposta alla domanda di continuazione (s/n)
I	valore	numerico	ciascuno dei valori
O	media	numerico	media dei valori
	somma	numerico	somma dei valori
	numeroDati	intero	contatore dei valori

```
inizio
    numeroDati ← 0
    somma ← 0
    ripeti
        scrivi "Inserire un valore"
        leggi valore
        somma ← somma + valore
        numeroDati ← numeroDati + 1
        scrivi "Vuoi continuare?"
        leggi risposta
    finché risposta = "n"
    media ← somma / numeroDati
    scrivi media
fine
```



Nota

Gli algoritmi di ripetizione che utilizzano la presentazione all'utente della richiesta di continuazione per stabilire quando terminare possono risultare un po' fastidiosi. Per semplificare l'esecuzione si può fare in modo che il procedimento termini all'inserimento di un dato particolare.

Uso di un valore specifico per terminare il ciclo

Si può terminare il procedimento quando l'utente inserisce un **valore particolare** fissato come segnale di terminazione.

Il **segnale di terminazione** deve essere scelto tra i dati che non intervengono normalmente nell'elaborazione.

Si possono distinguere due casi:

- **non si deve elaborare il dato** che costituisce il segnale di terminazione; allora si deve usare un ciclo con controllo in testa;
- **si deve elaborare anche il dato** che costituisce il segnale di terminazione; allora si deve usare un ciclo con controllo in coda.

Ciclo con controllo in testa: il valore che fa terminare il programma non viene elaborato.

```
inizio
    scrivi "Inserire i valori"
    leggi valore
    mentre valore ≠ segnale
        ...
        leggi valore
    fine mentre
fine
```

Ciclo con controllo in coda: viene elaborato anche il valore che fa terminare il programma.

```
inizio
    scrivi "Inserire i valori"
    ripeti
        leggi valore
        ...
    finché valore = segnale
fine
```

Nota

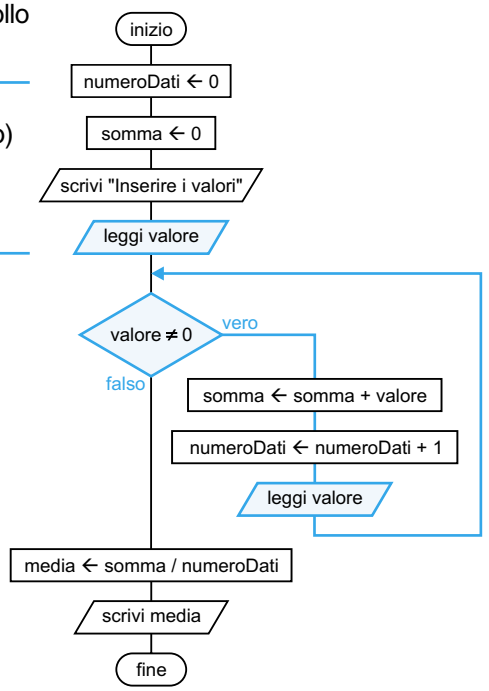
Si noti che nel ciclo con controllo in testa l'istruzione di lettura di un valore deve essere usata due volte: una prima di entrare nel ciclo e una all'interno del ciclo.

ESEMPIO 5.23

Calcolo della media usando la ripetizione con controllo in testa e il valore 0 per terminare il ciclo.

I	valore	numerico	ciascuno dei valori (0 fa terminare il ciclo)
O	media	numerico	media dei valori
	somma	numerico	somma dei valori
	numeroDati	intero	contatore dei valori

```
inizio
    numeroDati ← 0
    somma ← 0
    scrivi "Inserire i valori"
    leggi valore
    mentre valore ≠ 0
        somma ← somma + valore
        numeroDati ← numeroDati + 1
        leggi valore
    fine mentre
    media ← somma / numeroDati
    scrivi media
fine
```



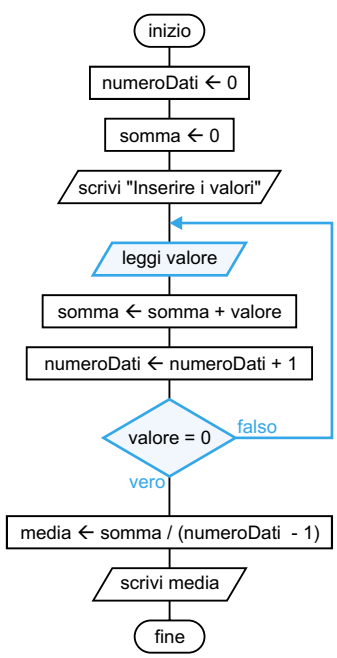
ESEMPIO 5.24

Calcolo della media usando la ripetizione con controllo in coda e il valore 0 per terminare il ciclo.

I	valore	numerico	ciascuno dei valori (0 fa terminare il ciclo)
O	media	numerico	media dei valori
	somma	numerico	somma dei valori
	numeroDati	intero	contatore dei valori

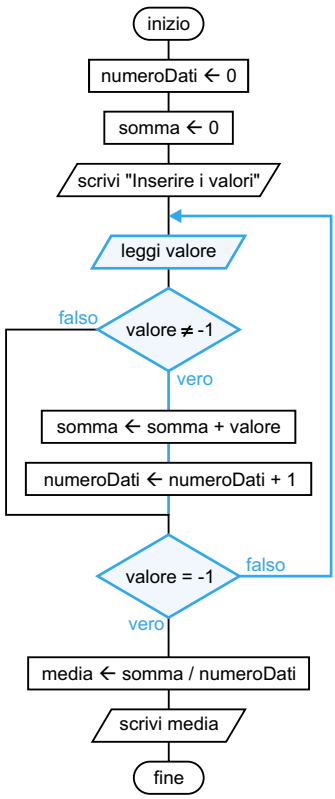
```
inizio
    numeroDati ← 0
    somma ← 0
    scrivi "Inserire i valori"
    ripeti
        leggi valore
        somma ← somma + valore
        numeroDati ← numeroDati + 1
    finché valore = 0
    media ← somma / (numeroDati -1)
    scrivi media
fine
```

Si noti che il valore che fa terminare il ciclo è stato elaborato; dato che è 0 e viene sommato non causa troppi problemi (basta decrementare *numeroDati*) ma non è sempre così.



Se come valore di terminazione fosse stato scelto -1 il risultato non sarebbe stato corretto; il problema in tal caso potrebbe essere risolto in questo modo:

```
inizio
    numeroDati ← 0
    somma ← 0
    scrivi "Inserire i valori"
    ripeti
        leggi valore
        se valore ≠ -1 allora
            somma ← somma + valore
            numeroDati ← numeroDati + 1
        fine se
    finché valore = -1
    media ← somma / numeroDati
    scrivi media
fine
```



Equivalenza delle strutture con controllo in testa o in coda

Un ciclo con controllo in **coda** può **sempre** essere realizzato utilizzando un ciclo con controllo in **testa**; basta fare in modo che le istruzioni del ciclo vengano sempre eseguite almeno una volta; per ottenere ciò, nel ciclo con controllo in testa bisogna fare in modo che la condizione sia sicuramente verificata prima dell'inizio del ciclo, con opportune inizializzazioni (qualche volta bisogna anche ripetere due volte un gruppo di istruzioni).

Viceversa, un ciclo con controllo in **testa** può **sempre** essere realizzato utilizzando un ciclo con controllo in **coda**, utilizzando una struttura condizionale aggiuntiva.

Con questi accorgimenti si possono rendere gli algoritmi che usano uno o l'altro ciclo assolutamente **equivalenti**, cioè si può fare in modo che diano esattamente gli stessi risultati.

Il fatto che l'uscita dal ciclo si verifichi quando la condizione risulta *falsa* nel ciclo con controllo in testa e quando risulta *vera* nel ciclo con controllo in coda (e quindi che le condizioni di uscita risultino diverse) è assolutamente irrilevante; il problema si può sempre risolvere utilizzando la condizione preceduta dall'operatore logico *not*, in modo da scambiarne il valore di verità. Questo è già stato fatto negli esempi 5.16 e 5.17.

Nota

Dagli esempi proposti si è visto come trasformare una struttura con controllo in testa in una con controllo in coda e viceversa.

La struttura ciclica con contatore (o enumerativa)

Quando il numero di ripetizioni da effettuare è noto in anticipo si può utilizzare un **ciclo enumerativo** che permette di descrivere questo tipo di algoritmi in modo più immediato.

per indice **da** valoreIniziale, **a** valoreFinale, con passo valorePasso
    istruzioni
fine per

L'**indice** del ciclo (o contatore) viene posto uguale al valore iniziale e confrontato col valore finale prima di entrare nel ciclo; se l'indice è minore del valore finale o uguale viene eseguita l'istruzione del corpo del ciclo; poi l'indice viene incrementato del passo (se il passo non è specificato, per default l'incremento è unitario) e confrontato nuovamente con il valore finale; il ciclo termina quando l'indice risulta maggiore del valore finale.

L'indice del ciclo può essere utilizzato nel gruppo di istruzioni all'interno del ciclo, ma non deve mai essere modificato.

L'**incremento dell'indice** non va mai specificato all'interno del ciclo: viene effettuato in modo automatico.

Anche questa struttura può essere usata in combinazione con altre, creando strutture nidificate.

Il ciclo enumerativo **equivale** a una struttura ciclica con controllo in testa in cui l'indice va inizializzato e incrementato in modo esplicito:

```
indice ← valoreIniziale
mentre indice ≤ valoreFinale esegui
    istruzioni
    indice ← indice + valorePasso
fine mentre
```

Caso particolare:

per indice **da** 1 **a** n con passo 1
    istruzioni
fine per

```
indice ← 1
mentre indice ≤ n esegui
    istruzioni
    indice ← indice + 1
fine mentre
```

Il passo del ciclo enumerativo

Il **passo** del ciclo enumerativo, cioè il valore da sommare all'indice ad ogni ripetizione, può essere diverso da 1, eventualmente anche negativo.
Se il passo è negativo l'indice viene decrementato; il valore iniziale deve quindi essere maggiore del valore finale perché il ciclo venga percorso.

Generazione di successioni

Il ciclo enumerativo si presta molto bene a risolvere problemi di **generazione di successioni numeriche**, cioè problemi in cui si deve calcolare una sequenza di valori (**successione**) data una legge generale per il calcolo di un valore generico.

C++ 5.8 L'istruzione for

L'istruzione ***for*** permette di codificare un ciclo enumerativo.

```
for(indice = valIniziale; indice <= valFinale; indice += incremento)
    istruzione;
```

incremento può avere valore sia positivo che negativo.

Nota — Il ciclo enumerativo è comunemente chiamato **ciclo for** dal nome dell'istruzione.

L'istruzione del corpo del ciclo può essere una istruzione composta (più istruzioni comprese tra parentesi graffe).

Se il passo è unitario positivo l'istruzione si può scrivere come:

```
for(indice = valoreIniziale; indice <= valoreFinale; indice++){
    istruzioni
}//for
```

L'**indice** del ciclo for può essere di tipo intero, carattere o booleano, e anche di un tipo enumerativo definito dall'utente. In questo ultimo caso, tuttavia, bisogna sovraccaricare l'operatore += (o l'operatore ++).

È possibile dichiarare nell'inizializzazione il tipo di *indice*; in tal caso *indice* esiste ed è visibile solo all'interno del ciclo:

```
for(tipo indice = valIniz; indice <= valFinale; indice += incremento){
    istruzioni
}//for
```

Nota

Conviene dichiarare l'indice all'interno del ciclo quando non viene più utilizzato nel resto del programma.

Il formato più generale è:

```
for(inizializzazione; condizione; aggiornamento){
    istruzioni
}//for
```

inizializzazione rappresenta l'assegnazione del valore iniziale all'indice del ciclo (e può comprendere la dichiarazione della variabile indice).

condizione è l'espressione booleana utilizzata per stabilire quando terminare il ciclo.

Il ciclo viene ripetuto mentre la condizione è **vera**; si esce dal ciclo *for* appena la condizione diventa falsa.

aggiornamento è un'espressione, in genere di incremento o decremento dell'indice del ciclo.

Nota

Inizializzazione e aggiornamento possono comprendere più istruzioni separate da una virgola.

Tutte le espressioni di un *for* sono **opzionali**; se viene omessa la condizione si presume che sia sempre *true*; in questo modo è possibile realizzare un ciclo infinito.

L'istruzione for può essere utilizzata in sostituzione delle istruzione ***while*** e ***do .. while***.

```
while(condizione){
    istruzioni
}//while
```

```
for( ; condizione; ){
    istruzioni
}//for
```

```
do{
    istruzioni
}while(condizione)
```

```
for(condizione=true; condizione; ){
    istruzioni
}//for
```

ISTRUZIONI DI SALTO

`break`	permette di uscire dai blocchi di istruzioni dei cicli;
`continue`	usata all'interno del blocco di istruzioni di un ciclo permette di saltare le successive istruzioni del ciclo e di passare alla successiva iterazione.

Nota Si sconsiglia l'uso delle istruzioni *break* e *continue* perché producono algoritmi non strutturati.

ESEMPIO 5.25

Stampa dei numeri da 1 a un valore limite inserito dall'utente.

L'indice del ciclo rappresenta di volta in volta uno dei valori da stampare.

I	n	intero	limite massimo dei numeri da stampare
O	i	intero	indice del ciclo e numero da stampare

```
inizio
    scrivi "Inserire il valore limite"
    leggi n
    per i da 1 a n
        scrivi i
    fine per
fine
```

```
#include <iostream>
using namespace std;

int main(void){
    int n;
    cout << "Inserire il valore limite ";
    cin >> n;
    for(int i = 1; i <= n; i++)
        cout << i << " ";
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 5.7 *Esecuzione dell'esempio*

ESEMPIO 5.26

Stampa dei multipli di 3 fino a un valore limite inserito.
Si tratta di un problema di generazione di una successione: bisogna individuare una legge che permetta di calcolare i valori successivi da produrre, in questo caso i multipli di 3. I multipli di 3 si possono calcolare facilmente in un ciclo partendo da 3 e aggiungendo un valore 3 ad ogni ripetizione.

Il problema viene risolto semplicemente utilizzando un ciclo for con passo 3.

I	n	intero	valore limite dei multipli di 3 da stampare
O	multiplo	intero	indice del ciclo e, di volta in volta, multiplo di 3

```
inizio
    scrivi "Inserire il valore limite"
    leggi n
```

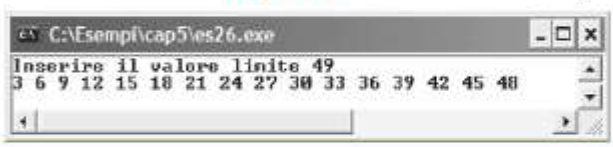
```
    per multiplo da 3 a n con passo 3
        scrivi multiplo
    fine per
fine
```

Il ciclo for termina quando l'indice diventa maggiore del valore finale; perciò l'algoritmo funziona senza causare loop anche se il valore limite inserito non è un multiplo di 3. In un ciclo while se la condizione di terminazione del ciclo fosse *mentre multiplo* $\neq n$ l'algoritmo sarebbe corretto per un valore di *n* multiplo di 3, ma si verificherebbe una condizione di loop infinito per un valore *n* che non fosse un multiplo esatto di 3.

Nota

```
#include <iostream>
using namespace std;

int main(void){
    int n;
    cout << "Inserire il valore limite ";
    cin >> n;
    for(int i = 3; i <= n; i += 3)
        cout << i << " ";
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 5.8 *Esecuzione dell'esempio*



ESEMPIO 5.27

Stampare i numeri naturali da N a 1 in ordine decrescente.

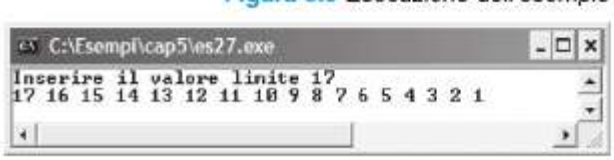
Si utilizza un ciclo con passo negativo.

I	n	intero	limite massimo dei numeri da stampare
O	i	intero	indice del ciclo e numero da stampare

```
inizio
    scrivi "Inserire il valore limite"
    leggi n
    per i da n a 1 con passo -1
        scrivi i
    fine per
fine
```

```
#include <iostream>
using namespace std;

int main(void){
    int n;
    cout << "Inserire il valore limite ";
    cin >> n;
    for(int i = n; i > 0; i--)
        cout << i << " ";
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 5.9 *Esecuzione dell'esempio*



Esempio 5.28

Ciclo con indice di tipo enumerativo.

```
#include <iostream>
using namespace std;

enum semi {cuori, quadri, fiori, picche};

/* Sovraccarico dell'operatore postfisso ++ affinchè possa operare
su un valore del tipo enumerativo semi */
semi operator++(semi& s, int i){
    i = s;
    s = (semi) (++i);
    return s;
}//operator++

int main(void){
    for(semi seme = cuori; seme <= picche; seme++)
        switch(seme){
        case cuori:
            cout << "cuori\n";
            break;
        case quadri:
            cout << "quadri\n";
            break;
        case fiori:
            cout << "fiori\n";
            break;
        case picche:
            cout << "picche\n";
        }//switch
    fflush(stdin);
    getchar();
    return 0;
}//main
```

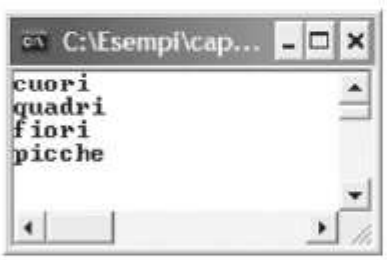


Figura 5.10 *Esecuzione dell'esempio*

I cicli enumerativi nidificati

Più cicli enumerativi possono essere **nidificati**; l'indice del ciclo deve essere diverso per ogni struttura (mentre per strutture in sequenza si può riutilizzare lo stesso indice).

per i da valoreIniziale1 a valoreFinale1 con passo valorePasso1
    per j da valoreIniziale2 a valoreFinale2 con passo valorePasso2
        istruzioni
    fine per
fine per

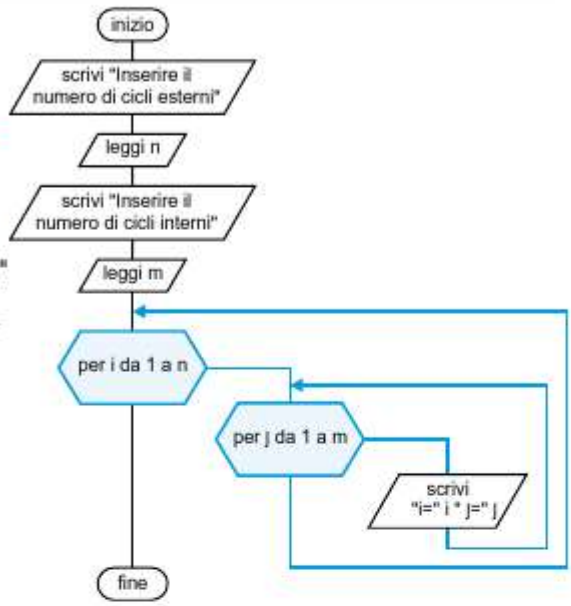
L'indice più esterno *i* parte dal valore iniziale; mantenendo fisso *i*, viene eseguito il ciclo più interno, variando *j* dal valore iniziale al valore finale; poi si incrementa *i* e si esegue di nuovo il ciclo più interno; si passa poi a incrementare nuovamente *i* e così via finché si raggiunge il valore finale.

ESEMPIO 5.29

Due cicli enumerativi nidificati; all'interno del ciclo vengono stampati gli indici; è possibile così verificare come vengono modificati durante l'esecuzione.

I	n, m	interi	numero delle ripetizioni
O	i, j	interi	indici dei cicli enumerativi

```
inizio
    scrivi "Inserire il numero di cicli esterni"
    leggi n
    scrivi "Inserire il numero di cicli interni"
    leggi m
    per i da 1 a n
        per j da 1 a m
            scrivi "i=" i " j=" j
        fine per
    fine per
fine
```




```
#include <iostream>
using namespace std;

int main(void){
    int n, m;
    cout << "Inserire il numero di cicli esterni ";
    cin >> n;
    cout << "Inserire il numero di cicli interni ";
    cin >> m;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
            cout << "i=" << i << " j=" << j << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

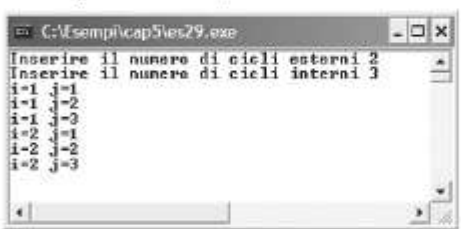


Figura 5.11 *Esecuzione dell'esempio*

ESEMPIO 5.30

Stampare la tavola pitagorica fino al fattore richiesto.

I	n	intero	ampiezza della tavola pitagorica desiderata
O	valore	intero	valore di ciascun elemento della tavola pitagorica
	i, j	interi	indici dei cicli, numeri da moltiplicare per ottenere ciascun valore

```
inizio
    scrivi "Inserire l'ampiezza della tavola pitagorica"
    leggi n
```

```
    per i da 1 a n
        per j da 1 a n
            valore ← i * j
            scrivi valore
        fine per
        scrivi ritorno a capo riga
    fine per
fine
```

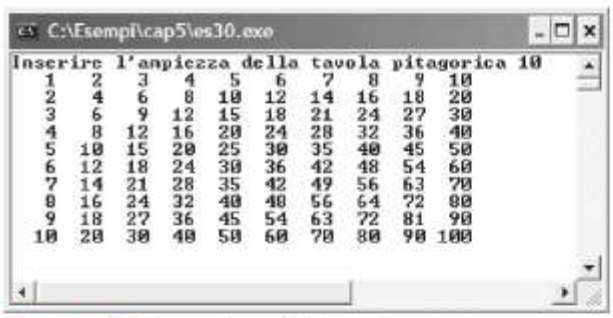


Figura 5.12 *Esecuzione dell'esempio*

```
#include <iostream>
#include <iomanip>
using namespace std;

int main(void){
    int n, valore;
    cout << "Inserire l'ampiezza della tavola pitagorica ";
    cin >> n;
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            valore = i * j;
            cout << setw(4) << valore;
        }//for
        cout << endl;
    }//for
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Trace con strutture cicliche

Eseguire la trace in programmi che utilizzano **strutture cicliche** può servire per determinare cosa accade **ad ogni ripetizione**. Per esempio:

- se il ciclo non termina mai si possono stampare all'interno del ciclo le variabili che intervengono nella condizione di terminazione per scoprire il motivo del **loop infinito**;
- se un calcolo prodotto con un ciclo risulta errato si possono stampare i **risultati intermedi** ottenuti ad ogni ripetizione;
- se un ciclo con controllo in testa o un ciclo enumerativo non viene mai eseguito (e si pensa che invece dovrebbe essere percorso) si possono stampare le variabili della **condizione** del ciclo con controllo in testa per determinarne il valore di verità o l'**indice** e il **limite** del ciclo enumerativo per stabilire se permettono l'ingresso nel ciclo.

Tabella di traccia dell'esempio 5.13:

	a	b	resto
leggi a	17		
leggi b	17	5	
resto ← a	17	5	17
resto ← resto - b	17	5	12
resto ← resto - b	17	5	7
resto ← resto - b	17	5	2

Esempio 5.31

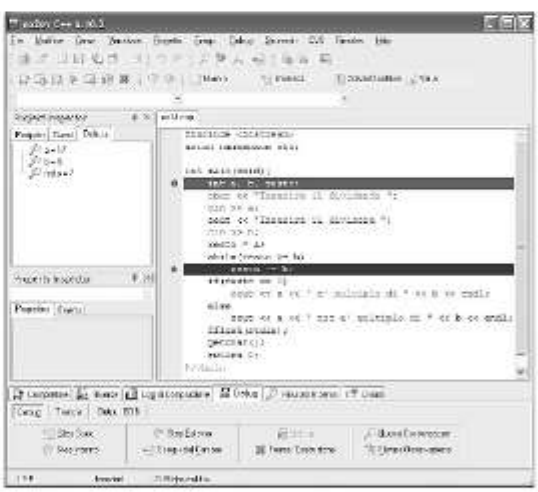
```
#include <iostream>
using namespace std;
```

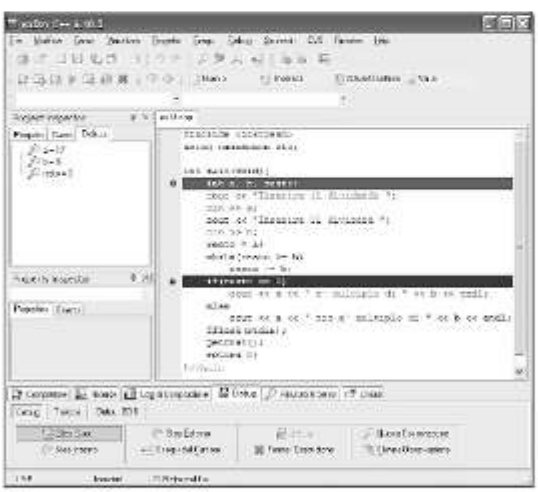
```
int main(void){
    int a, b, resto;
    cout << "Inserire il dividendo ";
    cin >> a;
    cout << "Inserire il divisore ";
    cin >> b;
    resto = a;
    while(resto >= b){
        resto -= b;
        cout << "resto = " << resto << endl;
    }//while
    if(resto == 0)
        cout << a << " e' multiplo di " << b << endl;
    else
        cout << a << " non e' multiplo di " << b << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 5.13 *Esecuzione dell'esempio*

Durante il Debug, la scheda *Debug* permette di tenere sotto controllo tutte le variazioni della variabile *resto*.

Figura 5.14 *Debug del programma per determinare se un numero è multiplo di un altro procedendo per sottrazioni successive*





In *gdb* l'uso di ***watch*** permette di tenere sotto controllo tutte le variazioni della variabile *resto*.

Figura 5.15
Debug con gdb del programma per determinare se un numero è multiplo di un altro procedendo per sottrazioni successive

5.5 IL CONTROLLO DEGLI ERRORI

Il problema dell'inserimento di dati errati

Se l'utente inserisce dei **dati errati** alle richieste del programma, si possono verificare degli **errori di esecuzione** oppure il programma può produrre dei **risultati privi di significato**.
L'utente potrebbe inserire dei valori **non ammessi** o **non previsti** dal procedimento, o valori apparentemente ammessi ma che causano errori di calcolo, o potrebbe commettere semplicemente degli **errori di scrittura**.

Il programma deve effettuare gli opportuni **controlli** per individuare e segnalare i casi di errore.

Se l'errore non viene segnalato dal programma, ma causa un errore durante l'esecuzione, il programma si blocca dando un messaggio di sistema; questa **situazione anomala** può mettere in crisi l'utente che non si rende conto del motivo di quanto è successo.
Se invece il programma riesce comunque a terminare, può dare in output risultati non corretti e l'utente potrebbe non accorgersene nemmeno.

Errori nell'input

Il tipo di dato inserito dall'utente deve essere **coerente** con quanto atteso dal programma: deve essere del tipo della variabile utilizzata per la lettura e deve avere un valore ammesso, per evitare errori di esecuzione.
Inoltre, ovviamente, deve essere effettivamente **corretto** anche come valore, se si vuole ottenere un risultato corretto.

L'inserimento di valori non ammessi dal tipo di variabile utilizzato provoca l'arresto dell'esecuzione del programma con una segnalazione di errore.
Questo è molto grave se i dati da inserire sono molti e l'errore si verifica dopo aver eseguito già una buona parte del lavoro; bisogna quindi prima di tutto specificare bene che cosa deve essere inserito con i **messaggi di richiesta**, in modo che l'utente possa capire cosa deve fare; il programma inoltre, invece di bloccarsi, dovrebbe controllare quello che è stato inserito, ed eventualmente chiedere di **reinserire solo il dato errato**.

Il controllo di valori particolari o compresi in intervalli di definizione

Il dato da inserire può avere soltanto **pochi valori** ammessi (per esempio i caratteri "S" o "N" per una risposta, o i numeri da 1 a 5 per la scelta di una tra 5 voci di menu), oppure molti ma all'interno di un **intervallo** (per esempio i numeri da 1 a 365 per i giorni di un anno, o quelli tra 0 e 100 per l'età di una persona), oppure ancora può essere un valore qualsiasi di un intervallo tranne certi valori (qualsiasi numero intero tranne lo zero o qualsiasi carattere tranne il punto).
Il programma deve controllare che il dato inserito sia **valido** in modo che l'esecuzione proceda correttamente.

La possibilità di correggere gli errori

Se l'inserimento di un dato non corretto fa terminare il programma con un messaggio di errore, per proseguire bisogna rieseguire da capo il programma, inserendo i dati corretti.
Se un programma è complesso e chiede di inserire molti dati questo procedimento non è adatto; ogni volta che si commette un errore si dovrebbe ricominciare da capo, commettendo facilmente altri errori.
Per evitare questo inconveniente, il programma deve permettere di **correggere il dato** quando riscontra un errore nell'inserimento.

È necessario utilizzare per la lettura del dato un ciclo che termina soltanto quando il dato introdotto è corretto.

ripeti
    leggi dato
finché il dato è corretto
elaborazione

Esempio 5.32

Eseguire un'operazione a scelta tra l'addizione e la sottrazione.

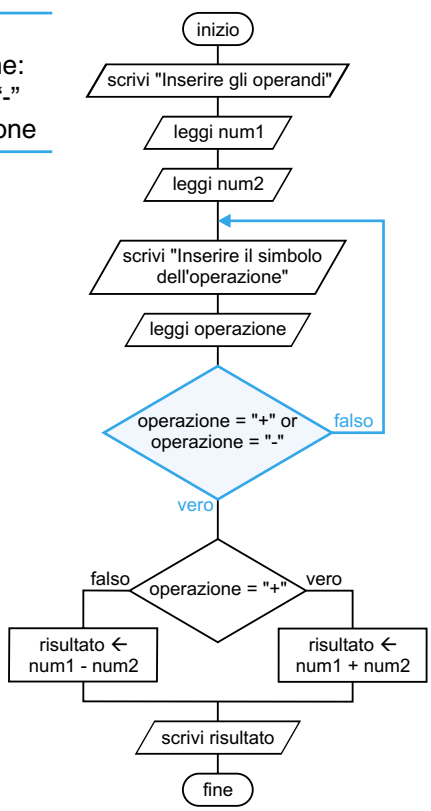
Il programma chiede all'utente due operandi e il simbolo dell'operazione da eseguire; come simbolo di operazione è possibile inserire soltanto uno dei caratteri "+" o "-".
Se il carattere inserito come simbolo di operazione non è uno di quelli ammessi, il programma permette di correggerlo.

Nota

Non è quindi necessario rilanciare il programma ed inserire nuovamente anche gli operandi.

I	num1, num2	numerici	operandi
I	operazione	carattere	simbolo di operazione: valori ammessi "+", "-"
O	risultato	numerico	risultato dell'operazione

```
inizio
    scrivi "Inserire gli operandi"
    leggi num1
    leggi num2
    ripeti
        scrivi "Inserire il simbolo dell'operazione"
        leggi operazione
    finché operazione = "+" or  operazione = "-"
    se operazione = "+" allora
        risultato ← num1 + num2
    altrimenti
        risultato ← num1 - num2
    fine se
    scrivi risultato
fine
```



La possibilità di interruzione del ciclo di controllo

Non sempre è sufficiente che il programma permetta di correggere il dato finché si inserisce un valore corretto; se l'utente continua a inserire valori errati il programma non riesce a proseguire; per far terminare comunque il programma non resterebbe altro da fare che interromperlo brutalmente.

Nota

In genere si possono interrompere i programmi usando la combinazione di tasti *Ctrl+c*; oppure si possono usare comandi di sistema per interrompere un processo.

La soluzione migliore è che il programma preveda anche la possibilità di **terminare** quando viene inserito un certo valore; perché questo metodo sia utile, comunque, deve essere segnalato in modo evidente all'utente.

```
ripeti
    leggi dato
finché il dato è corretto o è il segnale di uscita
se non è il segnale di uscita allora
    elaborazione
fine se
```

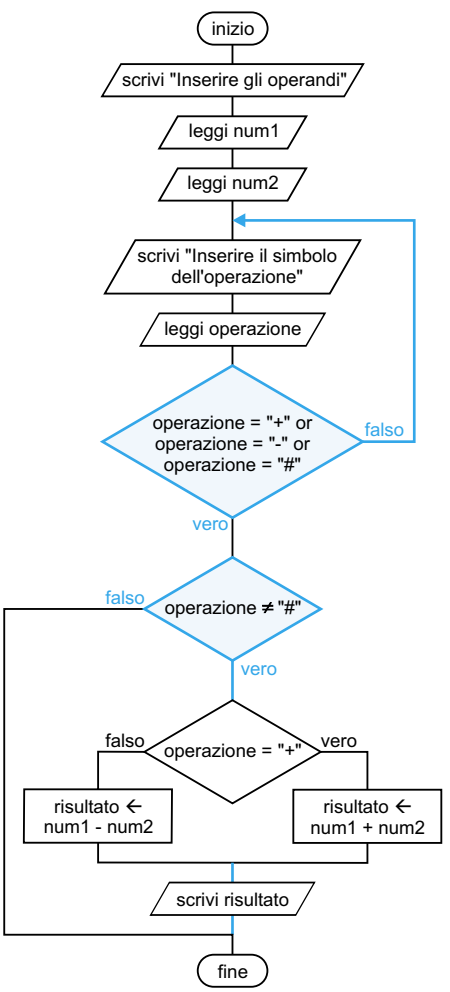
Esempio 5.33

Eseguire un'operazione a scelta tra l'addizione e la sottrazione.

Il programma chiede all'utente due operandi e il simbolo dell'operazione da eseguire; come simbolo di operazione è possibile inserire soltanto uno dei caratteri "+" o "-".
Se il carattere inserito come simbolo di operazione non è uno di quelli ammessi, il programma permette di correggerlo; permette però anche di far terminare il programma inserendo il simbolo "#".

I	num1, num2	numerici	operandi
I	operazione	carattere	simbolo di operazione: valori ammessi "+", "-" e "#"
O	risultato	numerico	risultato dell'operazione

```
inizio
    scrivi "Inserire gli operandi"
    leggi num1
    leggi num2
    ripeti
        scrivi "Inserire il simbolo dell'operazione"
        leggi operazione
    finché operazione = "+" or  operazione = "-"
    or operazione = "#"
    se operazione ≠ "#" allora
        se operazione = "+" allora
            risultato ← num1 + num2
        altrimenti
            risultato ← num1 - num2
        fine se
        scrivi risultato
    fine se
fine
```



Il problema della conferma dei dati

Il programma può controllare se i valori inseriti non corrispondono a valori ammessi; non è possibile in nessun modo però controllare se l'utente ha commesso un errore nello scrivere un dato, cioè se ha inserito un dato valido ma non corrispondente a quello che voleva effettivamente inserire.
L'unica soluzione a questo problema è riemettere i dati inseriti dall'utente e chiedere a lui stesso di **confermare** i valori. Se l'utente conferma l'inserimento il programma prosegue, altrimenti permette la correzione dei dati e ne chiede nuovamente la conferma.

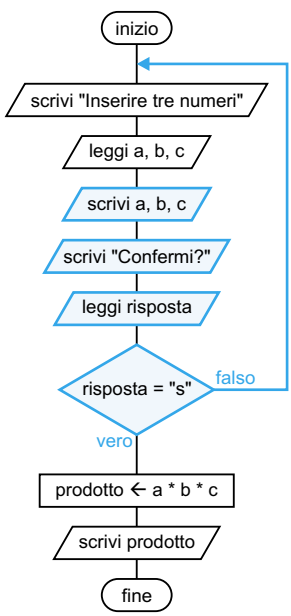
```
ripeti
    leggi dati
    scrivi dati
    scrivi "Confermi?"
    leggi risposta
finché risposta = "s"
elaborazione
```

Esempio 5.34

Richiedere tre numeri e calcolarne il prodotto, ma soltanto dopo che l'utente ha confermato i valori inseriti.

I	a, b, c	numerici	valori da moltiplicare
I	risposta	carattere	conferma della correttezza dei dati (s/n)
O	prodotto	numerico	prodotto dei valori inseriti

```
inizio
    ripeti
        scrivi "Inserire tre numeri"
        leggi a, b, c
        scrivi a, b, c
        scrivi "Confermi?"
        leggi risposta
    finché risposta = "s"
    prodotto ← a * b * c
    scrivi prodotto
fine
```



Nota Bisognerebbe anche controllare che la risposta alla domanda di conferma fosse un dato valido ("s" o "n").

Errori dovuti alla rappresentazione finita dei numeri

Il fatto che i numeri siano rappresentati con un numero finito di bit, e che quindi soltanto i valori di un certo intervallo siano ammessi, può causare degli **errori di esecuzione** al momento dell'inserimento dei valori o durante operazioni di calcolo.

I numeri in input

Se in una variabile numerica al momento dell'input viene inserito un valore **esterno all'intervallo** di definizione si verifica un errore di esecuzione.

Non è semplice evitare che si verifichino errori di questo tipo.
Un modo potrebbe essere quello di utilizzare per l'immissione un dato di un tipo più ampio e poi convertire il valore al tipo desiderato; per esempio si potrebbe inserire un numero intero in una variabile di tipo reale e verificare se tale valore è compreso nell'intervallo di definizione dei numeri interi, o addirittura utilizzare una variabile alfanumerica e poi controllare che il dato inserito possa essere considerato un valore numerico.

I NUMERI NEI CALCOLI

Anche se i valori iniziali sono compresi nell'intervallo di definizione ci possono essere degli errori dovuti al fatto che, attraverso dei calcoli, si ottengono dei valori non ammessi.

Per i **numeri interi** si potrebbe controllare, prima di eseguire l'operazione, che il risultato sia ancora un valore interno all'intervallo considerato.

> Per esempio se l'intervallo di definizione dei numeri interi va da -32.768 a 32.767 e si deve calcolare il cubo di un numero, bisogna che il valore di partenza sia compreso tra -32 e 31.

> Se si deve calcolare la somma di due numeri interi bisogna effettuare un controllo in modo relativo tra i dati da sommare: se il primo dato è positivo, il secondo deve essere minore della differenza tra il valore massimo e il primo dato; se il primo dato è negativo, il secondo deve essere maggiore della differenza tra il valore minimo e il primo dato.
> Questo controllo vale anche in procedimenti ripetuti, per esempio per una somma di più valori.

Molto più complesso diventa il problema del controllo degli errori utilizzando **numeri floating point**. Non tutti i numeri reali sono rappresentabili anche all'interno dell'intervallo di definizione a causa del numero limitato di cifre utilizzabili (precisione del numero); perdere alcune cifre significative della mantissa porta ad **errori di arrotondamento** che si possono verificare al momento dell'inserimento dei dati o per effetto di operazioni aritmetiche e che possono portare a risultati errati.

VERIFICA

QUESTIONARIO

1. Che cosa sono gli schemi di composizione fondamentali?
2. Che cos'è e a che cosa serve l'indentazione?
3. A che cosa serve la struttura sequenziale?
4. A che cosa servono le strutture condizionali?
5. A che cosa servono le strutture iterative o cicliche?

TEST

1 ..... *Indicare se le seguenti affermazioni sono vere o false.*

V	F	
☐	☐	1) Un ciclo con controllo in testa può sempre essere trasformato in un ciclo con controllo in coda.
☐	☐	2) Quando è noto il numero di ripetizioni si deve sempre usare un ciclo con contatore.
☐	☐	3) Un ciclo con contatore può sempre essere sostituito da un ciclo con controllo in testa.

V	F	
☐	☐	4) Un ciclo con contatore può sempre essere sostituito da un ciclo con controllo in coda.
☐	☐	5) Un ciclo con controllo in testa non può mai essere sostituito da un ciclo con contatore.
☐	☐	6) Un ciclo con controllo in coda non può mai essere sostituito da un ciclo con contatore.
☐	☐	7) Un ciclo con controllo in coda termina sempre quando la condizione di uscita è vera.
☐	☐	8) In certi casi il ciclo può non riuscire a terminare causando un loop infinito.
☐	☐	9) Le operazioni tra numeri floating point spesso non danno risultati esatti a causa di errori di arrotondamento.
☐	☐	10) Le operazioni tra numeri interi non causano errori di arrotondamento e quindi i risultati sono sempre esatti.

2 ..... *Individuare la risposta esatta.*

1) Quale output produce il programma seguente?

```
int main(void){
    int somma, k;
    somma = 0;
    k = 10;
    while((k - k/2) > 0){
        somma += k;
        k -=2;
    }//while
    cout << somma << endl;
    getchar();
    return 0;
}//main
```

☐ 45 ☐ 35
☐ 40 ☐ 30

2) Quale output produce il programma seguente?

```
const int T=8;
int main(void){
    int x = T;
    do
        x *= 2;
    while(x <= 8*T);
    cout << x << endl;
    getchar();
    return 0;
}//main
```

☐ 8 ☐ 128
☐ 64 ☐ 256

Esercizi

1 ..... Se la spesa è minore di 100 euro non c'è nessuno sconto; se è compresa tra 100 e 250 euro c'è uno sconto del 5%; se è compresa tra 250 e 500 c'è uno sconto del 10%; se è maggiore di 500 c'è uno sconto del 15%.
Il problema è ben definito? Dopo aver precisato la situazione ambigua, formalizzare sia la pseudocodifica che la codifica.

2 ..... Nell'esempio 5.3 confrontare le date di nascita espresse come anno, mese e giorno separati.

3 ..... Dato un tipo enumerativo con i nomi dei giorni della settimana:
- stampare i giorni;
- dato un giorno, stabilire qual è il precedente e il successivo.

4 ..... Inserire una serie di numeri con segno e dire quanti sono separatamente i valori positivi e negativi, utilizzando:
- una ripetizione con controllo in testa considerando noto il numero di valori;
- una ripetizione con controllo in coda considerando noto il numero di valori;
- una struttura con contatore considerando noto il numero di valori;
- una ripetizione con controllo in testa e richiesta esplicita per terminare;
- una ripetizione con controllo in coda e richiesta esplicita per terminare;
- una ripetizione con controllo in testa usando un valore specifico per terminare;
- una ripetizione con controllo in coda usando un valore specifico per terminare.

5 ..... Realizzare un menu per eseguire le quattro operazioni aritmetiche tra due numeri.

6 ..... Calcolare la media di *n* numeri usando un ciclo con contatore.

7 ..... Eseguire i seguenti esercizi usando un ciclo a contatore:
- calcolare i primi *k* multipli di un numero *n*;
- trovare i divisori interi di un numero *n*;
- stampare i quadrati e i cubi dei primi *n* numeri naturali;
- scrivere la somma dei quadrati dei primi *n* numeri naturali;
- calcolare la somma dei quadrati dei primi *n* numeri superiori a *m*;
- calcolare la somma di tutti i numeri dispari da 1 a *n*;
- calcolare la somma dei cubi dei primi *n* numeri pari.

8 ..... Stampare la tavola pitagorica, stampando anche come prima riga e prima colonna i valori che vengono moltiplicati tra di loro.

9 ..... Negli algoritmi con richiesta di continuazione controllare che la risposta alla domanda "Vuoi continuare?" sia solo uno dei caratteri "s" o "n"; in caso contrario ripetere la domanda.

10 ..... Controllare che un valore inserito sia numerico: leggerlo come stringa e controllare se soddisfa le condizioni: composto da cifre, ci può essere un punto decimale, può iniziare con + o -.

11 ..... Quale output produce il programma seguente?

```
const int T=6;
int main(void){
    int x = T;
    do{
       x *= 2;
        cout << x << endl;
    }while(x <= 10*T);
    getchar();
```

```
    return 0;
}//main
```

12 ..... Quale output produce il programma seguente?

```
for(int i = 1; i <= n; i++){
    for(int j = i; j > 0; j--)
        cout << i;
    cout << endl;
}//for
```

13 ..... Quale output produce il programma seguente?

```
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= 2*i; j++)
        cout << j*2 << " ";
    cout << endl;
}//for
```

PROPOSTE DI LAVORO

1 ..... Inserire un valore e stabilire se è positivo o negativo, e se è pari o dispari.

2 ..... Realizzare un programma che chieda due numeri e il loro prodotto e verifichi la correttezza della risposta.

3 ..... Dati i nomi e i tempi di gara di due concorrenti di una maratona espressi in ore, minuti e secondi stabilire chi è arrivato per primo.

4 ..... Data l'equazione di una retta nella forma $y=mx+q$ e le coordinate di un punto, dire se il punto appartiene alla retta.

5 ..... Dati tre numeri verificare se sono in ordine crescente o decrescente.

6 ..... Dati gli orari di tre appuntamenti disporli in ordine crescente e verificare che ci siano almeno 30 minuti tra un appuntamento e l'altro.

7 ..... Stabilire se tre numeri possono essere le misure dei lati di un triangolo (la misura del lato maggiore deve essere inferiore alla somma degli altri due); stabilire se si tratta di un triangolo isoscele, equilatero o scaleno, e se il triangolo è rettangolo (cioè se il quadrato del lato maggiore è uguale alla somma dei quadrati degli altri due).

8 ..... Dire se due numeri sono entrambi positivi o negativi e se sono entrambi pari o entrambi dispari.

9 ..... Verificare che tre nomi siano tutti diversi tra loro.

10 ..... Dato il numero da 1 a 12 corrispondente a un mese, scrivere il nome del mese.

11 ..... Convertire in lettere un numero inferiore a 999 (es.: 356 trecentocinquantasei).

12 ..... Realizzare un programma che permetta di decodificare le sigle delle province di una regione (il programma richiede di inserire la sigla di una provincia e fornisce il corrispondente nome per esteso).

13 ..... A scelta dell'utente eseguire una delle due trasformazioni:
- da un tempo in ore, minuti e secondi al valore corrispondente in secondi;
- da un tempo in secondi al valore corrispondente in ore, minuti e secondi.

14 ..... Il costo di un biglietto di ingresso è ridotto per i bambini di età inferiore a 14 anni, per gli studenti minori di 25 anni, per gli anziani con età maggiore di 70 anni e per tutti i componenti di una famiglia, se è composta da almeno 5 persone. Scrivere le domande necessarie a stabilire se il prezzo del biglietto è ridotto.

15 ..... Realizzare un programma che porga le domande di un questionario e produca un prospetto con le risposte.
Richiedere nome, anno di nascita, conoscenza delle lingue inglese, francese e spagnolo, e per ogni lingua conosciuta il livello (basso, medio, alto); se il livello per una lingua è alto stampare tre asterischi per evidenziarlo:
es.:
```
nome...   età...
inglese: alto ***
spagnolo: basso
```

16 ..... Realizzare un programma che porga le domande di un questionario e produca un prospetto con le risposte.
Richiedere il nome e la situazione (studente, lavoratore o disoccupato); agli studenti chiedere il tipo di scuola frequentata, ai lavoratori il tipo di lavoro e da quanti anni lavorano, ai disoccupati da quanto tempo sono senza lavoro.

17 ..... Data una serie di coppie nome prodotto e prezzo, stabilire qual è il prodotto che ha prezzo più alto.

18 ..... Data una serie di spese effettuate, calcolare la spesa totale.

19 ..... Calcolare il perimetro di una serie di poligoni di cui venga dato di volta in volta il numero di lati e la misura del lato.

20 ..... Dati i valori della popolazione di una serie di città per 2 anni consecutivi, determinare per quale città l'incremento è stato maggiore.

21 ..... Dati i voti riportati da alcuni studenti in una prova, stabilire quanti sono sufficienti e quanti insufficienti.

22 ..... Data una serie di temperature, stabilire quante sono superiori e quante inferiori allo zero e se la temperatura media è superiore o inferiore allo zero.

23 ..... Data una serie di dati, nome città e temperature minima e massima, individuare la città più fredda e quella più calda, e quelle con maggiore e minore escursione termica.

24 ..... Controllare se i tempi di arrivo di una serie di concorrenti sono disposti correttamente in ordine crescente:
- con i tempi espressi in secondi;
- con i tempi espressi in minuti e secondi.

25 ..... Data una espressione composta da addizioni e sottrazioni (in cui si alternano un numero e un operatore e che termina col simbolo =) calcolare il risultato.

26 ..... Dati nomi e date di nascita di una serie di persone, stabilire chi sono le due persone più giovani.

27 ..... Calcolare il minimo comune multiplo tra due numeri *m* e *n* procedendo nel seguente modo: confrontare i due valori e sommare se stesso al valore più piccolo, confrontare la somma ottenuta con l'altro valore, sommare un altro valore alla somma più piccola finché le due somme diventano uguali.
Esempio: m = 3, n = 5 3+3,5 6,5+5 6+3,10 9+3,10 12,10+5 12+3,15 15,15

28 ..... Data una serie di altezze, contare le persone che hanno altezza compresa tra due valori limite inseriti all'inizio.

29 ..... Data la successione 1, 2, 4, 8, 16, 32 ... in cui ogni elemento è il doppio del precedente, stabilire qual è il primo termine maggiore di un valore *n* introdotto.

30 ..... Trovare la radice quadrata intera di un numero, cioè il più grande numero intero *k* per cui $k^2 < n$.

31 ..... Stampare i multipli di un numero *k* compresi tra due numeri *n* e *m*.

32 ..... Stampare una alla volta le cifre di un numero.

33 ..... Sommare i primi numeri naturali fino ad ottenere un numero maggiore o uguale di un valore prefissato e dire quanti valori è stato necessario sommare.

34 ..... Moltiplicare i primi numeri naturali fino ad ottenere il massimo numero inferiore ad un valore prefissato.

35 ..... Dire quanti numeri pari a partire da 2 occorre moltiplicare per superare un valore *n*.

36 ..... Dati due numeri, verificare se sono primi fra loro (non hanno divisori comuni).

37 ..... Dato un numero, scomporlo in fattori primi.

38 ..... Dato un numero, scrivere una sola volta tutti i suoi fattori primi.

39 ..... Dato un numero *n*, calcolare la differenza tra la somma dei numeri pari e dei numeri dispari inferiori.

40 ..... Inserire una frase che termina col punto e stampare il numero di vocali e di consonanti presenti.

41 ..... Inserire una serie di caratteri e dire quante volte compaiono due caratteri uguali consecutivamente.

42 ..... Di più confezioni di un certo prodotto, di ognuna delle quali sia noto il peso e il prezzo, stabilire qual è la più conveniente dal punto di vista economico.

43 ..... Calcolare la media per ogni classe dei risultati ottenuti in una prova da più classi (non è noto inizialmente il numero di classi che partecipano alla gara). Controllare che i risultati siano compresi tra 1 e 10.

44 ..... Realizzare un programma che funzioni come un registratore di cassa: dopo aver richiesto la somma iniziale presente in cassa, deve calcolare per ogni cliente il totale da pagare, in base al numero di articoli di ciascun tipo acquistati e al prezzo unitario di ciascun articolo; al termine deve visualizzare la somma finale presente in cassa.

45 ..... Chiedere all'utente le varie parti che compongono il nome di un file (eventuale lettera di unità, parti del percorso, nome file ed estensione), poi creare una stringa con il nome completo del file (aggiungendo i simboli necessari come : e \ per *Windows*, / per *Linux*, . tra nome ed estensione) e stamparlo.

46 ..... Realizzare un programma per giocare a dadi col computer; ognuno lancia due dadi, vince il primo che ottiene lo stesso risultato sui due dadi.

47 ..... Realizzare un programma che porga le domande di un questionario e produca un prospetto con le risposte.
Richiedere nome, sesso, coniugato (sì o no) e se sì il nome del coniuge, se ha figli (sì, no) e se sì il numero e poi il nome dei figli.

48 ..... Realizzare un programma per chiedere a una serie di persone qual è la lingua conosciuta tra inglese, francese, spagnolo (ogni persona può dare una sola risposta. Riportare il numero delle persone e il numero e percentuale di ciascuna scelta e delle persone che non conoscono nessuna lingua.

49 ..... Realizzare un programma che, dopo aver chiesto il testo di una domanda, il testo di quattro possibili risposte e il numero della risposta esatta, porga tale domanda ad una serie di persone. Presentare il testo della domanda e delle risposte e chiedere di inserire il numero della risposta scelta (da 1 a 4); se la risposta è errata viene chiesta una seconda risposta. Si vuole ottenere il numero e la percentuale delle persone che hanno risposto esattamente al primo tentativo, di quelle che hanno risposto esattamente ma solo al secondo tentativo e di quelle che hanno risposto in modo errato.

50 ..... Si vuole gestire un conto corrente. Inserito il saldo iniziale, per ogni operazione deve essere indicato il tipo (versamento, prelievo, assegno) e la somma.
Il programma deve segnalare eventuali scoperti e in tal caso non permettere l'operazione relativa.
Si vuole ottenere a richiesta:
- il numero dei versamenti e la somma totale versata;
- il numero dei prelievi e la somma totale prelevata;
- il numero degli assegni e per che somma totale;
- il saldo finale.

Ciclo con contatore

51 ..... Stampare i primi *n* numeri che sono quadrati perfetti (1^2, 2^2, 3^2 ...).

52 ..... Calcolare tutte le terne pitagoriche con valori minori di 100, cioè i numeri *x*, *y*, *z*, tali che $x^2 + y^2 = z^2$.

53 ..... Un numero triangolare è un numero *n* uguale alla somma dei primi *n* numeri naturali; dato un numero, stabilire se è triangolare.

54 ..... Stampare i numeri dispari minori di un valore inserito, prima in ordine crescente, poi decrescente.

55 ..... Produrre i primi *n* numeri delle sequenze:
- 1, 3, 5, 7, 9 ...
- -1, -2, -3 ...
- 0, 1, 4, 9, 16, 25, 36, 49 ...
- 1, 2, 4, 8, 16, 32, 64, 128, 256 ...
- 1, -2, 3, -4, 5, -6 ...
- 1/2, 2/3, 3/4, 4/5 ...
- 1/2, 2/3, -3/4, 4/5, 5/6, -6/7... (negativi quelli che al numeratore sono multipli di 3)

56 ..... Stampare un triangolo rettangolo costituito da numeri consecutivi in modo che sulla riga *n* ci siano *n* numeri (triangolo di Floyd).

```
1
2   3
4   5   6
7   8   9   10
11  12  13  14  15
```

57 ..... Stampare un quadrato di asterischi di lato *n*.

58 ..... Stampare triangoli di *n* righe del tipo:

```
*          ***          *          *****
**         **          ***          ***
***        *          *****          *
```

59 ..... Stampare *n* righe del tipo:

```
...*...
..***..
.*****.
*******
```

60 ..... Stampare una riga per ogni lettera dell'alfabeto in modo che ogni lettera compaia *n* volte se *n* è la sua posizione nell'alfabeto (la *A* una volta, la *B* due volte, la *C* tre volte ecc.).
Poi fare la stessa cosa solo per le vocali (A, EE, III ...) e per le consonanti (B, CC, DDD, FFFF ...).
(Suggerimento: ricordare che il tipo *char* è un sottoinsieme del tipo *int*).

PROCEDURE E FUNZIONI

6

PREREQUISITI

Saper descrivere algoritmi e realizzare programmi in C++ che usano:

- strutture sequenziali
- strutture condizionali
- strutture cicliche

OBIETTIVI

CONOSCENZE

- Analisi top down
- Procedure e funzioni
- Variabili globali e locali e ambito di visibilità
- Parametri
- Ricorsione

ABILITÀ/CAPACITÀ

- Descrivere algoritmi utilizzando la metodologia top down
- Scrivere programmi in C++ che usano procedure e funzioni
- Realizzare file header e namespace personali

6.1 L'ANALISI TOP DOWN

La soluzione di un problema complesso può essere descritta attraverso istruzioni generiche, descritte più in dettaglio in passi successivi.
In pratica per risolvere un problema lo si può dividere in **sottoproblemi** più semplici.
Ogni sottoproblema deve essere descritto in seguito in modo dettagliato e, se è complesso, può a sua volta essere suddiviso ulteriormente in sottoproblemi.
Questa metodologia permette una maggior **astrazione** del problema: quando si divide il problema in sottoproblemi ci si disinteressa per il momento di come si risolverà poi ciascun sottoproblema.

Nella progettazione si procede per passi, per **raffinamenti successivi**, ponendo l'attenzione prima sui punti fondamentali. Questo modo di procedere viene detto **top down** cioè dall'alto al basso (o dal generale al particolare).

La metodologia top down è una metodologia di analisi dei problemi che si affianca alla metodologia bottom up, che parte invece dai particolari per poi combinare insieme le varie parti (strategia induttiva).

Nota

La metodologia top down ha molti **vantaggi**, tra cui:

- permette di semplificare la soluzione del problema, suddividendolo in problemi più semplici;
- permette di suddividere il lavoro tra persone diverse, che si possono occupare ciascuna di un sottoproblema;
- permette di rendere più comprensibile il lavoro e quindi più facile la manutenzione successiva.

6.2 I SOTTOPROGRAMMI

I linguaggi di programmazione permettono di suddividere il programma in **moduli** (chiamati sottoprogrammi, procedure o subroutine o anche funzioni).

Nel seguito si vedrà la differenza tra procedura e funzione.

Nota

Un problema, scomposto in sottoproblemi mediante l'analisi top down, può essere implementato utilizzando:

- un programma principale per il problema generale,
- un modulo per la soluzione di ciascun sottoproblema.

Il programma principale richiama il modulo necessario in un certo punto (il richiamo di solito avviene tramite il nome del modulo o una semplice istruzione). Un modulo può richiamare altri moduli (si parla di nidificazione delle procedure); un modulo inoltre può venire richiamato più volte, in punti diversi, senza la necessità di dover ripetere le istruzioni che lo descrivono.

I principali **vantaggi** dell'utilizzo di moduli separati sono:

- migliorano la leggibilità del programma;
- permettono di ridurre la quantità di codice da scrivere, se devono essere richiamati in più punti;
- sono riutilizzabili: si possono conservare moduli che risolvono problemi che si presentano frequentemente, per riutilizzarli in altri casi, creando delle librerie di software.

Funzionamento dei sottoprogrammi

Quando un programma richiama una procedura, si sospende l'esecuzione sul programma e si passa ad eseguire le istruzioni della procedura; quando la procedura termina, si riprende l'esecuzione del programma chiamante, dall'istruzione successiva a quella di richiamo della procedura (**rientro dalla procedura**).
Una procedura può a sua volta richiamare altre procedure e così via (**chiamate nidificate**); l'esecuzione viene sospesa e prosegue sulla procedura richiamata; quando la procedura termina, si riprende l'esecuzione del modulo chiamante, dall'istruzione successiva a quella di richiamo.

Uso di una pila per la gestione delle chiamate a procedure

Si possono gestire i rientri nell'ordine corretto memorizzando gli indirizzi di rientro nello stack: ogni volta che viene chiamata una procedura, si inserisce nello **stack** l'indirizzo dell'istruzione successiva a quella di richiamo; quando termina una procedura, si estrae dallo stack un valore; tale valore rappresenta l'indirizzo di rientro della procedura appena terminata.

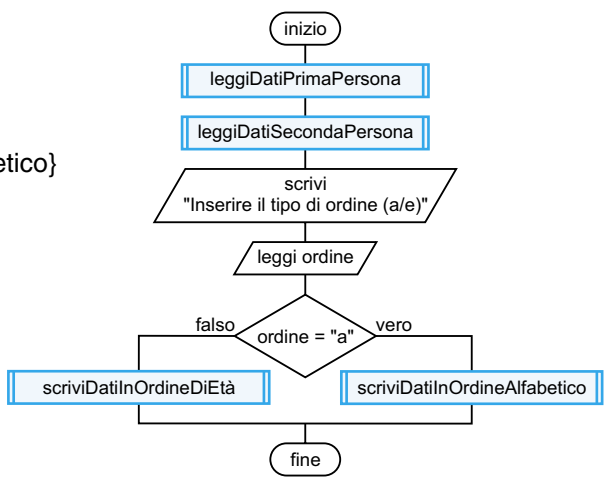
Esempio 6.1

Richiedere i nomi e le date di nascita di due persone; permettere poi, a richiesta, di ottenere i dati delle due persone in ordine alfabetico in base al nome, o in ordine di età.

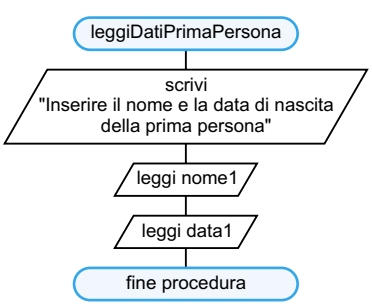
Si risolve il problema scomponendolo in sottoproblemi.

I/O	nome1, nome2	stringhe	nomi delle due persone
I/O	data1, data2	stringhe	date di nascita delle due persone
I	ordine	carattere	opzione per la scelta del tipo di ordinamento: "a" alfabetico, "e" per età

```
inizio
    leggiDatiPrimaPersona
    leggiDatiSecondaPersona
    scrivi "Inserire il tipo di ordine (a/e)"
    leggi ordine
    se ordine = "a" allora  {ordine alfabetico}
        scriviDatiInOrdineAlfabetico
    altrimenti  {ordine di età}
        scriviDatiInOrdineDiEtà
    fine se
fine
```



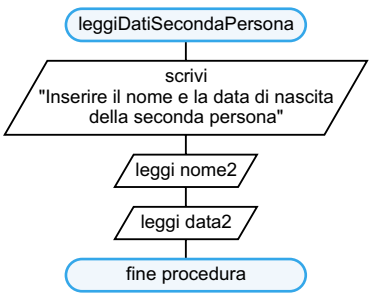

```
leggiDatiPrimaPersona
    scrivi "Inserire il nome e la data di nascita
    della prima persona"
    leggi nome1
    leggi data1
fine
```



leggiDatiSecondaPersona

scrivi "Inserire il nome e la data di nascita della seconda persona"
leggi nome2
leggi data2

fine



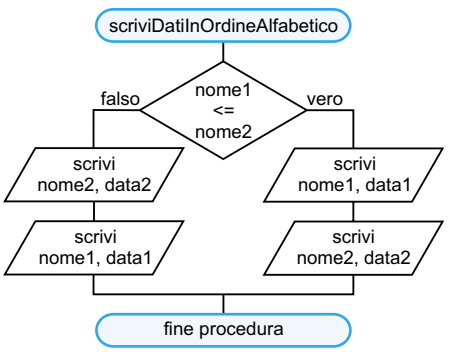
Nota

Si potrebbe anche utilizzare un unico modulo che permette di inserire i dati di una persona, usandolo più volte, ogni volta per inserire i dati di una persona diversa.

scriviDatiInOrdineAlfabetico

se nome1 ≤ nome2 allora
    scrivi nome1, data1
    scrivi nome2, data2
altrimenti
    scrivi nome2, data2
    scrivi nome1, data1
fine se

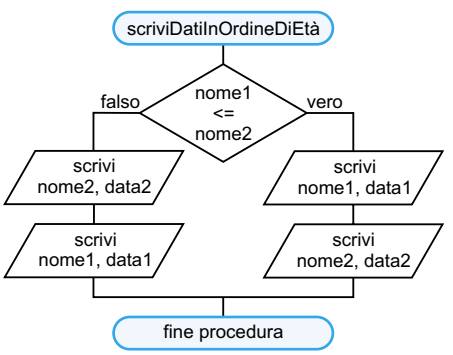
fine



scriviDatiInOrdineDiEtà

se data1 ≤ data2 allora
    scrivi nome1, data1
    scrivi nome2, data2
altrimenti
    scrivi nome2, data2
    scrivi nome1, data1
fine se

fine



Si potrebbero raffinare ulteriormente questi moduli considerando dei sottoproblemi per la stampa.

scriviAB

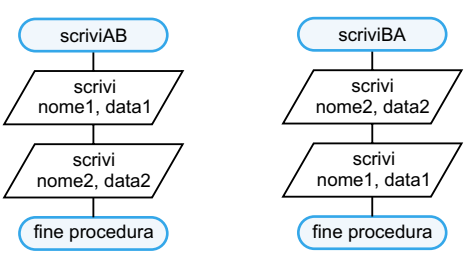
scrivi nome1, data1
scrivi nome2, data2

fine

scriviBA

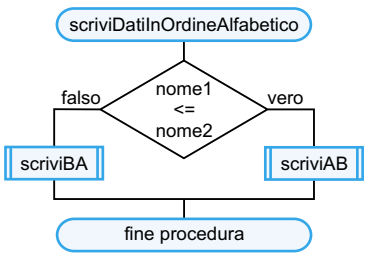
scrivi nome2, data2
scrivi nome1, data1

fine



e allora si avrebbe per esempio:

```
scriviDatiInOrdineAlfabetico
    se nome1 ≤ nome2 allora
        scriviAB
    altrimenti
        scriviBA
    fine se
fine
```



Si potrebbe anche utilizzare un unico modulo che permette di scrivere i dati di una persona, usandolo più volte, ogni volta per scrivere i dati di una persona diversa.

6.3 COME SCOMPORRE IL PROBLEMA

La suddivisione in moduli permette di semplificare un problema, ma se i moduli sono troppi si rischia di arrivare ad una complicazione ancora maggiore; non bisogna quindi esagerare nella scomposizione del problema.

Non esistono delle **regole** precise su come scomporre il problema da affrontare. La logica deve suggerire di volta in volta il modo più appropriato. Si possono comunque dare alcune indicazioni di carattere generale.

Conviene risolvere un problema con un modulo:

- se viene riutilizzato più volte nello stesso programma;
- se è un problema comune a più programmi, di interesse generale;
- se aumenta la leggibilità perché descrive in modo astratto un problema complesso.

Nota: Di solito vengono realizzati come moduli separati i moduli per la gestione dell'input/output, che possono comprendere alcuni controlli sui dati inseriti.

Ogni modulo deve svolgere un **compito preciso** e **nascondere** il suo funzionamento al modulo che lo richiama.

Ogni modulo deve avere una forte **coesione** e **pochi collegamenti con l'esterno**, deve cioè risolvere un problema particolare e avere pochi dati in comune con l'esterno.

Nota: Lo scambio dei dati con l'esterno avviene tramite i parametri, come si vedrà più avanti. Dire che il modulo deve avere pochi dati in comune con l'esterno significa che deve avere **pochi parametri di input/output**.

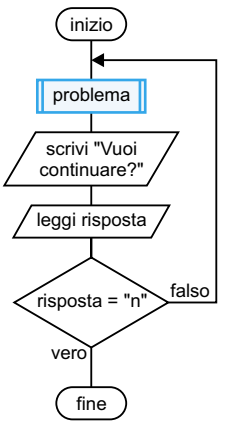
ESEMPIO 6.2

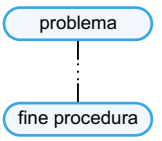
Ripetere più volte un problema a richiesta dell'utente.

Nota: Il problema può essere di qualsiasi tipo, per esempio il calcolo di una equazione di secondo grado proposto nell'esempio successivo.

I	risposta	carattere	risposta alla domanda di continuazione (s/n)

```
inizio
    ripeti
        problema
        scrivi "Vuoi continuare?"
        leggi risposta
    finché risposta = "n"
fine

problema
    ...
fine
```



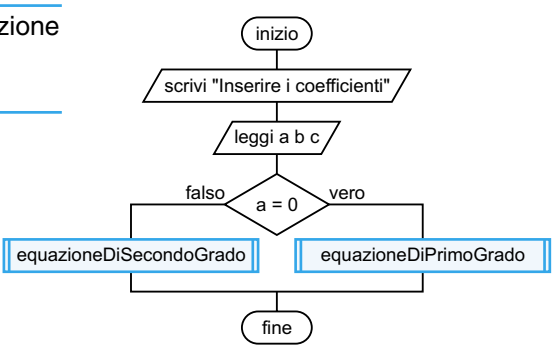


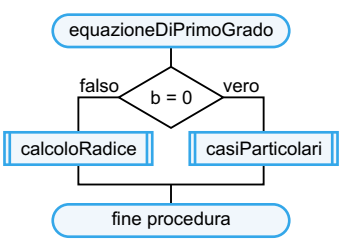
ESEMPIO 6.3

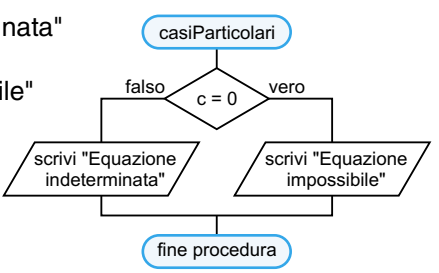
Soluzione di una equazione di secondo grado.

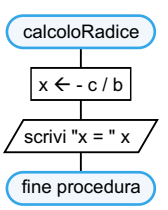
I	a, b, c	reali	coefficienti dell'equazione
O	x, x1	reali	radici dell'equazione
	d	reale	delta

```
inizio
    scrivi "Inserire i coefficienti"
    leggi a b c
    se a = 0 allora
        equazioneDiPrimoGrado
    altrimenti
        equazioneDiSecondoGrado
    fine se
fine

equazioneDiPrimoGrado
    se b = 0 allora
        casiParticolari
    altrimenti
        calcoloRadice
    fine se
fine

casiParticolari
    se c = 0 allora
        scrivi "Equazione indeterminata"
    altrimenti
        scrivi "Equazione impossibile"
    fine se
fine

calcoloRadice
    x ← -c / b
    scrivi "x = " x
fine
```

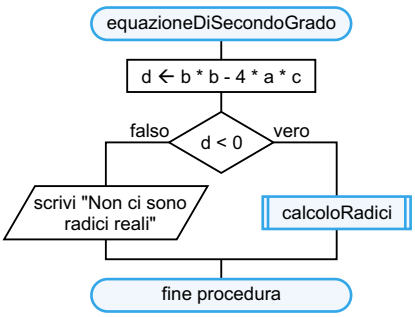


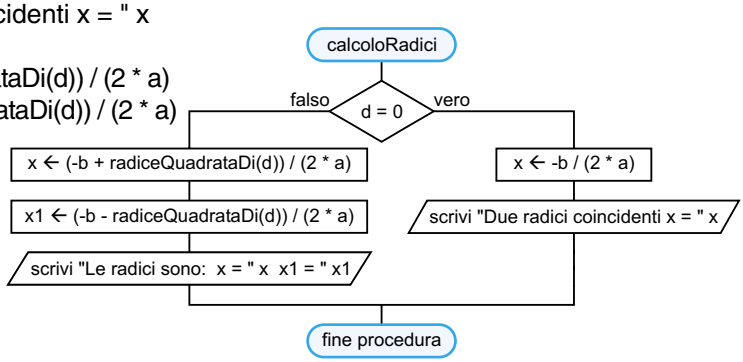






```
equazioneDiSecondoGrado
    d ← b * b - 4 * a * c
    se d < 0 allora
        scrivi "Non ci sono radici reali"
    altrimenti
        calcoloRadici
    fine se
fine

calcoloRadici
    se d = 0 allora
        x ← -b / (2 * a)
        scrivi "Due radici coincidenti x = " x
    altrimenti
        x ← (-b + radiceQuadrataDi(d)) / (2 * a)
        x1 ← (-b - radiceQuadrataDi(d)) / (2 * a)
        scrivi "Le radici sono:
        x = " x " x1 = " x1
    fine se
fine
```





Definizione e richiamo di procedure

C++ 6.1

Il concetto di procedura in C++ è definito come caso particolare di funzione: una funzione che non restituisce alcun valore.

La definizione di una procedura ha il seguente formato:

```
void nomeProcedura(void){
    istruzioni
}//nomeProcedura
```

Le istruzioni della procedura sono comprese tra le parentesi graffe.

La parola riservata ***void*** seguita prima del nome della procedura indica che non viene restituito alcun valore; la parola *void* tra parentesi dopo il nome indica che non viene passato alcun valore; è possibile anche non scrivere nulla tra le parentesi tonde.

Nota La definizione completa di funzione è presentata nel paragrafo C++ 6.4.

Per **richiamare** una procedura si usa come istruzione il **nome** della procedura stessa seguito dalle parentesi tonde.

Si devono definire prima le procedure di livello inferiore, cioè quelle che non richiamano al loro interno altre procedure, poi via via le altre, risalendo di livello, in modo che se una procedura ne richiama un'altra, la procedura richiamata sia già stata definita.
Il programma principale, che è esso stesso una funzione, va alla fine.

Esempio 6.4

Codifica dell'esempio 6.2.

```
#include <iostream>
using namespace std;
char risposta;

void problema(void){
    cout << "Soluzione del problema" << endl;
}//problema

int main(void){
    do{
        problema();
        cout << "Vuoi continuare? ";
        cin >> risposta;
    }while(risposta != 'n');
    getchar();
    return 0;
}//main
```

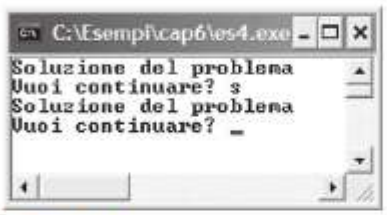


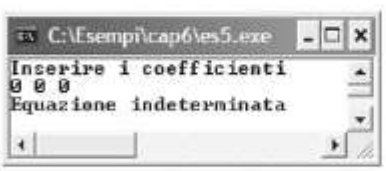
Figura 6.1 *Esecuzione dell'esempio*

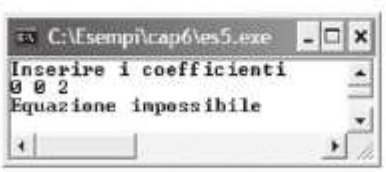
Esempio 6.5

Codifica dell'esempio 6.3.

Bisogna partire dalle procedure di livello inferiore, quindi per esempio da *casiParticolari* o indifferentemente da *calcoloRadice*.
equazioneDiPrimoGrado deve seguire le altre due, poiché le richiama.
equazioneDiSecondoGrado deve seguire *calcoloRadici* perché la richiama; è però indifferente la posizione di queste due procedure rispetto alle altre tre.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
float a, b, c, d, x, x1;

void casiParticolari(void){
    if(c == 0)
        cout << "Equazione indeterminata\n";
    else
        cout << "Equazione impossibile\n";
}//casiParticolari

void calcoloRadice(void){
    x = -c / b;
    cout << "x = " << setprecision(5) << x << endl;
}//calcoloRadice

void equazioneDiPrimoGrado(void){
    if(b == 0)
        casiParticolari();
    else
        calcoloRadice();
}//equazioneDiPrimoGrado
```





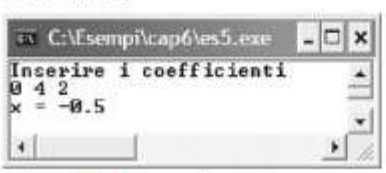


Figura 6.2 *Alcune fasi di esecuzione dell'esempio*

```
void calcoloRadici(void){
    if(d == 0){
        x = -b / (2 * a);
        cout << "Due radici coincidenti x = ";
        cout << setprecision(5) << x << endl;
    }//if
    else {
        x = (-b + sqrt(d)) / (2 * a);
        x1 = (-b - sqrt(d)) / (2 * a);
        cout << "le radici sono: x = " << setprecision(5) << x;
        cout << "  x1 = " << x1 << endl;
    }//else
}//calcoloRadici

void equazioneDiSecondoGrado(void){
    d = b * b - 4 * a * c;
    if(d < 0)
        cout << "Non ci sono radici reali\n";
    else
        calcoloRadici();
}//equazioneDiSecondoGrado

int main(void){
    cout << "Inserire i coefficienti\n";
    cin >> a >> b >> c;
    if(a == 0)
        equazioneDiPrimoGrado();
    else
        equazioneDiSecondoGrado();
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 6.3 *Alcune fasi di esecuzione dell'esempio*

6.4 LA GESTIONE DELLE VARIABILI

Variabili globali e locali

I diversi moduli (programma principale e procedure) che compongono il programma sono composti da istruzioni che operano su dati; alcuni dati possono essere necessari a più moduli, altri ad un solo modulo del programma.

Si dice che le variabili sono **globali** quando vi si può fare riferimento da qualunque modulo del programma; si dice invece che sono **locali** quando sono utilizzabili solo all'interno di un modulo (e dei suoi sottomoduli).

Nell'esempio 6.2 *risposta* è una variabile globale.

Le variabili **globali** sono allocate per tutta la durata del programma, cioè sono disponibili durante tutta l'esecuzione del programma.
Le variabili **locali** vengono allocate solo all'ingresso del modulo e deallocate all'uscita.

Le variabili globali sono memorizzate in locazioni della memoria, nell'area dati del programma.
Le variabili locali sono memorizzate nello stack.

Un modulo che modifica una variabile globale produce un **effetto collaterale** o **side effect**.

I side effect sono difficili da controllare e sono spesso causa di errori logici nei programmi.

Nota

Il modo in cui vengono gestite le variabili varia da linguaggio a linguaggio; per esempio alcuni linguaggi consentono di utilizzare solo variabili globali.

Normalmente è possibile definire **variabili locali** ad un modulo e scambiare dati tra il modulo chiamante e il modulo chiamato tramite una lista di variabili (**parametri**) messe in comune.

Nota

Nel seguito verrà approfondita questa situazione che è la più comune ed è la soluzione adottata dal C++.

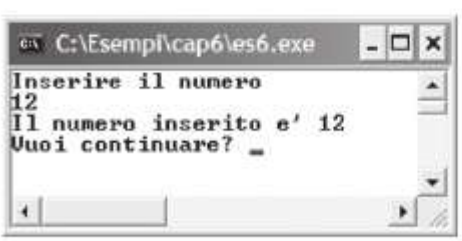
È bene, ogni volta che è possibile, evitare l'uso di variabili globali.

L'uso di variabili locali consente di rendere di più facile lettura e manutenzione i programmi, di evitare errori dovuti a side effect indesiderati, di riutilizzare più facilmente i moduli in altri programmi e di ridurre l'occupazione di memoria (perché la memoria occupata dalle variabili locali viene liberata all'uscita dal modulo).

Esempio 6.6

Variante dell'esempio 6.2 con variabili globali e locali.

```
#include <iostream>
using namespace std;
char risposta;          //variabile globale

void problema(void){
    int numero;         //variabile locale
    cout << "Inserire il numero" << endl;
    cin >> numero;
    cout << "Il numero inserito e' " << numero << endl;
}//problema

int main(void){
    do{
        problema();
        cout << "Vuoi continuare? ";
        cin >> risposta;
    }while(risposta != 'n');
    getchar();
    return 0;
}//main
```



La variabile *risposta* è globale.
La variabile *numero* è locale alla procedura *problema*.

Figura 6.4 *Esecuzione dell'esempio*

Ambito di visibilità degli identificatori (scope)

Ogni oggetto (qualsiasi risorsa: costanti, variabili o moduli) ha un suo **campo di validità** (**ambito di visibilità** o **scope**) in cui può essere usato.

Le **regole di visibilità** sono i criteri che definiscono il campo di validità di ciascun oggetto.

In genere valgono le seguenti regole:

- le variabili **globali** definite nel programma principale sono accessibili sia al programma principale che a ciascuno dei moduli; è cioè possibile farvi riferimento in qualsiasi modulo del programma;
- le variabili definite all'interno di un modulo sono **locali** al modulo e non possono essere utilizzate negli altri moduli o nel programma principale; non esistono al di fuori del modulo in cui sono definite.

In moduli diversi possono essere definite variabili con lo stesso nome (anche di tipo diverso) senza problemi poiché si tratta a tutti gli effetti di variabili diverse.

Un caso particolare che si può verificare è quello in cui variabili con lo stesso nome sono definite nel programma principale e in un modulo; anche questa situazione non causa alcun conflitto: nel modulo è valida solo la definizione locale; la variabile locale, all'interno del modulo, impedisce la visibilità della variabile globale (**shadowing**).

Visibilità degli identificatori

C++ 6.2

In C++ una variabile è detta **statica** se il suo periodo di vita termina solo con la fine dell'esecuzione, altrimenti è detta **dinamica**.

Le variabili globali (e le costanti globali) sono sempre statiche e solitamente sono dichiarate subito dopo le direttive *#include*. Possono essere dichiarate anche al di fuori della definizione di una qualsiasi procedura/funzione e sono visibili da tutti i moduli seguenti ma non precedenti.

Le variabili locali possono essere statiche o dinamiche e sono dichiarate in una procedura o funzione. Le variabili locali dinamiche vengono inizializzate ogni volta che si entra nella procedura o funzione, mentre quelle statiche mantengono ad ogni accesso alla funzione il valore che avevano nel precedente accesso.

Affinché una variabile locale sia statica è necessario dichiararla usando lo specificatore ***static***

```
static tipo nomevar;
```

Nota

Le variabili globali sono sempre statiche e non serve dichiararle con lo specificatore *static*.

In C++ ogni procedura può contenere definizioni di costanti e di variabili.

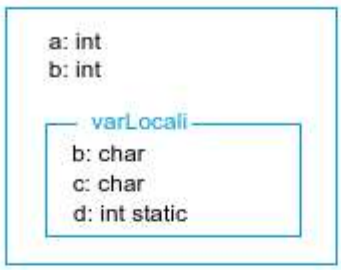
Anche ogni blocco (tutto ciò che è racchiuso tra due parentesi graffe) può contenere definizioni di costanti e variabili.

Ogni identificatore è visibile nel modulo o blocco in cui è definito.

Per accedere ad una variabile globale oscurata da una locale è possibile utilizzare l'operatore **::** chiamato, in questo contesto, **operatore di riferimento globale**.

Esempio 6.7

Variabili globali e locali.




```
#include <iostream>
using namespace std;
int a, b;           //variabili globali

void varLocali(void){
    static int d = 4; //variabile locale statica
    char b, c;        //variabili locali dinamiche
    b = 'h';
    c = 'k';
    cout << "a=" << a << " b=" << b << " c="
         << c << " d=" << d << endl;
    d *= 10;
    if(b < c){
        char a;    //variabile locale dinamica del blocco if
        a = b;
        b = a;
        c = a;
        cout << "locale a=" << a << endl;
        cout << "globale a=" << ::a << endl;  //variabile globale
    }//if
}//varLocali

int main(void){
    a = 3;
    b = 5;
    varLocali();
    cout << "a=" << a << " b=" << b << endl;
    varLocali();
    getchar();
    return 0;
}//main
```

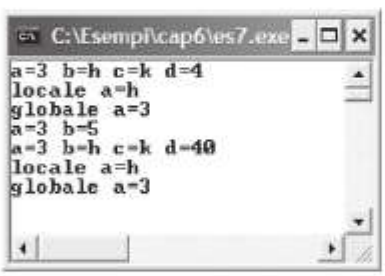


Figura 6.5 *Esecuzione dell'esempio*

Nel programma principale sono definite le variabili globali *a* e *b* di tipo intero.
Nel programma principale non sono visibili le variabili definite nella procedura.

Nella procedura sono definite:

- le variabili locali dinamiche *b* e *c* di tipo *char*;
- la variabile locale dinamica *a* del blocco *if* di tipo *char*;
- la variabile locale statica *d* di tipo *int*.

Nella procedura è inoltre utilizzabile la variabile globale *a*.

Se la procedura modifica la variabile globale *a* produce un side effect.

La variabile globale *b* rimane nascosta alla procedura, poiché nella procedura è definita una variabile locale con lo stesso nome. (La variabile globale *b* non perde né cambia il suo valore quando si entra nella procedura, ma rimane semplicemente non visibile; all'uscita dalla procedura la variabile ha ancora il valore che aveva in precedenza).

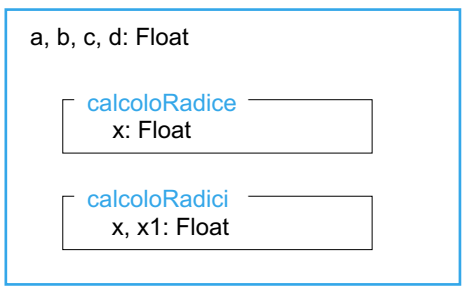
ESEMPIO 6.8

Soluzione dell'esempio 6.3 utilizzando variabili locali e procedure.
Sono riportate solo le procedure variate rispetto agli esempi 6.3 e 6.5.

Le variabili *a*, *b*, *c* e *d* sono globali.

La variabile *x* è locale alla procedura *calcoloRadice*.

Le variabili *x* e *x1* sono locali alla procedura *calcoloRadici*; la variabile *x* definita in questa procedura e la variabile *x* definita nella procedura *calcoloRadice* sono due variabili diverse; ognuna esiste soltanto all'interno della procedura in cui è definita.




```
float a, b, c, d;   //variabili globali

void calcoloRadice(void){
    float x;        //variabile locale
    x = -c / b;
    cout << "x = " << setprecision(5) << x << endl;
}//calcoloRadice

void calcoloRadici(void){
    float x, x1;    //variabili locali
    if(d == 0){
        x = -b / (2 * a);
        cout << "Due radici coincidenti x = ";
        cout << setprecision(5) << x << endl;
    }//if
    else {
        x = (-b + sqrt(d)) / (2 * a);
        x1 = (-b - sqrt(d)) / (2 * a);
        cout << "le radici sono: x = " << setprecision(5) << x;
        cout << "  x1 = " << x1 << endl;
    }//else
}//calcoloRadici
```

6.5 I PARAMETRI

Usare molte variabili globali e lavorare direttamente su queste all'interno delle procedure non è un buon metodo di programmazione.
Il modo migliore di procedere è passare i dati che servono alle procedure come parametri e ricevere i risultati allo stesso modo.

I **parametri** (o **argomenti**) sono una lista di dati che può essere scambiata tra un modulo e l'altro. Le variabili che compaiono nella definizione di un modulo vengono dette parametri **formali**; i dati che vengono sostituiti ai parametri formali al momento del richiamo del modulo vengono detti parametri **attuali** (o **argomenti attuali**).

Nella definizione della procedura vengono descritti i **parametri formali**, cioè le variabili utilizzate all'interno della procedura.

I parametri formali dal punto di vista dell'ambito di visibilità si comportano come variabili locali: sono visibili all'interno del modulo in cui sono definiti e di tutti i suoi sottomoduli e nascondono eventuali variabili globali con lo stesso nome.

I parametri formali si distinguono in parametri di **input** se permettono al modulo chiamante di fornire un valore al modulo chiamato, e parametri di **output** se permettono al modulo chiamante di ricevere un valore dal modulo chiamato; alcuni parametri possono essere di input/output.

Al momento del richiamo della procedura si devono fornire i **parametri attuali**, cioè i valori che andranno a sostituirsi ai parametri formali.

Il nome delle variabili usate come parametri può essere diverso tra i parametri formali e i parametri attuali; l'assegnazione dei valori avviene in base alla **posizione**, associando il primo parametro attuale al primo parametro formale, il secondo parametro attuale al secondo parametro formale e così via.
I parametri attuali nell'istruzione di chiamata di un modulo e i parametri formali nella definizione del modulo stesso devono essere uguali per numero e tipo.

Parametri formali

I parametri formali sono identificati da:

- un identificatore;
- un tipo;
- una posizione;
- una direzione (input/output) che dipende dalla modalità di passaggio del parametro.

Nel momento in cui vengono usati, cioè il richiamo della procedura e la sostituzione dei parametri attuali, sono inoltre caratterizzati da un **valore**.

Nella documentazione per i programmatori che devono riutilizzare il modulo è importante precisare, in base alla posizione, il tipo e la direzione del parametro.

Le modalità di passaggio dei parametri

Il passaggio dei parametri può avvenire con **modalità** diverse; le modalità principali sono quella per valore e quella per riferimento o indirizzo.

Nella trasmissione **per valore** al parametro formale viene trasmesso il valore del parametro attuale, ma i due dati sono distinti, in locazioni diverse di memoria.

Il parametro formale è una **copia** temporanea del dato; quindi, anche se il parametro viene modificato all'interno del modulo, non si ha alcun effetto sul parametro attuale, una volta che il modulo è terminato (le modifiche non hanno effetto sulle variabili del modulo chiamante).

Nella trasmissione **per riferimento** o **indirizzo** il parametro formale e il parametro attuale fanno riferimento alla stessa locazione di memoria.

Al modulo chiamato viene trasmesso l'**indirizzo** del parametro (puntatore); il modulo accede direttamente alla variabile e quindi qualsiasi modifica della variabile all'interno del modulo ha effetto anche dopo che il modulo è terminato; si deve usare il passaggio per riferimento per restituire valori al modulo chiamante.

I parametri di una procedura possono essere passati in parte per valore e in parte per riferimento.

I parametri che devono essere modificati dalla procedura (parametri di **output** o di **input/output**) devono essere passati per **riferimento**. I parametri che non devono essere modificati (parametri di **input**) devono essere passati per **valore**.

A volte viene utilizzato il passaggio per riferimento anche se i parametri non devono venire modificati. Infatti il passaggio per riferimento permette di ridurre l'occupazione di memoria, dato che nel passaggio per valore viene fatta una copia del parametro attuale, mentre nel passaggio per riferimento viene trasmesso solo l'indirizzo.
Ciò avviene soprattutto quando i parametri sono strutture di dati, magari di grandi dimensioni, come gli array. In genere i file addirittura non possono essere passati per valore, ma devono essere passati solo per riferimento.

Nota

Esempio 6.9

Per capire le modalità del passaggio dei parametri.

```
I    x    intero
I/O  y    intero

incrementa (val x, rif y)
    x ← x + 1
    y ← y + 1
    scrivi x y          output: 1 1
fine
```

```
I/O  a, b  interi

inizio
    a ← 0
    b ← 0
    incrementa(a, b)
    scrivi a b          output: 0 1
fine
```

Progettazione delle procedure con parametri

Alcuni consigli per la progettazione di procedure parametriche:

- le procedure devono comunicare all'esterno solo tramite parametri;
- il numero dei parametri non deve essere eccessivo (altrimenti forse è sbagliata la progettazione);
- le procedure non devono usare variabili globali e soprattutto non devono modificarle (cioè non devono produrre side effect);
- le procedure che risolvono un certo problema non devono usare istruzioni di input/output: devono comunicare i dati solo tramite parametri;
- la gestione dell'input/output deve avvenire in procedure apposite.

Le procedure parametriche

C++ 6.3

La definizione di una procedura parametrica ha il formato:

```
void nomeProcedura(listaParametriFormali)
```

I **parametri formali** vanno indicati tra parentesi separati da virgole.

Per ogni **parametro** deve essere specificato il tipo:

```
tipo nomeParametro
```

I parametri definiti in questo modo sono passati per **valore**.

Se i parametri sono passati per **riferimento** allora il tipo è seguito dal carattere **&** chiamato, in questo contesto, **operatore di referenza**:

```
tipo& nomeParametro
```

Il C++ possiede anche una sintassi diversa per il passaggio dei parametri per indirizzo, derivante dal C, che si basa sul concetto vero e proprio di puntatore. Tale sintassi sarà maggiormente utilizzata nel capitolo 8.

Nota

Per richiamare una procedura parametrica bisogna specificare tra parentesi, dopo il nome della procedura, la lista dei **parametri attuali**.

```
nomeProcedura(listaParametriAttuali);
```

I parametri passati per **valore** possono essere **variabili di tipo compatibile**, o anche **costanti** o **espressioni** che danno un valore compatibile.
Le conversioni di tipo vengono effettuate implicitamente, in modo automatico.

I parametri passati per valore usano pesantemente lo stack. La dimensione totale di tutti i parametri formali non deve essere superiore a 32 KB (o 64 KB secondo le piattaforme).

Nota

Nel passaggio per **riferimento** i parametri attuali devono essere **esattamente dello stesso tipo** dei parametri formali, altrimenti viene segnalato un errore di compilazione.

Esempio 6.10

Variazione dell'esempio 6.8 con l'uso di procedure parametriche.
Sono riportate solo le procedure modificate rispetto all'esempio 6.8.

```
float a, b, c;   //variabili globali

void calcoloRadici(float delta){
    float x, x1;    //variabili locali
    if(delta == 0){
        x = -b / (2 * a);
        cout << "Due radici coincidenti x = ";
        cout << setprecision(5) << x << endl;
    }//if
    else {
        x = (-b + sqrt(delta)) / (2 * a);
        x1 = (-b - sqrt(delta)) / (2 * a);
        cout << "le radici sono: x = " << setprecision(5) << x;
        cout << "  x1 = " << x1 << endl;
    }//else
}//calcoloRadici

void equazioneDiSecondoGrado(void){
    float d = b * b - 4 * a * c;    //variabile locale
    if(d < 0)
        cout << "Non ci sono radici reali\n";
    else
        calcoloRadici(d);
}//equazioneDiSecondoGrado
```

Esempio 6.11

Procedura per la ricerca del massimo tra due numeri interi.

I	a, b	interi	valori da confrontare
O	max	intero	massimo

cercaMax(val a, val b, rif max);
    se a >= b allora
        max ← a
    altrimenti
        max ← b
    fine se
fine

```
void cercaMax(int a, int b, int& max){
    if(a >= b)
        max = a;
    else
        max = b;
}//cercaMax
```

Uso della procedura in un programma: i valori da confrontare possono essere di tipo compatibile, perché sono passati per valore, ma il risultato deve per forza essere *int*.

```
#include <iostream>
using namespace std;

void cercaMax(int a, int b, int& max){
    if(a >= b)
        max = a;
    else
        max = b;
}//cercaMax

int main(void){
    short int primo, secondo;
    int massimo;
    cout << "Inserire due numeri" << endl;
    cin >> primo >> secondo;
    cercaMax(primo, secondo, massimo);
    cout << "Il massimo dei due numeri e' " << massimo << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

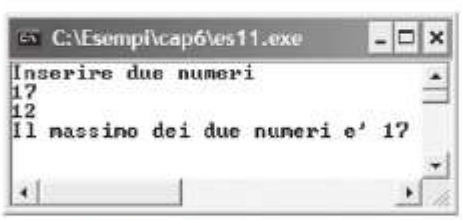


Figura 6.6 *Esecuzione dell'esempio*

Esempio 6.12

Procedura per il calcolo del risultato della divisione tra due numeri con controllo che il divisore non sia 0.

I	x, y	numerici	operandi della divisione
O	r	numerico	risultato della divisione, se *errore* ha valore *falso*
O	errore	booleano	ha valore *vero* se il divisore è 0 e la divisione non è valida

```
divisione(val x, val y, rif r, rif errore);
    errore ← falso;
    se y ≠ 0 allora
        r ← x / y
    altrimenti
        errore ← vero
    fine se
fine
```

```
void divisione(float x, float y, float& r, bool& errore){
    errore = false;
    if(y != 0)
        r = x / y;
    else
        errore = true;
}//divisione
```

La procedura *divisione* restituisce un risultato in *r* solo se non si è verificato un errore segnalato dalla variabile booleana *errore*.
Gli operandi della divisione sono passati per valore, mentre *r* ed *errore* sono passati per riferimento.

Uso della procedura in un programma:

```
#include <iostream>
#include <iomanip>
using namespace std;
void divisione(float x, float y, float& r, bool& errore){
    errore = false;
    if(y != 0)
        r = x / y;
    else
        errore = true;
}//divisione
int main(void){
    float a, b, c;
    bool errore;
    cout << "Inserire due numeri" << endl;
    cin >> a >> b;
    divisione(a, b, c, errore);
    if(!errore)
        cout << setw(4) << setprecision(4) << a
             << setw(4) << ":" << setw(4) << b
             << setw(4) << "=" << setw(6) << c << endl;
    else
        cout << "Impossibile dividere per 0" << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

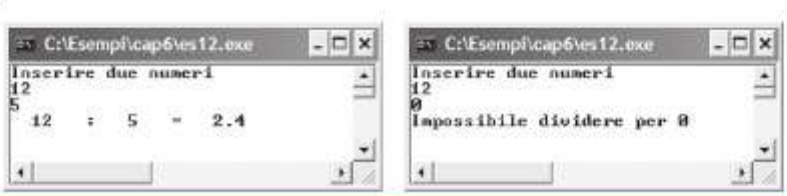


Figura 6.7 *Esecuzione dell'esempio*

Il passaggio per indirizzo

In C/C++ esiste un tipo di variabile chiamata **puntatore** il cui contenuto è l'indirizzo di memoria di un'altra variabile. Il tipo puntatore non è un tipo universale ma è dipendente dal tipo della variabile di cui contiene l'indirizzo.

Nota: I puntatori, gli operatori sui puntatori e gli ambiti d'uso sono spiegati nel capitolo 8.

Nel passaggio per indirizzo, viene utilizzata una variabile puntatore; il puntatore permette di modificare (definitivamente) il valore della variabile a cui si riferisce (in gergo, a cui punta).

Per dichiarare una variabile puntatore si usa il simbolo * tra il tipo e il nome della variabile.

```
tipoDominio *nomePuntatore;
```

definisce un puntatore che punta a variabili di tipo *tipoDominio*.

Nota: La presenza di spazi tra il tipo, l'asterisco e il puntatore è ininfluente.

Per assegnare a un puntatore l'indirizzo di una variabile si usa l'operatore **&** chiamato, in questo contesto, **operatore di indirizzo**.

```
tipoDominio *nomePuntatore;
tipoDominio nomeVar;
```

```
nomePuntatore = &nomeVar;
```
assegna al puntatore *nomePuntatore* l'indirizzo della variabile *nomeVar*.

Il puntatore contiene l'indirizzo della variabile e si può usare per fare riferimento alla variabile.

Per **riferirsi** al contenuto della variabile puntata, se *nomePuntatore* è il puntatore, si usa la notazione ***nomePuntatore***; in tal modo si sta **dereferenziando** il puntatore. In questo contesto il simbolo * prende il nome di **operatore di indirezione**.

In una **dichiarazione di funzione**, per dire che un **parametro è passato per indirizzo** lo si dichiara come **puntatore**; nel corpo della funzione, per accedere al contenuto della variabile si dereferenzia il puntatore con la notazione ***nomePuntatore***.

Nella chiamata della funzione, per indicare che il parametro è passato per indirizzo, bisogna passare proprio l'indirizzo del parametro con l'operatore &:

```
nomeProcedura(parXValore, &parXIndirizzo);
```

Nota: La leggibilità della chiamata, utilizzando il passaggio per indirizzo è massima: si notano subito quali parametri sono passati per valore e quali per indirizzo.

Esempio 6.13

La procedura *cercaMax* dell'esempio 6.11 può essere riscritta con la sintassi del passaggio dei parametri per indirizzo.

```
void cercaMax(int a, int b, int* max){
    if(a >= b)
        *max = a;
    else
        *max = b;
}//cercaMax
```

Il programma principale differisce dal precedente solo per la chiamata a funzione che diventa:

```
cercaMax(primo, secondo, &massimo);
```

Esempio 6.14

La procedura *divisione* dell'esempio 6.12 può essere riscritta con la sintassi del passaggio dei parametri per indirizzo.

```
void divisione(float x, float y, float* r, bool* errore){
    *errore = false;
    if(y != 0)
        *r = x / y;
    else
        *errore = true;
}//divisione
```

Il programma principale differisce dal precedente solo per la chiamata a funzione che diventa:

```
divisione(a, b, &c, &errore);
```

in cui è ben chiaro che *a* e *b* sono passati per valore mentre *c* ed *errore* per indirizzo; al contrario, osservando la chiamata relativa all'esempio con passaggio per riferimento:

```
divisione(a, b, c, errore);
```

nulla porta a capire come sono passati i parametri.

Confronto tra passaggio per riferimento e passaggio per indirizzo

Entrambi consentono la modifica dei valori.

	Svantaggi	Vantaggi
Passaggio per riferimento	■ l'individuazione della tipologia dei parametri avviene solo nella dichiarazione della funzione;	■ semplicità nel corpo della funzione: il parametro è sempre il nome di una variabile.
Passaggio per indirizzo	■ complessità delle notazioni nel corpo della funzione: a volte la leggibilità è minima.	■ l'individuazione della tipologia dei parametri avviene sia nella dichiarazione della funzione che nella chiamata; ■ possibilità di utilizzare tutti gli strumenti offerti dai puntatori.

6.6 FUNZIONI

Una **funzione** è un sottoprogramma che può avere una lista di parametri e che restituisce un valore, associato al nome della funzione stessa.
La funzione ha un **tipo** che è il tipo del valore restituito.

Le procedure che restituiscono un solo valore, cioè che hanno un solo parametro passato per riferimento, possono essere realizzate come funzioni.

Passare uno o più parametri della funzione per riferimento è contrario alla definizione di funzione, che dice che la funzione restituisce un unico risultato; in questo caso si avrebbero dei **side effect**.

Esempio 6.15

Funzione per la ricerca del massimo tra due numeri interi.

O	**max**	**intero**	massimo dei valori
I	a, b	interi	valori da confrontare

```
funzione max(val a, val b)
    se a >= b allora
        max ← a
    altrimenti
        max ← b
    fine se
fine
```

Le funzioni

C++ 6.4

La definizione di una funzione in C++ ha il formato:

```
tipoRisultato nomeFunzione(listaParametriFormali){
    istruzioni;
    return valoreDaRitornare;
}//nomeFunzione
```

Il valore restituito dalla funzione deve essere specificato in una istruzione ***return***.

Nota

Ci possono essere più istruzione *return* in corrispondenza di percorsi alternativi (come ad esempio i rami di una *if*).

Per **richiamare** una funzione si usa il **nome** della funzione seguito dalla lista dei parametri attuali.

```
nomeFunzione(listaParametriAttuali);
```

Esempio 6.16

Codifica dell'esempio 6.15.

```
int max(int a, int b){
    if(a >= b)
        return a;
    else
        return b;
}//max
```

Uso della funzione in un programma:

```
#include <iostream>
using namespace std;

int max(int a, int b){
```

```
    ...
}//max

int main(void){
    int primo, secondo, massimo;
    cout << "Inserire due numeri" << endl;
    cin >> primo >> secondo;
    massimo = max(primo, secondo);
    cout << "Il massimo dei due numeri e' " << massimo << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Il risultato della funzione viene assegnato a una variabile.

6.7 LA RICORSIONE

Un algoritmo è **ricorsivo** quando utilizza delle procedure ricorsive, cioè che al loro interno richiamano se stesse.

```
problema                    inizio
    ...                         problema
    problema                fine
    ...
fine
```

In pratica si può dire che, durante la descrizione di un procedimento, viene richiesta la ripetizione dell'intero procedimento.
L'algoritmo non deve però continuare a ripetere il procedimento all'infinito.

Le procedure ricorsive devono soddisfare delle **condizioni** che garantiscano la **fine** del procedimento:

- ogni chiamata successiva della procedura deve applicarsi ad un **sottoinsieme dei dati** di quella precedente o a **valori minori**;
- deve essere definita una condizione di terminazione delle ripetizioni (o di uscita), chiamata **base della ricorsione**.

Quando viene raggiunta la base della ricorsione vengono effettuati a ritroso tutti i calcoli rimasti in sospeso.

USO DI UNA PILA PER LA GESTIONE DELLE CHIAMATE A PROCEDURE RICORSIVE

Le chiamate ricorsive sono rese possibili dalla gestione dello **stack**, dato che ad ogni chiamata della procedura o funzione viene inserita nello stack una copia delle variabili locali, oltre all'indirizzo di rientro; ogni chiamata della procedura o funzione mantiene perciò le proprie variabili e il proprio ambiente.

Se si scrive un algoritmo ricorsivo in cui non è definita una condizione di terminazione si continua indefinitamente ad occupare spazio nello stack saturando la memoria.

Nota

Esempio 6.17

Stampa dei numeri da 0 a n.

```
#include <iostream>
using namespace std;

void stampa(int n){
    if(n > 0)
        stampa(n-1);
    cout << n << endl;
}//stampa

int main(void){
    int numero;
    cout << "Inserire il numero ";
    cin >> numero;
    stampa(numero);
    fflush(stdin);
    getchar();
    return 0;
}//main
```

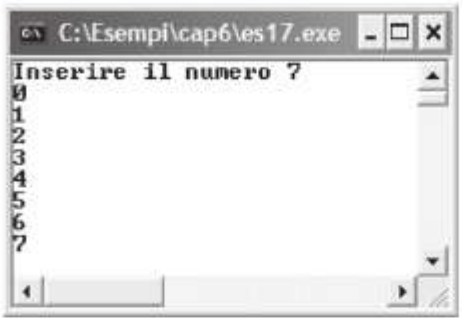


Figura 6.8 *Esecuzione dell'esempio*

Ad ogni chiamata la procedura si applica a un valore minore di *n*.

La condizione di terminazione della ricorsione è che il numero sia 0.

Quando si raggiunge la base della ricorsione, viene eseguita l'istruzione *cout << n << endl* in tutte le chiamate della procedura rimaste in sospeso, dall'ultima alla prima; il numero *n* è quello della chiamata della procedura.

Esempio 6.18

Stampa dei numeri da n a 0.

In questo caso nella procedura *stampa* l'istruzione *cout << n << endl* viene eseguita prima della chiamata ricorsiva.

```
#include <iostream>
using namespace std;

void stampa(int n){
    cout << n << endl;
    if(n > 0)
        stampa(n-1);
}//stampa

int main(void){
    int numero;
    cout << "Inserire il numero ";
    cin >> numero;
    stampa(numero);
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 6.9 *Esecuzione dell'esempio*

Esempio 6.19

Calcolo del fattoriale di un numero utilizzando una funzione ricorsiva.

La definizione ricorsiva di fattoriale è:

0! = 1	base della ricorsione;
n! = n * (n-1)!	chiamata ricorsiva: viene richiesto il calcolo del fattoriale del numero inferiore *n-1*.

Esempio: Calcolo di *fattoriale(4)*.

fattoriale(4) = 4 * fattoriale(3)	fattoriale(1) = 1 * 1 = 1
fattoriale(3) = 3 * fattoriale(2)	fattoriale(2) = 2 * 1 = 2
fattoriale(2) = 2 * fattoriale(1)	fattoriale(3) = 3 * 2 = 6
fattoriale(1) = 1 * fattoriale(0)	fattoriale(4) = 4 * 6 = 24

Si può vedere meglio che cosa succede inserendo delle istruzioni di stampa nella funzione *fattoriale*.

Figura 6.10 *Esecuzione dell'esempio*

```
#include <iostream>
#include <iomanip>
using namespace std;

long fattoriale(int n){
    cout << "n=" << setw(2) << n << endl;
    if(n == 0)
        return 1;
    else {
        long f = n * fattoriale(n-1);
        cout << "n=" << setw(2) << n
             << " - fattoriale(" << n << ")="
             << setw(3) << f << endl;
        return f;
    }//else
}//fattoriale

int main(void){
    int limite;
    long risultato;
    cout << "Inserire il valore di n in n! ";
    cin >> limite;
    risultato = fattoriale(limite);
    cout << "Il fattoriale di " << limite << " e' "
         << risultato << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Ricorsione e iterazione

Non tutti i linguaggi di programmazione permettono l'uso della ricorsione; in ogni caso una **struttura ricorsiva** può sempre essere descritta con un **procedimento iterativo** (che utilizza le strutture di controllo cicliche).
La descrizione del procedimento iterativo è spesso più complicata di quella del procedimento ricorsivo, che presenta una maggior semplicità di formulazione.

Al contrario però, verificare che l'algoritmo è corretto risulta più semplice nella versione iterativa che non nella versione ricorsiva, infatti in un algoritmo ricorsivo le operazioni da compiere sono descritte in modo implicito e vengono eseguite nell'ordine inverso a quello descritto.

I due esempi di stampa dei numeri equivalgono a:

```
per i da 1 a n
    scrivi i
fine per

per i da n a 1 con passo -1
    scrivi i
fine per
```

La definizione iterativa di fattoriale è:

```
fattoriale ← 1
i ← 1
mentre i <= n esegui
    fattoriale ← i * fattoriale
    i ← i + 1
fine mentre
```

Esempio 6.20

La torre di Hanoi: ci sono tre aste, una di partenza (1), una intermedia (2) e una di arrivo (3) e *n* dischi forati di misura decrescente infilati nell'asta di partenza. Bisogna spostare i dischi nell'asta di arrivo (usando l'asta intermedia), tenendo presente che si può spostare un solo disco alla volta e che non si può appoggiare un disco più grande su uno più piccolo.

Questo problema si può risolvere facilmente in modo ricorsivo, infatti, se a un certo punto *n-1* dischi si trovano sull'asta intermedia, è facile spostare il disco più grande dall'asta di partenza all'asta di arrivo.
Per risolvere il problema si può quindi procedere in questo modo:

- spostare *n-1* dischi dall'asta di partenza all'asta intermedia,
- spostare un disco dall'asta di partenza all'asta di arrivo,
- spostare *n-1* dischi dall'asta intermedia all'asta di arrivo.

```
#include <iostream>
using namespace std;

void torre(short n, short partenza, short intermedia, short
arrivo){
    if(n == 1)
        cout << "Sposta un disco da " << partenza << " a " <<
arrivo << endl;
    else {
        torre(n-1, partenza, arrivo, intermedia);
        torre(1, partenza, intermedia, arrivo);
        torre(n-1, intermedia, partenza, arrivo);
    }//else
}//torre

int main(void){
    int numero;
    cout << "Inserire il numero di
         dischi ";
    cin >> numero;
    torre(numero, 1, 2, 3);
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 6.11 *Esecuzione dell'esempio*

Ricorsione indiretta (o mutua)

La ricorsione in un programma può essere semplice, se una procedura richiama se stessa direttamente, o **indiretta** (o **mutua**) se la procedura richiama un'altra procedura che a sua volta richiama la prima.

La procedura *A* richiama la procedura *B* e la procedura *B* richiama la procedura *A*.

```
proceduraA                proceduraB                inizio
    ...                       ...                       proceduraA
    proceduraB                proceduraA                fine
    ...                       ...
fine                      fine
```

C++ 6.5

Prototipo di funzione

Se la procedura *A* richiama la procedura *B* e la procedura *B* richiama la procedura *A*, quale deve essere definita per prima?
Per supportare la mutua ricorsione il C++ mette a disposizione il concetto di **prototipo** di funzione.
Il **prototipo di una funzione** consiste nella scrittura della sola dichiarazione della funzione, senza il corpo, dopo le direttive *#include*.
Nella scrittura del prototipo possono essere omessi i nomi dei parametri perché al compilatore interessa solo controllare la corrispondenza dei tipi.

```
int max(int, int);
```
prototipo equivalente a
```
int max(int a, int b);
```

Nota

Dopo aver definito i prototipi, l'implementazione delle funzioni può avvenire in qualsiasi ordine, quindi è sempre utile usare i prototipi, in qualsiasi programma.

Esempio 6.21

Stabilire se un numero è pari. Si possono usare due funzioni *pari* e *dispari* mutuamente ricorsive.

Opportune istruzioni di stampa all'interno delle funzioni permettono di capire meglio che cosa succede.

```
#include <iostream>
using namespace std;

bool pari(long n);       //prototipo della funzione pari

bool dispari(long n);    //prototipo della funzione dispari

bool dispari(long n){
    cout << "dispari(" << n << ")" << endl;
    if(n == 1)
        return true;
    else
        return pari(n-1);
}//dispari

bool pari(long n){
    cout << "pari(" << n << ")" << endl;
```

```
    if(n == 1)
        return false;
    else
        return dispari(n-1);
}//pari

int main(void){
    int numero;
    cout << "Inserire il numero ";
    cin >> numero;
    if(pari(numero))
        cout << numero << " e' pari" << endl;
    else
        cout << numero << " e' dispari" << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 6.12 *Esecuzione dell'esempio*

PUNTATORE A FUNZIONE

C++ 6.6

Il nome di una funzione corrisponde al suo indirizzo di memoria.

Essendo un indirizzo, il nome di una funzione può essere memorizzato in un opportuno puntatore.

L'istruzione

```
typedef tipo1 (*tipoPuntFunzione) (tipo2 primoP, ..., tipoN nP);
```

definisce il tipo puntatore a tutte le funzioni che hanno gli stessi tipi di parametri e lo stesso tipo di ritorno indicati.

Si può dichiarare una variabile di tale tipo e assegnarle un valore con la notazione:

```
tipoPuntFunzione nomePuntFunz = nomeFunzione;
```

nomeFunzione è una funzione che rispetta i vincoli sopra esposti.

Nota

La funzione può essere richiamata anche con il puntatore:

```
tipo1 nomeVar = nomePuntFunz(primoP, ..., nP);
```

In questo modo diventa possibile passare come parametro di una funzione, un'altra funzione.

ESEMPIO 6.22

Scelta della funzione da applicare, per il calcolo del massimo o del minimo.

La funzione *applica* ha come terzo parametro un puntatore a funzione con il quale richiama nel proprio corpo la funzione desiderata.

Nota

```
#include <iostream>
using namespace std;

typedef int (*funzione) (int, int);
```

```
int max(int, int);
int min(int, int);
int applica(int, int, funzione);

int main(void){
    int primo, secondo, massimo, minimo;
    cout << "Inserire due numeri" << endl;
    cin >> primo >> secondo;
    massimo = applica(primo, secondo, max);
    minimo = applica(primo, secondo, min);
    cout << "Il massimo e' " << massimo << endl;
    cout << "Il minimo e' " << minimo << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main

int applica(int x, int y, funzione f){
    return f(x, y);
}//applica

int max(int a, int b){
    if(a >= b)
        return a;
    else
        return b;
}//max

int min(int a, int b){
    if(a <= b)
        return a;
    else
        return b;
}//min
```

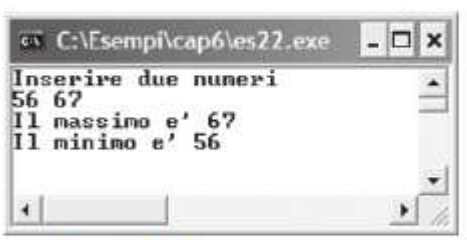


Figura 6.13 *Esecuzione dell'esempio*

C++ 6.7 Funzioni inline

Il C++ permette di definire una particolare tipologia di funzioni, le funzioni ***inline***.

Il compilatore, quando incontra una funzione definita *inline*, in sostanza sostituisce all'istruzione della chiamata l'intero corpo della funzione.

Questo comporta un allungamento del programma ma un certo risparmio di tempo perché non entrano in gioco i meccanismi del passaggio dei parametri e del salvataggio (e recupero) nello stack dell'indirizzo dell'istruzione successiva a quella della chiamata, tipici delle consuete funzioni e procedure.

Una **funzione inline** si definisce semplicemente scrivendo la parola riservata *inline* prima del nome della funzione nella dichiarazione della stessa; il corpo della funzione e la chiamata mantengono le usuali notazioni.

La funzione *max* dell'esempio 6.16 resa *inline* diventa:

```
int inline max(int a, int b){
    if(a >= b)
```

```
        return a;
    else
        return b;
}//max
```

Nota

Le funzioni *inline* trovano utilizzo nei file di libreria. Nelle classi della libreria STL le funzioni o i metodi che sovraccaricano vari operatori (come ++, --, []) sono offerti anche *inline*.

MACRO

C++ 6.8

La direttiva #***define*** permette non solo la definizione di costanti ma anche la definizione di macro.

Una **macro** è un insieme di istruzioni identificate da un'etichetta; la macro viene richiamata specificandone il nome e i parametri tra parentesi, in modo analogo a una funzione; durante la compilazione, ad ogni occorrenza dell'etichetta viene sostituito il relativo codice.

```
#define nomeMacro(listaParametri) corpoDellaMacro
```

Al contrario di quello che avviene nelle funzioni *inline* non c'è nessun controllo sui tipi dei parametri della macro.

Nota

Le macro trovano utilizzo nei file di libreria nonostante la crescente preferenza all'uso delle funzioni *inline*. Spesso i nomi delle macro sono in maiuscolo, anche se le specifiche ISO C++ non hanno confermato tale convenzione.

ESEMPIO 6.23

Calcolo del massimo tra due interi mediante una macro.

```
#include <iostream>
using namespace std;

#define MAX(a, b) ((a) > (b)) ? (a) : (b)

int main(void){
    int primo, secondo, massimo;
    cout << "Inserire due numeri" << endl;
    cin >> primo >> secondo;
    massimo = MAX(primo, secondo);
    cout << "Il massimo e' " << massimo << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

FUNZIONI CON NUMERO VARIABILE DI PARAMETRI

C++ 6.9

È possibile dichiarare funzioni che possono ricevere un numero variabile di parametri.

La funzione deve avere almeno un parametro fisso; tre puntini (...) al posto dell'ultimo parametro indicano che si può passare un numero variabile di parametri, di qualsiasi tipo.

È necessario includere il file di intestazione ***<cstdarg>*** contenente le definizioni del tipo ***va_list*** e delle macro ***va_start***, ***va_arg*** e ***va_end***.

Il tipo *va_list* definisce un puntatore ai valori dei parametri variabili; invocando la macro *va_start* si inizializza il puntatore al primo dei parametri variabili.

La scansione avviene invocando la macro *va_arg* che necessita del tipo del parametro, incrementa il puntatore e restituisce il valore del corrente parametro.

Una volta terminata l'elaborazione dei parametri, attraverso la macro *va_end* si rilasciano le risorse occupate.

Esempio 6.24

Funzione che stampa i valori che gli vengono passati, qualsiasi sia il loro numero. Il numero dei dati è il primo parametro della funzione.

```
#include <iostream>
#include <cstdarg>
using namespace std;

void parametri(int, ...);

int main(void){
    parametri(3, 2356, 345, 3467);
    parametri(5, 56, 3425, 3467, -3421, 2345);
    parametri(1, 356);
    parametri(0);
    getchar();
    return 0;
}//main

void parametri(int nV, ...){
    long v;
    va_list pPar;
    va_start(pPar, nV);
    for(int i = 0; i < nV; i++){
        v = va_arg(pPar, long);
        cout << v << " ";
    }//for
    cout << endl;
    va_end(pPar);
}//parametri
```

Figura 6.14 *Esecuzione dell'esempio*

C++ 6.10 Programmi con più file sorgenti

Un programma in C++ può essere composto da più file sorgenti separati.
È possibile dichiarare variabili globali la cui visibilità è estesa a più file.
Per questo è necessario:

- definire (e inizializzare) la variabile in un solo file sorgente, con l'usuale notazione;
- premettere lo specificatore ***extern*** prima della dichiarazione della variabile in tutti gli altri file in cui deve essere visibile.

È possibile limitare la visibilità di una variabile globale al solo file in cui è definita usando lo specificatore *static* (negli altri file sorgente non è possibile usare per tale variabile lo specificatore *extern*).

Compilazione con file separati

Con il comando *Esegui*, *Compila* dell'ambiente wxDev-C++ il programma viene sia compilato che linkato.

Nota

L'ambiente wxDev-C++ è configurato in modo da avere sempre la funzione *main()* all'interno del file del codice.

Per compilare più file sorgenti e poi linkarli insieme conviene richiamare il compilatore *g++* da linea di comando.

```
g++ -c nomeFile.cpp
g++ -c fileMain.cpp
```

genera i due file oggetto *nomeFile.o* e *fileMain.o*

```
g++ -o nomeEseguibile.exe fileMain.o nomeFile.o
```

linka il tutto generando l'eseguibile *nomeEseguibile*.

File header personali

C++ 6.11

In C/C++ un file di libreria è chiamato anche file di intestazione (file **header**) perché solitamente ha l'estensione **.h** e viene incluso mediante la direttiva ***#include***. Con le specifiche ISO C++ è scomparsa l'estensione (si pensi a **<iostream>**), tuttavia i file continuano a chiamarsi header.

È possibile creare un file header personale e salvarlo nella stessa directory del programma che deve includerlo oppure nella directory predefinita dei file di libreria.

Per includere un file header personale salvato nella directory predefinita si usa la sintassi:

```
#include <nomeFile>
```

mentre per includerne uno salvato nella medesima directory del programma che lo usa si scrive:

```
#include "nomeFile"
```

Il file header può contenere riferimenti ad altri file header, a namespace, e soprattutto la definizione (con relativa implementazione) di funzioni personali.

Esempio 6.25

Creazione di un file header e sua inclusione.

```
//es25h
int max(int a, int b){
    if(a >= b)
        return a;
    else
        return b;
}//max
```

```
//es25.cpp
#include <iostream>
#include "es25h"
using namespace std;
```

```
int main(void){
    int primo, secondo, massimo;
    cout << "Inserire due numeri" << endl;
    cin >> primo >> secondo;
    massimo = max(primo, secondo);
    cout << "Il massimo dei due numeri e' " << massimo << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

C++ 6.12

NAMESPACE PERSONALI

Le specifiche ISO C++ hanno potenziato il concetto di **namespace** e ridotto quello di file header. Se il programmatore ha bisogno di funzioni personali è conveniente la creazione di un proprio *namespace* in cui definire (dichiarare e implementare) le funzioni.
Il *namespace* creato può essere salvato nella stessa directory del file che lo deve usare oppure nella directory predefinita dei file di libreria,

La struttura di un ***namespace*** è:

```
namespace nomeNamespace{
    // definizione dei campi
    // dichiarazione delle funzioni
}//nomeNameSpace
// implementazione delle funzioni
```

L'implementazione di ciascuna funzione ha il formato:

```
tipoValoreRitornato nomeNamespace::nomeFunzione(listaParametri){
    istruzioni
}//nomeNamespace::nomeFunzione
```

L'operatore **::** chiamato **operatore di risoluzione della visibilità** (**scope resolution operator**) stabilisce l'appartenenza della funzione al namespace.

Per indicare in un programma che si utilizza un namespace si usa la clausola ***using***.

```
using namespace nomeNamespace;
```

La sintassi della chiamata di una funzione di namespace specificato nella clausola using resta identica a quella delle chiamate a funzioni del programma stesso. La clausola ***using*** nasconde i dettagli di implementazione del *namespace*.

Nota

Un namespace può essere incluso anche con la direttiva #***include***.
L'inclusione di un namespace salvato nella directory predefinita ha il formato:

```
#include <nomeNamespace>
```

mentre per un namespace salvato nella medesima directory del programma che lo usa si scrive:

```
#include "nomeNamespace"
```

Se un namespace è incluso con la direttiva #*include* però, la chiamata di una funzione del namespace deve specificare esplicitamente il nome del namespace:

```
nomeNamespace::nomeFunzione(listaParametri)
```

ESEMPIO 6.26

Creazione e uso di un namespace.

```
//es26ns
namespace Massimo{
    int a, b;                   //definizione campi
    int max(int a, int b);      //definizione funzioni
}//Massimo

//implementazione funzioni
int Massimo::max(int a, int b){
    if(a >= b)
        return a;
    else
        return b;
}//Massimo::max
```

```
//es26.cpp
#include <iostream>
#include "es26ns"
using namespace std;
using namespace Massimo;

int main(void){
    int primo, secondo, massimo;
    cout << "Inserire due numeri" << endl;
    cin >> primo >> secondo;
    massimo = max(primo, secondo);
    cout << "Il massimo dei due numeri e' " << massimo << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

INTRODUZIONE ALLA GESTIONE DELLE ECCEZIONI

C++ 6.13

In C++ gli **errori** sono gestiti con il meccanismo della **gestione delle eccezioni**.

Quando si verifica un errore di runtime (cioè a tempo di esecuzione) il sistema genera (o solleva) automaticamente un'eccezione.
Anche il programmatore può generare un'eccezione per segnalare un errore o una situazione anomala.

Nota

Il meccanismo della gestione delle eccezioni è valido sia per le funzioni che per i metodi di una classe.

Un'**eccezione** può essere:

- un valore di un **tipo predefinito**; il tipo può essere indicato nella dichiarazione della funzione e viene utilizzato per riconoscere l'eccezione stessa;
- un oggetto che viene creato e lanciato nel codice di una funzione o di un metodo con l'istruzione *throw*; il tipo dell'oggetto deve essere una classe derivata della classe ***exception*** definita nel file di libreria ***<exception>*** (da includere).

Le funzioni o i metodi che generano eccezioni possono indicarle nella loro dichiarazione con la clausola ***throw****(listaEccezioni)*.

Nota

L'indicazione non è obbligatoria ma è consigliabile perché aumenta la leggibilità del metodo.

È possibile impedire che una funzione o un metodo lancino eccezioni indicando la clausola *throw()* senza nulla tra parentesi.

Se le eccezioni non vengono gestite, quando viene sollevata un'eccezione viene generato un errore di sistema.

Il modo più semplice per gestire le eccezioni è intercettarle inserendo le chiamate ai metodi che possono generarle in un blocco *try* e usando poi una o più clausole *catch* per specificare cosa fare quando si verifica una determinata eccezione.
Un blocco ***try catch*** è chiamato anche **gestore di eccezioni**.

Le eccezioni come tipi

Una funzione che lancia una o più eccezioni di un **tipo predefinito** segue lo schema:

```
tipoRitorno nomeFunzione(listaParametri) throw(tipo1, tipo2, ...){
    //istruzioni
    if ... {
        //istruzioni
        throw valoreDiTipo1;
    }
    //istruzioni
    throw valoreDiTipo2;
    //istruzioni e altri throw
}//nomeFunzione
```

Il blocco ***try catch*** che gestisce le eccezioni di un tipo predefinito segue lo schema:

```
try{
    //istruzioni
    //richiamo di una funzione che lancia eccezioni
    //altre istruzioni
}catch(tipo1 varTipo1){
    //istruzioni per la gestione dell'eccezione
}catch(tipo2 varTipo2){
    //istruzioni per la gestione dell'eccezione
}...//altri catch
```

Nota

Le variabili del tipo dell'eccezione (come *varTipo1*, *varTipo2*, ...), se non vengono usate all'interno del blocco *catch* corrispondente, possono essere omesse.

Esempio 6.27

Calcolo del reciproco di un numero.
Un tentativo di divisione per zero viene gestito con un'eccezione di tipo *int*.

```
#include <iostream>
using namespace std;
```

```
double reciproco(int n) throw (int){
    if(n == 0)
        throw 0;
    return 1.0/n;
}//reciproco

int main(void){
    int numero;
    double r;
    cout << "Inserisci un numero intero diverso da 0\n";
    cin >> numero;
    try{
        r = reciproco(numero);
    }catch(int){
        cerr << "E' richiesto un numero diverso da 0!\n";
        return 1;
    }//catch
    cout << "Il reciproco di " << numero
         << " e' " << r << endl;
    return 0;
}//main
```

Figura 6.15 *Esecuzione dell'esempio*

Se si verifica l'eccezione viene stampato un messaggio e il programma termina con valore di ritorno 1 per attestare che l'esecuzione non si è conclusa normalmente. Se non viene generata l'eccezione vengono eseguite normalmente le istruzioni successive al blocco *catch*.

VERIFICA

QUESTIONARIO

1. Che cosa si intende per metodologia top down e quali sono i suoi vantaggi?
2. Quali sono i vantaggi dell'uso di moduli nella scrittura del codice?
3. Come funzionano le chiamate a sottoprogrammi?
4. Come conviene scomporre un problema?
5. Qual è la differenza tra variabili globali e variabili locali?
6. Che cosa si intende per ambito di visibilità o scope?
7. Che cosa sono i parametri?
8. Come funziona il passaggio dei parametri per valore?
9. Come funziona il passaggio dei parametri per riferimento?
10. Che cos'è una funzione?
11. Che cos'è la ricorsione?
12. Qual è la relazione tra ricorsione e iterazione?
13. Che cosa si intende per ricorsione indiretta o mutua?
14. Che cosa sono le eccezioni e come possono essere gestite?

TEST

1 ..... *Indicare se le seguenti affermazioni sono vere o false.*

V	F	
☐	☐	1) L'indirizzo di rientro di procedure e funzioni viene memorizzato nello stack.
☐	☐	2) Le variabili globali sono memorizzate nello stack.
☐	☐	3) Le variabili locali a procedure e funzioni sono memorizzate nello stack.
☐	☐	4) I parametri di procedure e funzioni sono memorizzati nello stack.
☐	☐	5) I parametri passati per riferimento occupano più spazio di quelli passati per valore.

2 ..... *Individuare la risposta esatta.*

1) Qual è l'output del programma seguente?

```
short prova(int x, int& y){
    x = x + y;
    y = x * 4 / 2;
    return x;
}//prova
int main(void){
    short r;
    int m = 17, g = 5;
    r = prova(m, g);
    cout << m << " " << g << " " << r;
    getchar();
    return 0;
}//main
```

☐ 17 5 22 ☐ 22 44 22
☐ 17 44 22 ☐ 22 5 17

2) Quali coppie di valori stampa il programma seguente?

```
void passa(int r, int& t){
    r = r + 1;
    t = t + 1;
    cout << r+t << " " << t-r;
}//passa
int main(void){
    int m=1, g=1;
    passa(m, g);
    cout << "  " << m-g << " " << g+m;
    getchar();
    return 0;
}//main
```

☐ 4 0 -1 3 ☐ 4 0 0 4
☐ 0 4 3 -1 ☐ 3 -1 -1 3

3) Osservando la funzione *ricorsiva* che cosa si può dire?

```
long ricorsiva(int y){
   if(y == 1)
      retrun 1;
   else
      return y * ricorsiva(y+1);
}//ricorsiva
```

- ☐ la funzione calcola il fattoriale di y;
- ☐ la funzione non ha una condizione di uscita;
- ☐ non si raggiunge mai la condizione di uscita;
- ☐ ci sono errori sintattici.

4) Quale serie di valori produce la chiamata *cosaFa(20)*?

```
void cosaFa(short n){
    if(n > 0)
        cosaFa(n / 2);
    cout << n << " ";
}//cosaFa
```

- ☐ 20 10 5 2 1 0
- ☐ 20 10 5 2 1
- ☐ 0 1 2 5 10
- ☐ 0 1 2 5 10 20

3 ..... Per *a* intero e *b* reale, opportunamente inizializzate, dati i seguenti prototipi, individuare tutte le chiamate che producono un errore di compilazione oppure un avvertimento (warning):

1) `void proc3(int x);`
- ☐ `proc1(a);`
- ☐ `proc1(b);`
- ☐ `proc1(1);`

2) `void proc2(double x);`
- ☐ `proc2(a);`
- ☐ `proc2(b);`
- ☐ `proc2(1);`

3) `void proc3(int& x);`
- ☐ `proc3(a);`
- ☐ `proc3(b);`
- ☐ `proc3(1);`

4) `void proc4(double& x);`
- ☐ `proc4(a);`
- ☐ `proc4(b);`
- ☐ `proc4(1);`

ESERCIZI

1 ..... Che cosa produce in output il programma seguente:

```
int main(void){
    long a = 2, b = 3, r;
    r = modifica(a, b);
    cout << a << " " << b << " " << r;
    getchar();
    return 0;
}//main
```

- ■ se la funzione *modifica* è definita come segue?

```
long modifica(long x, long y){
    long m = x * y;
```

```
    x = x * y;
    y = x * y;
    return m;
}//modifica
```

e se i parametri vengono modificati nel modo seguente?

- `long modifica(long& x, long y);`
- `long modifica(long x, long& y);`
- `long modifica(long& x, long& y);`

2 ..... Dato il seguente frammento di programma:

```
bool ping(int t){
    return !(t > 0);
}//modifica
void pong(int& t){
if(ping(t))
    cout << "Niente da fare\n";
else
    t = t - 1;
}//pong
```

trasformare la procedura *pong* in una funzione equivalente che non scriva nulla a video e che ritorni il valore negato di *ping(t)*.

PROPOSTE DI LAVORO

1 ..... Eseguire le seguenti operazioni usando opportune procedure e funzioni:

- somma addizionando una unità alla volta;
- moltiplicazione per addizioni successive (del tipo precedente);
- potenza per moltiplicazioni successive (del tipo precedente);
- sottrazione sottraendo una unità alla volta;
- divisione per sottrazioni successive (del tipo precedente).

2 ..... Eseguire le seguenti operazioni usando opportune procedure e funzioni:

- calcolare la potenza di *x* alla *y*;
- calcolare le potenze da *m* a *n* di un numero;
- calcolare le potenze da *m* a *n* di ciascun numero intero compreso tra *s* e *t*;
- sommare le potenze di 2 con esponente da *m* a *n*;
- sommare le potenze di 2 con esponente da 0 a *n*.

3 ..... Eseguire le seguenti operazioni usando opportune procedure e funzioni:

- determinare se un numero è perfetto (un numero è perfetto se è uguale alla somma dei suoi divisori tranne se stesso);
- stampare i primi *n* numeri perfetti;
- trovare tutti i numeri perfetti minori di *n*;
- data una serie di numeri, per ognuno stabilire se è perfetto o no.

4 ..... Eseguire le seguenti operazioni usando opportune procedure e funzioni:

- stabilire se un numero è primo (è primo se non ha divisori tranne 1 e se stesso);
- calcolare i primi *n* numeri primi;
- trovare tutti i numeri primi minori di un valore dato;
- calcolare il numero primo immediatamente superiore a un numero dato;
- scrivere gli *m* numeri primi immediatamente superiori a *n*;
- dato un numero, determinare il numero primo più vicino;

- trovare i numeri primi gemelli minori di un valore inserito (due numeri primi si dicono gemelli se la loro differenza è 2);
- trovare le prime *n* coppie di numeri primi gemelli.

5 ..... Realizzare una serie di procedure e funzioni per la gestione di un testo:
- cercare se una parola è presente in un testo;
- contare quante volte una parola è presente in un testo;
- sostituire una parola con un'altra in un testo (solo la prima occorrenza);
- sostituire una parola con un'altra in un testo (per tutte le occorrenze);
- eliminare la prima occorrenza di una parola in un testo;
- eliminare tutte le occorrenze di una parola in un testo.

6 ..... Realizzare una serie di procedure e funzioni per la gestione delle date:
- controllare la correttezza di una data;
- calcolare quanti giorni sono trascorsi dall'inizio dell'anno a una data inserita;
- calcolare quanti giorni intercorrono tra due date;
- calcolare la data del giorno precedente e del giorno successivo a una data;
- dire a quale numero ordinale da 1 a 366 corrisponde una data.

7 ..... Realizzare una serie di procedure e funzioni per la gestione delle frazioni:
- semplificare una frazione o dire se è irriducibile;
- date due frazioni, ridurle al medesimo denominatore;
- eseguire le operazioni aritmetiche tra le frazioni.

8 ..... Realizzare un file di intestazione per rendere disponibili in un programma le procedure e funzioni di uno dei gruppi precedenti.

9 ..... Realizzare un namespace per rendere disponibili in un programma le procedure e funzioni di uno dei gruppi precedenti.

Algoritmi ricorsivi

10 ..... Convertire un numero da decimale a binario e viceversa usando procedimenti ricorsivi.

11 ..... Invertire una stringa.

12 ..... Eseguire la moltiplicazione di *a*b* in modo ricorsivo:
a*b = 0 se b= 0
a*b = a*(b-1) se b>0

13 ..... Calcolare la potenza di *x* alla *n* in modo ricorsivo:
potenza (x, 0) = 1
potenza (x, n) = x * potenza(x, (n-1))

14 ..... Eseguire la somma dei numeri da 1 a *n* in modo ricorsivo:
somma(1) =1
somma(n) = n + somma(n-1)

15 ..... Calcolare il quadrato di un numero intero positivo sapendo che è dato dalla somma dei primi *n* numeri dispari. (Esempio: 25 = 1 + 3 + 5 + 7 + 9)
(Suggerimento: l'*n*-esimo numero dispari è 2**n*-1
Somma dei primi *n* numeri dispari, posto *d = 2*n-1*:
somma(1) = 1
somma(d) = d + somma(d-2)
Dare anche una soluzione iterativa.

16 ..... Calcolare l'*n*-esimo numero di Fibonacci. Il valore di un termine viene calcolato sommando i valori dei due termini precedenti.
fib(0) = 1 fib(1) = 1
fib(n) = fib(n - 1) + fib(n - 2)

Dare anche una soluzione iterativa.
In questo caso la soluzione iterativa è preferibile perché la soluzione ricorsiva è molto inefficiente (ci sono due chiamate ricorsive per il calcolo di ogni termine).

17 ..... Calcolare il Massimo Comun Divisore di due numeri sapendo che *MCD(a, b) = MCD(a - b, b)* per *a > b* e *MCD(a, b) = MCD(a, b - a)* per *a < b*.

18 ..... Trovare i possibili anagrammi di una parola di *n* lettere.

19 ..... Data una espressione in notazione polacca prefissa trasformarla in notazione normale (infissa) con parentesi.

Suggerimenti per progetti

Nota

Seguono alcuni suggerimenti di problemi da analizzare per realizzare un progetto mediante l'analisi top down. Per la soluzione possono essere necessarie le strutture di dati spiegate nei capitoli 7 e 8 e i file spiegati nel capitolo 9.

20 ..... Indovinare un numero casuale.
Il programma deve generare un numero casuale. L'utente deve tentare di indovinare il numero mentre il programma risponde con le frasi "Il numero da indovinare è più grande", "Il numero da indovinare è più piccolo", oppure "Hai indovinato" e indica il numero di tentativi effettuati.
Si possono stabilire dei livelli di difficoltà impostando all'inizio l'intervallo in cui può essere scelto il numero e il numero di tentativi che si possono fare, dopo i quali il computer viene dichiarato vincitore.

21 ..... Realizzare un editor di testo che permetta di lavorare su un testo riga per riga, con operazioni come: cancellare una riga, inserire una nuova riga, modificare una riga, salvare il testo, modificare un testo esistente ecc.

22 ..... Realizzare un programma di autoapprendimento che permetta di navigare tra vari menu per scegliere le spiegazioni relative agli argomenti predisposti desiderati.

23 ..... Realizzare un programma per gestire la rappresentazione di un dato (per esempio numerico intero con varie dimensioni di rappresentazioni, reale float o double, alfanumerico in codice Ascii o Unicode ecc.), indicando il tipo di rappresentazione desiderato e restituendo la sequenza di bit corrispondente. Viceversa, data una sequenza di bit interpretarla nel modo desiderato. Gestire anche la possibilità di effettuare operazioni sulle rappresentazioni (per esempio le operazioni aritmetiche sulle rappresentazioni dei numeri, operazioni di confronto, calcolo del precedente e del successivo ecc.).

24 ..... Controllare se un documento XHTML è scritto correttamente, con tutti i tag chiusi, nidificati correttamente, gli attributi tra virgolette ecc.

25 ..... Realizzare un programma che permetta di definire una grammatica indicando simboli terminali e non terminali e regole di riscrittura, come spiegato nel capitolo 1. Il programma deve poi permettere di inserire una frase e di verificare se la frase appartiene al linguaggio o di indicare il primo errore.

26 ..... Realizzare un programma per giocare a carte contro il computer; il valore più alto vince; se i valori sono uguali vince il rosso, se sono uguali e dello stesso colore, fiori vincono su picche e cuori su quadri. Gestire opportunamente il mazzo di carte con operazioni come inizializzare il mazzo in modo ordinato, mescolare il mazzo, estrarre la prima carta dal mazzo, rimettere una carta in fondo al mazzo.

27 ..... Realizzare un programma che riesca a simulare una conversazione, definendo alcune frasi possibili, possibili risposte alternative, frasi generiche da utilizzare in caso non ci siano altre soluzioni ecc.

STRUTTURE DI DATI

7

PREREQUISITI

Saper descrivere algoritmi e realizzare programmi in C++ che usano:

- strutture di controllo
- procedure e funzioni

OBIETTIVI

CONOSCENZE

- Strutture di dati
- Array
- Algoritmi di ricerca e ordinamento sugli array
- Matrici
- Record
- Tabelle

ABILITÀ/CAPACITÀ

- Descrivere algoritmi che utilizzano strutture di dati
- Scrivere programmi in C++ che usano array e record

7.1 LE STRUTTURE DI DATI

Una **struttura di dati** è un raggruppamento di dati, con una certa organizzazione, che si può considerare come un unico oggetto o come composto dai singoli dati; è possibile quindi riferirsi all'intera struttura di dati o operare su ciascun singolo dato.

Al programmatore interessano le strutture di dati informative o astratte.

Si dice struttura di dati **informativa** o **astratta** una struttura definita da un punto di vista logico, descrivendo cioè soltanto le associazioni logiche tra i dati e i metodi per utilizzare la struttura.

Il programmatore sceglie la struttura astratta più adatta a memorizzare le informazioni del problema.

Esempi di strutture di dati astratte sono array, matrici, liste, ecc.

Le **caratteristiche** principali che differenziano le strutture di dati astratte sono:

- la possibilità di cambiare o meno dimensione durante l'esecuzione (**dinamica** o **statica**);
- il fatto che i dati siano tutti dello stesso tipo oppure no (**omogenea** o **eterogenea**);
- il fatto che sia possibile accedere direttamente a un elemento, o che sia invece necessario scorrere tutti gli elementi precedenti (**accesso diretto** o **sequenziale**).

Per poter utilizzare una struttura di dati in un programma si deve poter memorizzare e gestire la struttura nella **memoria** del computer.

Si dice struttura di dati **concreta** la rappresentazione nella memoria del computer di una struttura astratta.

La caratteristica principale che differenzia le strutture di dati concrete è il fatto che i dati siano memorizzati in memoria in **locazioni contigue** oppure no (sequenziale o no).

Le strutture concrete dipendono dal linguaggio di programmazione e da come il compilatore organizza le informazioni in memoria.

Le strutture di dati sono memorizzate nella **memoria RAM** ed esistono soltanto all'interno del programma che le utilizza; quando il programma termina, i dati inseriti nella struttura non sono più utilizzabili; non è possibile conservare dati in queste strutture, nè usarle per comunicare dati tra un programma e un altro.

Per poter conservare delle informazioni in modo permanente bisogna memorizzarle in **file**.

Nota — I file sono memorizzati su memoria di massa ed esistono indipendentemente dall'esecuzione del programma.

C++ 7.1 TIPI STRUTTURATI

I **tipi strutturati** sono tipi che contengono più valori in una variabile.

I principali tipi strutturati in C++ sono array, strutture, oggetti e file.

Nota — I file sono spiegati nel capitolo 9, gli oggetti nel capitolo 10.

7.2 GLI ARRAY

Un **array** è un insieme di valori tutti dello stesso tipo, cioè **omogenei** tra loro.

Nota

Gli array sono chiamati comunemente anche vettori ma, poiché in alcuni linguaggi (come Java) array e vettori non sono esattamente la stessa cosa, è preferibile usare il termine array.

Gli array permettono di risolvere problemi in cui bisogna operare ripetutamente su dati dello stesso tipo, o effettuare ricerche o ordinamenti su gruppi di dati.

Si può pensare a un array come a una serie di caselle una vicina all'altra; ogni casella viene chiamata **elemento** dell'array.
La **dimensione** dell'array una volta fissata non può essere modificata.
Ogni elemento si distingue dagli altri in base alla sua **posizione**.

Il valore corrispondente alla posizione di un elemento viene detto **indice** dell'elemento.

In molti linguaggi non esistono operazioni che agiscono sull'intero array; si lavora sempre su un elemento alla volta; per eseguire un'operazione su tutti gli elementi si utilizza un ciclo.

Per operare su un elemento dell'array si deve individuare l'elemento attraverso l'indice.

L'**indice** (che di solito è un numero intero) può essere un valore costante oppure una variabile in cui inserire di volta in volta il valore corrispondente all'elemento desiderato.

L'indice deve sempre fare riferimento ad un elemento **valido**.

Nota

L'indice di solito deve essere compreso tra 1 e la dimensione dell'array. In alcuni linguaggi l'indice degli elementi dell'array parte da 0 e quindi deve essere compreso tra 0 e la dimensione -1. In alcuni casi l'indice può essere anche negativo o addirittura qualcosa di diverso da un numero intero.

Se l'indice non fa riferimento ad un elemento valido si possono verificare dei problemi durante l'esecuzione poiché si cerca di invadere una parte di memoria non riservata all'array. In genere in questo caso viene segnalato un errore di esecuzione.

Il nome della variabile usata come indice è indifferente; si possono usare variabili diverse riferendosi allo stesso array, o la stessa variabile riferendosi ad array diversi; il nome dell'indice cioè non è legato all'array.

Come struttura **concreta** un array viene memorizzato in una struttura **sequenziale**, cioè usa locazioni **contigue** di memoria.

Come conseguenza dell'uso di una struttura sequenziale per la memorizzazione, l'array:

- ha dimensione statica;
- presenta difficoltà di inserimento e cancellazione di elementi.

Nota

In alcuni linguaggi è possibile ridimensionare gli array aggiungendo elementi quando un array è pieno.

7.3 CARICAMENTO E STAMPA DI UN ARRAY

I dati in un array possono essere caricati mediante una **lettura sequenziale**, cioè elemento per elemento, oppure **in modo casuale**, specificando attraverso l'indice l'elemento in cui si desidera inserire il valore.

Analogamente i dati possono essere stampati sequenzialmente o in modo casuale.

Lettura e stampa sequenziale

Se si lavora su tutti gli elementi in sequenza basta usare un indice inizializzato in modo da individuare il primo elemento e poi incrementarlo per ogni elemento dell'array.

Se si conosce in anticipo il numero degli elementi su cui operare risulta particolarmente adatto l'uso del ciclo con contatore.

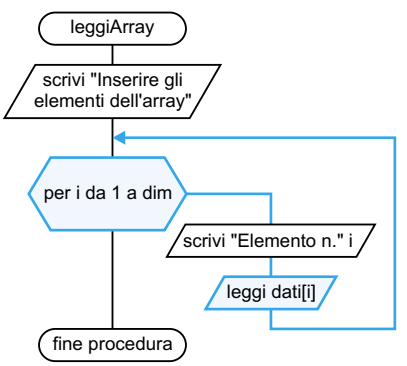
Se il primo elemento dell'array ha indice 1, usando un ciclo con contatore e variando l'indice da 1 alla dimensione dell'array si indicano successivamente tutti gli elementi.

La **dimensione** dell'array viene precisata al momento della definizione della variabile; sarebbe molto rigido però dover usare sempre lo stesso numero di elementi ad ogni esecuzione e dover creare un programma o una procedura diversi per leggere e stampare un array con un numero diverso di elementi; conviene chiedere all'utente quanti elementi dell'array devono essere effettivamente utilizzati; il programma così rimane valido per qualsiasi numero di elementi (naturalmente inferiore alla dimensione massima).

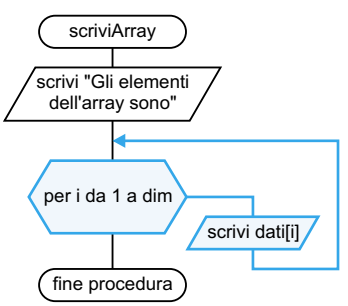
Esempio 7.1

Lettura e stampa sequenziale di un array.

```
O  dati   array
I  dim    intero     numero di elementi dell'array

leggiArray(rif dati, val dim)
    i     intero     usato come indice dell'array
      scrivi "Inserire gli elementi dell'array"
      per i da 1 a dim
          scrivi "Elemento n." i
          leggi dati[i]
      fine per
fine
```




```
I  dati   array
I  dim    intero     numero di elementi dell'array

scriviArray(val dati, val dim)
    i     intero     usato come indice dell'array
      scrivi "Gli elementi dell'array sono"
      per i da 1 dim
          scrivi dati[i]
      fine per
fine
```



Esempio 7.2

Lettura sequenziale di un array, senza conoscere in anticipo il numero di elementi da inserire.

Nota

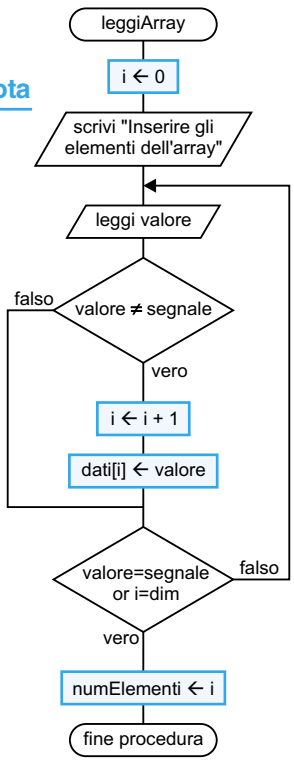
Bisogna usare uno dei metodi visti per terminare il ciclo, per esempio usare un valore come segnale per terminare.

O	dati	array	
I	dim	intero	numero di elementi dell'array
O	numElementi	intero	numero di elementi effettivamente inseriti

leggiArray(rif dati, val dim, rif numElementi)

	i	intero usato come indice dell'array
I	valore	del tipo degli elementi dell'array

```
    i ← 0
    scrivi "Inserire gli elementi dell'array"
    ripeti
        leggi valore
        se valore ≠ segnale allora
            i ← i + 1
            dati[i] ← valore
        fine se
    finché valore=segnale or i=dim
    numElementi ← i
fine
```



Lettura e stampa casuale

Per lavorare su un elemento qualsiasi dell'array bisogna assegnare all'indice il valore desiderato (che deve essere un indice valido).

Esempio 7.3

Lettura e stampa casuale di un elemento di un array.

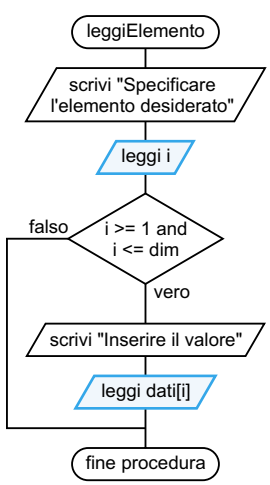
Lettura

I/O	dati	array	
I	dim	intero	numero massimo di elementi dell'array

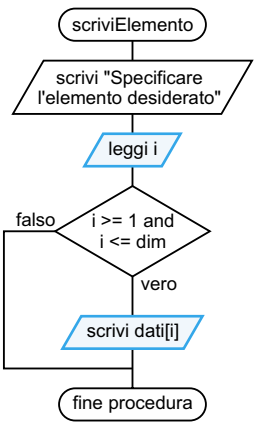
leggiElemento(rif dati, val dim)

	i	intero usato come indice dell'elemento

```
    scrivi "Specificare l'elemento desiderato"
    leggi i
    se i ≥ 1 and i ≤ dim allora
        scrivi "Inserire il valore"
        leggi dati[i]
    fine se
fine
```



```
I   dati   array
I   dim    intero      numero di elementi dell'array

scriviElemento(val dati, val dim)
    i   intero      usato come indice dell'elemento
      scrivi "Specificare l'elemento desiderato"
      leggi i
      se i ≥ 1 and i ≤ dim allora
            scrivi dati[i]
      fine se
fine
```



C++ 7.2

Il tipo array

Definizione di un array

La definizione di un **array** di dimensione fissa in C++ ha il formato:

```
tipoElementi nomeArray[dimensione];
```

dove *tipoElementi* indica il tipo di ciascun elemento dell'array e *dimensione* indica il numero di elementi dell'array.

```
int dati[10];
```
indica un array di 10 elementi interi individuati da un indice che varia da 0 a 9.

Le specifiche ISO C++ permettono che *dimensione* sia una variabile e non solo una costante.

```
int n, m = 10;
cin >> n;
int dati[n];
float misure[n+m];
```

Il tipo degli elementi può essere uno qualsiasi dei tipi utilizzabili in C++.

Per accedere ad un elemento dell'array si usa la notazione:

```
nomeArray[indice]
```

l'**indice** può essere una costante, una variabile o una espressione che una volta calcolata assume un valore minore di *dimensione*.

Per l'array *dati* dell'esempio precedente deve essere un numero tra 0 e 9 (compresi).

Se l'indice è esterno all'intervallo di definizione si verifica un errore di esecuzione.

Gli array sono memorizzati in **locazioni contigue** di memoria.
Il nome dell'array corrisponde alla locazione di memoria del suo primo elemento.

Array come parametri di funzioni o procedure

Un array usato come parametro di una funzione/procedura **viene sempre passato per riferimento**.

Nota

Il passaggio per riferimento permette di risparmiare memoria: gli array potrebbero occupare molto spazio e si evita la creazione della copia che si avrebbe con il passaggio per valore.

Per indicare che un parametro è un array, nella definizione della funzione il nome del parametro formale deve essere seguito da parentesi quadre, aperta e chiusa.

Al momento del richiamo della funzione invece il parametro attuale è indicato solo col nome.

Nota

Nella sintassi della chiamata non è distinguibile il fatto che un parametro sia un array.

```
void leggiArray(int dati[], unsigned int dim);
```
il parametro *dati* è un array di interi.

```
leggiArray(datiInteri, dimensione);
```
chiamata alle procedura (con *datiInteri* array correttamente dichiarato).

Il qualificatore const

Poiché gli array sono passati per riferimento, sono modificabili all'interno delle funzioni.
Se si desidera che un array passato come parametro non sia modificabile in una funzione, si deve far precedere la definizione del parametro dal qualificatore ***const*** nella dichiarazione della funzione.

```
void scriviArray(const int dati[], unsigned int dim);
```
una procedura che stampa i dati di un array non deve modificare l'array.

Il tentativo di modificare un valore dell'array definito mediante il qualificatore *const* provoca un errore di compilazione.

Esempio 7.4

Programma che usa procedure e funzioni per la lettura e stampa sequenziale di un array.

```
#include <iostream>
using namespace std;

const short Max = 100;

short leggiDimensione(short massimo);
void leggiArray(int dati[], short dim);
void scriviArray(const int dati[], short dim);

short leggiDimensione(short massimo){
    short dim;
    do{
        cout << "Inserire il numero di elementi (da 1 a "
             << massimo << ") ";
        cin >> dim;
    }while(!((dim >= 1) && (dim <= massimo)));
    return dim;
}//leggiDimensione

void leggiArray(int dati[], short dim){
    cout << "Inserire gli elementi dell'array" << endl;
    for(short i = 0; i < dim; i++){
        cout << "Elemento di indice " << i << ": ";
        cin >> dati[i];
    }//for
}//leggiArray
```

```
void scriviArray(const int dati[], short dim){
    cout << "Gli elementi dell'array sono:" << endl;
    for(short i = 0; i < dim; i++)
        cout << dati[i] << " ";
    cout << endl;
}//scriviArray

int main(void){
    short n;
    int numeri[Max];
    n = leggiDimensione(Max);
    leggiArray(numeri, n);
    scriviArray(numeri, n);
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 7.1 *Esecuzione dell'esempio*

Assegnazione di array

In C++ **non** è possibile **assegnare** un array ad un altro con un'unica operazione di assegnazione. La libreria **STL** nel file ***<algorithm>*** mette a disposizione la procedura ***copy*** per copiare il contenuto di un qualsiasi contenitore in un altro. Nel caso di due array dello stesso tipo e della stessa dimensione è possibile utilizzare *copy* con la sintassi

```
copy(v1, v1+dimensione, v2);
```

L'intero array *v1* viene copiato in *v2*; a questo punto esistono due array uguali.

Inizializzazione di array

Gli array si possono **inizializzare** al momento della **dichiarazione**; i valori iniziali devono essere specificati tra parentesi graffe, separati da virgole; non è necessario indicare esplicitamente la dimensione perché viene calcolata automaticamente in base al numero di valori.
Se viene indicata una dimensione minore del numero dei valori specificati, si verifica un errore di compilazione; se invece la dimensione è superiore al numero di valori specificati, l'array viene generato correttamente, con gli elementi finali nulli.

```
int primo[]={33, 56 , 78, 34 ,22, 11};
int secondo[6]={33, 56 , 78, 34 ,22, 11};
```

primo e *secondo* sono array di 6 elementi;

```
int terzo[5]={33, 56 , 78, 34 ,22, 11};
```

si ha un errore di compilazione poiché sono elencati 6 valori e la dimensione di *terzo* è 5;

```
int quarto[8]={33, 56 , 78, 34 ,22, 11};
```

quarto è un array di 8 elementi di cui gli ultimi due nulli.

L'indice dell'array

Per quanto riguarda il **tipo dell'indice**:

- se è numerico deve essere un **tipo intero** (quindi non può mai essere di tipo *float* o *double*);
- può essere anche un **tipo enumerativo** definito dall'utente; tuttavia se usato in un ciclo *for*, può essere necessario il sovraccarico (**overloading**) dell'operatore ++ o dell'operatore +.

Esempio 7.5

Uso del nome di un mese come indice di un array.

```
#include <iostream>
using namespace std;

enum mesi {gennaio, febbraio, marzo, aprile, maggio, giugno,
luglio, agosto, settembre, ottobre, novembre, dicembre};

mesi operator++(mesi& m, int i);

void inizializza(unsigned short numGiorni[]);

int main(void){
    unsigned short numGiorni[12];
    inizializza(numGiorni);
    for(mesi mese = gennaio; mese <= dicembre; mese++)
        cout << "Il mese " << mese+1 << " ha "
             << numGiorni[mese] << " giorni" << endl;
    getchar();
    return 0;
}//main

void inizializza(unsigned short numGiorni[]){
    numGiorni[gennaio] = 31;
    numGiorni[febbraio] = 28;
    numGiorni[marzo] = 31;
    numGiorni[aprile] = 30;
    numGiorni[maggio] = 31;
    numGiorni[giugno] = 30;
    numGiorni[luglio] = 31;
    numGiorni[agosto] = 31;
    numGiorni[settembre] = 30;
    numGiorni[ottobre] = 31;
    numGiorni[novembre] = 30;
    numGiorni[dicembre] = 31;
}//inizializza

/* Overloading dell'operatore postfisso ++ affinché possa operare
su un valore del tipo enumerativo mesi */
mesi operator++(mesi& m, int i){
    i = m;
    m = (mesi) (++i);
    return m;
}//operator++
```

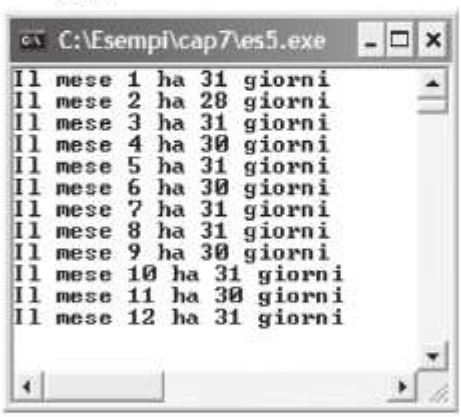


Figura 7.2 *Esecuzione dell'esempio*

Stringhe e array

Una **stringa** in realtà è un **array di caratteri**.
Il C permette la definizione di stringhe solo come array di caratteri:

```
char nomeStringa[dimensione];
```

In analogia con gli array numerici è possibile inizializzare una stringa al momento della dichiarazione:

```
char nomeStringa[] = "testo effettivo";
```

Conviene non specificare la dimensione, che viene determinata automaticamente in base ai caratteri inseriti; la dimensione effettiva è data dal numero dei caratteri + 1: il carattere aggiuntivo è la **sequenza di escape \0** che segnala la fine di una stringa.

Nota

È invece errata l'assegnazione:

```
nomeStringa = "testo effettivo";
```

perché il nome di un array corrisponde all'indirizzo della locazione di memoria del suo primo elemento.

Nota

Il file di libreria ***<cstring>*** contiene i prototipi delle funzioni per agire sulle stringhe viste come array di caratteri.

Funzione	Tipo parametri	Tipo restituito	Significato
strlen(str)	char[]	intero	restituisce il numero di caratteri della stringa;
strcmp(str1, str2)	char[], char[]	intero	confronta le stringhe e restituisce un numero <, = o > di 0 rispettivamente se *str1* precede *str2*, *str1* è uguale a *str2* e *srt1* segue *str2*;
strcpy(str1, str2)	char[], char[]	nessuno	copia in *str1* i caratteri di *str2*;
strcat(str1, str2)	char[], char[]	char*	concatena *str2* alla fine di *str1*; restituisce un puntatore a *str1*;

Il C++, oltre alla modalità come array di caratteri, mette a disposizione la classe ***string*** per la gestione delle stringhe; è possibile convertire da una modalità all'altra una stringa e accedere ad un singolo carattere di una stringa dichiarata *string*.

Date le seguenti dichiarazioni:

```
char s1[] = "ciao mondo";
string s2;
```

sono valide le istruzioni:

```
s2 = s1;
s2[5] = 'M';
strcpy(s1, s2.c_str());
cout << "La stringa s2 e' " << s2 << endl;
cout << "Il nono carattere di s2 e' " << s2[8] << endl;
cout << "La stringa s1 e' " << s1 << endl;
```

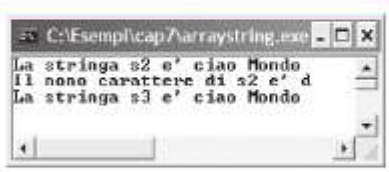


Figura 7.3 *Esecuzione dell'esempio*

Passaggio di parametri al programma

I **parametri** vengono passati al programma come un **array di stringhe**, argomento della funzione ***main()***.

Come nome di questo array di solito si usa ***argv***, per convenzione.

Nota

È possibile dichiarare un array di stringhe di cui si ignora la dimensione con le sintassi:

```
char *nomeArray[];
char **nomeArray;
```

Il motivo è spiegato nel capitolo 8.

Nota

Per gestire il passaggio dei parametri al programma la funzione *main* deve essere definita come:

```
int main(int argc, char *argv[])
```

oppure

```
int main(int argc, char **argv)
```

dove ***argv*** è l'array di stringhe che contiene i parametri e ***argc*** è la dimensione effettiva di *argv*.

Il primo elemento di *argv* (***argv[0]***) contiene il percorso assoluto e il nome dell'eseguibile.

Per passare i valori al *main*, basta scrivere da riga di comando:

```
nomeEseguibile valore1, valore2, ... , valoreN
```

Figura 7.4
La finestra Parametri

Nell'ambiente wxDev-C++ è possibile passare i valori scrivendoli nella relativa casella della finestra *Parametri* ottenuta dal comando *Esegui, Parametri...*.

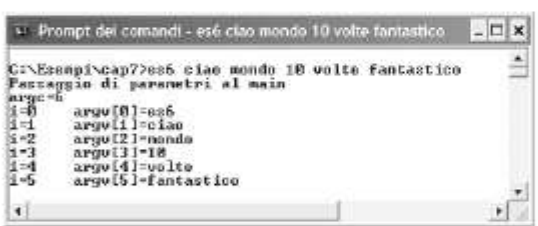
Se il programma è eseguito nel debug bisogna riscrivere i valori nella relativa casella della finestra *Parametri* ottenuta dal comando *Debug, Parametri...*.

Esempio 7.6

Stampa dei parametri passati a un programma.

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
    cout << "Passaggio di parametri al main\n";
    cout << "argc=" << argc << endl;
    for(int i=0; i<argc; i++)
        cout << "i=" << i << "    argv[" << i << "]="
             << argv[i] << endl;
    getchar();
    return 0;
}//main
```

Figura 7.5
Esecuzione dell'esempio



I parametri sono sempre stringhe. Se si vogliono passare dei parametri numerici bisogna convertire le stringhe in numeri.

Le funzioni ***atoi***, ***atof***, ***atol*** descritte nel file di intestazione ***<cstdlib>*** permettono la conversione rispettivamente in intero, in *float*, in *long*.

Funzione	Tipo parametri	Tipo risultato	Significato
atoi(str)	char[]	int	converte la stringa *str* in un *int*;
atol(str)	char[]	long	converte la stringa *str* in un *long*;
atof(str)	char[]	float	converte la stringa *str* in un *float*;

Esempio 7.7

Addizione di due numeri inseriti come parametri.

```
#include <iostream>
using namespace std;
```

```
int main(int argc, char *argv[]){
    int somma = atoi(argv[1])+atoi(argv[2]);
    cout << "La somma e': " << somma << endl;
    getchar();
    return 0;
}//main
```

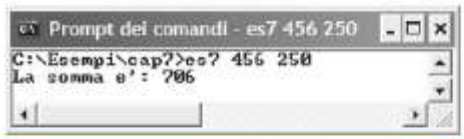


Figura 7.6 *Esecuzione dell'esempio*

7.4 UTILIZZO DI ARRAY

Array di contatori o totalizzatori

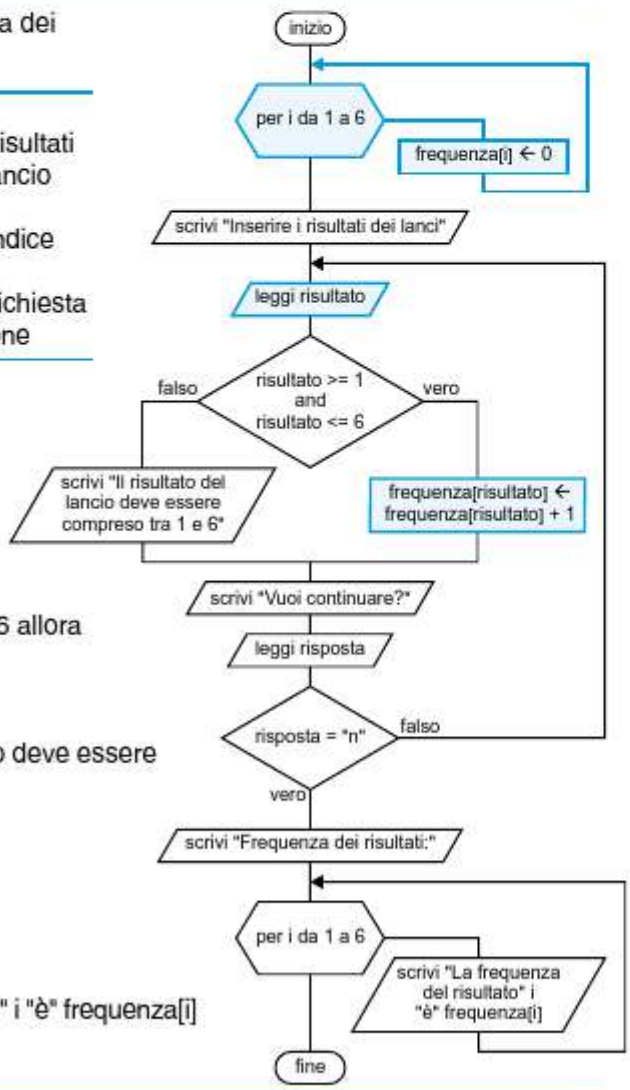
Si possono usare gli elementi di un array come **contatori** o **totalizzatori**.

Esempio 7.8

Uso di un array per contare la frequenza dei risultati dei lanci di un dado.

I/O	frequenza	array di 6 interi	contatori dei risultati
I	risultato	intero	risultato del lancio (da 1 a 6)
	i	intero	usato come indice dell'array
I	risposta	carattere	risposta alla richiesta di continuazione

```
inizio
    per i da 1 a 6
        frequenza[i] ← 0
    fine per
    scrivi "Inserire i risultati dei lanci"
    ripeti
        leggi risultato
        se risultato ≥ 1 and risultato ≤ 6 allora
            frequenza[risultato] ←
            frequenza[risultato] + 1
        altrimenti
            scrivi "Il risultato del lancio deve essere
            compreso tra 1 e 6"
        fine se
        scrivi "Vuoi continuare?"
        leggi risposta
    finché risposta = "n"
    scrivi "Frequenza dei risultati:"
    per i da 1 a 6
        scrivi "La frequenza del risultato" i "è" frequenza[i]
    fine per
fine
```



Array paralleli (o corrispondenti)

Spesso è utile usare due o più **array paralleli**, cioè array in cui gli elementi corrispondenti sono correlati tra di loro.

Per lavorare sugli array conviene usare un unico ciclo, usando lo stesso indice sui due array per fare riferimento agli elementi corrispondenti, cioè nella stessa posizione.

Per esempio conviene caricare simultaneamente lo stesso elemento nei due array.

Se si fanno operazioni che modificano l'ordine degli elementi, bisogna farle su tutti e due gli array.

Nota

Al posto di due array paralleli si può usare un **array di record**.

Esempio 7.9

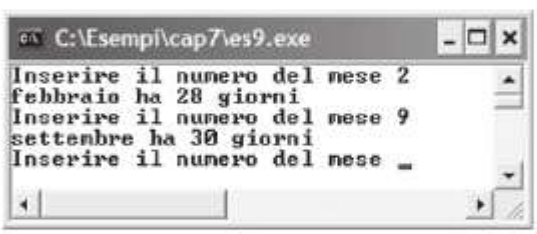
Inserire in due array paralleli i nomi dei mesi e il numero di giorni corrispondente.

I	m	intero (da 1 a 12)	mese richiesto
O	mesi	array di 12 stringhe	nomi dei mese
O	numGiorni	array di 12 interi (tra 28 e 31)	numero dei giorni del mese

```
inizio
    inizializza;
    ripeti
    scrivi "Inserire il numero del mese "
    leggi m
    se (m≥1) and (m≤12) allora
        scrivi(mesi[m], " ha ", numGiorni[m], " giorni");
    finché m=0;
fine
```

```
inizializza
    mesi[1] ← "Gennaio";
    numGiorni[1] ← 31;
    ...
    mesi[12] ← "Dicembre";
    numGiorni[12] ← 31;
fine
```

Figura 7.7 *Esecuzione dell'esempio*




```
#include <iostream>
using namespace std;

int main(void){
    unsigned short numGiorni[] = {31, 28, 31, 30, 31, 30, 31, 31,
30, 31, 30, 31};
    string mesi[] = {"gennaio", "febbraio", "marzo", "aprile",
"maggio", "giugno", "luglio", "agosto", "settembre", "ottobre",
"novembre", "dicembre"};
    int m;
    do{
        cout << "Inserire il numero del mese ";
        cin >> m;
        if((m >= 1) && (m <= 12))
            cout << mesi[m-1] << " ha " << numGiorni[m-1]
                 << " giorni" << endl;
    }while(m != 0);
    getchar();
    return 0;
}//main
```

Shift e rotazione

L'operazione di **shift** di un array consiste in uno spostamento dei suoi elementi; se si considera la struttura come circolare, è possibile pensare ad una **rotazione** degli elementi.
Nello shift verso sinistra o in senso **antiorario** tutti gli elementi vengono spostati verso sinistra di una posizione; nella rotazione il primo elemento deve essere portato nell'ultima posizione dell'array.
Nello shift verso destra o in senso **orario** tutti gli elementi vengono spostati verso destra di una posizione; nella rotazione l'ultimo elemento deve essere portato nella prima posizione dell'array.

Shift in senso antiorario

```
per i da 1 a dim-1
      dati[i] ← dati[i+1]
fine per
```

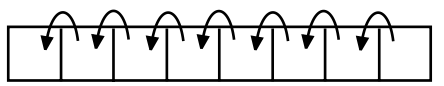


Figura 7.8 *Shift in senso antiorario*

Shift in senso orario

```
per i da dim a 2 con passo -1
      dati[i] ← dati[i-1]
fine per
```

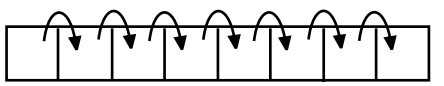


Figura 7.9 *Shift in senso orario*

Rotazione in senso antiorario

```
salva ← dati[1]
per i da 1 a dim-1
      dati[i] ← dati[i+1]
fine per
dati[dim] ← salva
```

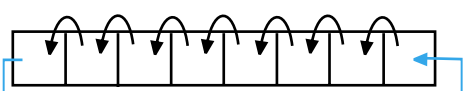


Figura 7.10 *Rotazione in senso antiorario*

Rotazione in senso orario

```
salva ← dati[dim]
per i da dim a 2 con passo -1
      dati[i] ← dati[i-1]
fine per
dati[1] ← salva
```

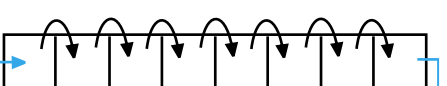


Figura 7.11 *Rotazione in senso orario*

Esempio 7.10

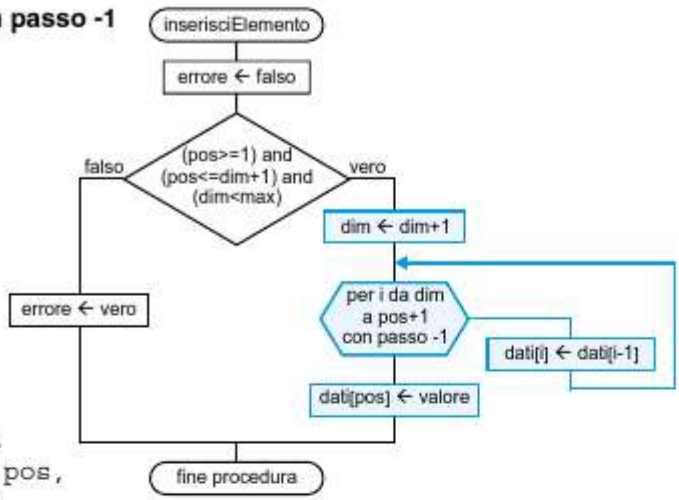
Inserimento di un elemento in un array (shift verso destra).

I/O	dati	array	
I/O	dim	intero	dimensione dell'array
I	max	intero	dimensione massima dell'array
I	pos	intero	posizione in cui si desidera inserire un nuovo elemento
I	valore	del tipo degli elementi dell'array	valore da inserire
O	errore	booleano	ha valore *vero* se l'inserimento non è riuscito

inserisciElemento(rif dati, rif dim, val max, val pos, val valore, rif errore)

```
    i       intero      usato come indice dell'array
    errore ← falso
    se (pos≥1) and (pos≤dim+1) and (dim<max) allora
        dim ← dim+1
```

```
        per i da dim a pos+1 con passo -1
            dati[i] ← dati[i-1]
        fine per
        dati[pos] ← valore
    altrimenti
        errore ← vero
    fine se
fine
```




```
void inserisciElemento(int
dati[], short& dim, short pos,
int valore, bool& errore){
    errore = false;
    if((pos >= 0) && (pos <= dim) && (pos < Max)){
        dim++;
        for(short i = dim; i > pos; i--)
            dati[i] = dati[i-1];
        dati[pos] = valore;
    }//if
    else
        errore = true;
}//inserisciElemento
```

Figura 7.12 *Esecuzione dell'esempio*

ESEMPIO 7.11

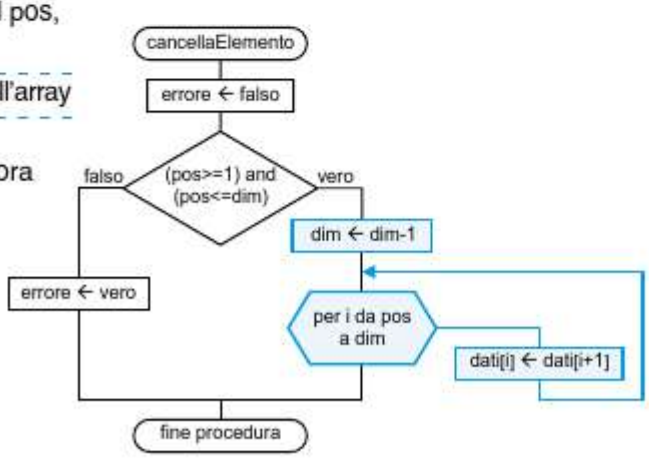
Eliminazione di un elemento in un array (shift verso sinistra).

I/O	dati	array	
I/O	dim	intero	dimensione dell'array
I	pos	intero	posizione dell'elemento da cancellare
O	errore	booleano	ha valore *vero* se l'eliminazione non è riuscita

cancellaElemento(rif dati, rif dim, val pos, rif errore)

i	intero	usato come indice dell'array

```
    errore ← falso
    se (pos ≥ 1) and (pos ≤ dim) allora
        dim ← dim-1
        per i da pos a dim
            dati[i] ← dati[i+1]
        fine per
    altrimenti
        errore ← vero
    fine se
fine
```



```
void cancellaElemento(int dati[], short& dim, short pos, bool&
errore){
    errore = false;
    if((pos >= 0) && (pos < dim)){
        dim--;
        for(short i=pos; i<dim; i++)
            dati[i] = dati[i+1];
    }//if
    else
        errore = true;
}//cancellaElemento
```

Figura 7.13 *Esecuzione dell'esempio*

Operazioni su array numerici

Su array con elementi di tipo numerico sono definite alcune operazioni, come la somma di due array, il prodotto per uno scalare (cioè per un numero) e il prodotto scalare (chiamato così perché dà come risultato un numero e non un array).

Somma di due array

La **somma** di due array può essere eseguita soltanto se i due array hanno lo stesso numero di elementi. Si ottiene come risultato un array, con lo stesso numero di elementi, in cui ogni componente risulta dalla **somma degli elementi corrispondenti** dei due array di partenza.

```
per i da 1 a dim
    somma[i] ← dati1[i] + dati2[i]
fine per
```

Prodotto di un array per uno scalare

Il **prodotto** di un array per un valore scalare viene eseguito moltiplicando ciascun elemento dell'array per il valore.

```
per i da 1 a dim
    dati[i] ← dati[i] * valore
fine per
```

Prodotto scalare di due array

Tra due array con lo stesso numero di elementi è definita una particolare operazione di prodotto detta prodotto scalare.
Il **prodotto scalare** di due array *a* e *b* di *n* elementi è definito come:

a[1] x b[1] + a[2] x b[2] + ... + a[n] x b[n]

```
prodscalare ← 0
per i da 1 a dim
    prodscalare ← prodscalare + dati1[i] * dati2[i]
fine per
```

7.5 RICERCA IN UN ARRAY

In molti problemi con gli array si richiede di individuare un elemento dell'array che soddisfi una determinata **condizione**.
Il programma deve esaminare ciascun elemento per stabilire quale soddisfi la condizione richiesta e restituire il **valore** individuato o la sua **posizione**.

Cercare un valore in un array significa individuare in quale elemento dell'array è presente il dato cercato, oppure stabilire che tale valore non è presente nell'array.

Per determinare l'**esito** della ricerca si può utilizzare una variabile booleana (che indica se il valore è stato trovato oppure no).
Se il valore viene trovato, per poterlo individuare è necessario conoscere l'**indice** dell'elemento che lo contiene.

Nota
Se il valore è presente più volte, la ricerca può restituire un array di posizioni.

Il metodo di ricerca può essere sequenziale o dicotomico.

Ricerca sequenziale

Compiere una **ricerca sequenziale** significa scandire ordinatamente tutti gli elementi dell'array, confrontandoli col valore cercato.
Bisogna esaminare uno alla volta gli elementi, finché si trova il valore cercato oppure si esauriscono gli elementi.

Esempio 7.12

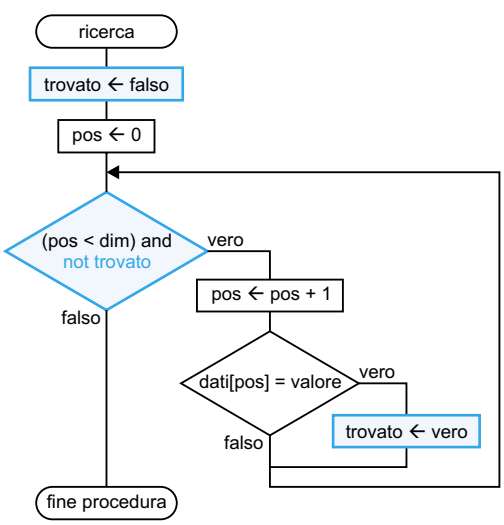
Procedura per la ricerca sequenziale di un valore in un array.

Viene usata una variabile booleana (*trovato*) inizialmente impostata a *falso*.
Il ciclo di ricerca termina se viene trovato il valore cercato (cioè *trovato* assume il valore *vero*) oppure se sono stati esaminati tutti gli elementi.

Nota
Non si usa un ciclo *for* perché non si devono sempre esaminare tutti gli elementi dell'array: quando il valore viene trovato il ciclo termina.

I	dati	array di *max* elementi	
I	dim	intero	dimensione dell'array
I	val	del tipo degli elementi dell'array	valore da cercare
O	trovato	booleano	ha valore *vero* se il valore è presente
O	pos	intero	se *trovato* è *vero* indica la posizione del valore cercato

```
ricerca(val dati, val dim, val valore, rif trovato, rif pos)
    trovato ← falso
    pos ← 0
    mentre (pos < dim) and not trovato
        pos ← pos + 1
        se dati[pos] = valore allora
            trovato ← vero
        fine se
    fine mentre
fine
```



Il programma che richiama la procedura deve controllare il valore di *trovato* e, se *trovato* ha valore *vero*, può usare il valore della posizione dell'elemento.

```
#include <iostream>
using namespace std;

const short Max = 100;

short leggiDimensione(short massimo);

void leggiArray(int dati[], short dim);

void ricerca(const int dati[], short dim, int valore, bool& trovato,
short& pos);

int main(void){
    short n, p;
    char risposta;
    int elemento;
    int numeri[Max];
    bool trovato;
    n = leggiDimensione(Max);
    leggiArray(numeri, n);
    do{
        cout << "Inserire il valore da cercare ";
        cin >> elemento;
        ricerca(numeri, n, elemento, trovato, p);
        if(trovato)
            cout << "Il valore " << elemento
                 << " e' presente all'indice " << p << endl;
        else
            cout << "Il valore " << elemento
                 << " non e' presente" << endl;
        cout << "Vuoi continuare (s/n)? ";
        cin >> risposta;
    }while(risposta != 'n');
    fflush(stdin);
    getchar();
    return 0;
}//main

void ricerca(const int dati[], short dim, int valore, bool& trovato,
short& pos){
    trovato = false;
    pos = -1; //perché il primo elemento ha indice 0
    while((pos < dim-1) && (!trovato)){
        if(dati[++pos] == valore)
            trovato = true;
    }//while
}//ricerca

short leggiDimensione(short massimo){
   ...
}//leggiDimensione

void leggiArray(int dati[], short dim){
   ...
}//leggiArray
```

Figura 7.14 *Esecuzione dell'esempio*

Ricerca dicotomica

La ricerca **binaria** o **dicotomica** può essere effettuata soltanto su di un insieme **ordinato** di dati; permette di suddividere i dati su cui si compie la ricerca in insiemi sempre più piccoli, fino ad individuare il valore cercato o a stabilire che tale valore non è presente.
Il valore da cercare viene confrontato con quello situato al centro dell'insieme di dati e, secondo l'esito del confronto, si prosegue la ricerca sui dati alla sinistra oppure su quelli alla destra, ripetendo sempre lo stesso procedimento.
La ricerca termina quando si trova il valore richiesto, oppure quando non esiste un altro intervallo di ricerca, perché l'ultimo si è ristretto ad un solo elemento (e quindi si può affermare che il valore non è presente).

Esempio 7.13

Ricerca dicotomica in un array ordinato.

Nota Se ci sono più valori uguali a quello cercato non è prevedibile quale venga segnalato; l'esito dipende dalle suddivisioni operate per la ricerca.

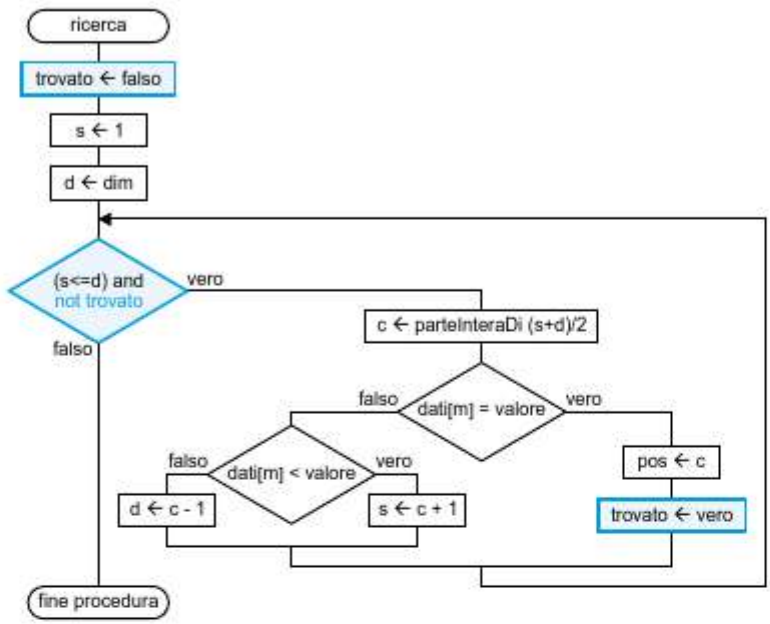
I	dati	array di *max* elementi	
I	dim	intero	dimensione dell'array
I	val	del tipo degli elementi dell'array	valore da cercare
O	trovato	booleano	ha valore *vero* se il valore è presente
O	pos	intero	se *trovato* è *vero* indica la posizione del valore cercato

```
ricerca(val dati, val dim, val valore, rif trovato, rif pos)
    s, d, c  interi  indici degli elementi a sinistra, a destra e al centro dell'intervallo considerato
    trovato ← falso
    s ← 1
    d ← dim
    mentre (s ≤ d) and not trovato
        c ← parteInteraDi (s+d)/2
        se dati[m] = valore allora
            pos ← c
            trovato ← vero
        altrimenti
            se dati[m] < valore allora
                s ← c + 1
            altrimenti
                d ← c - 1
            fine se
        fine se
    fine mentre
fine
```

Viene usata una variabile booleana (*trovato*) inizialmente impostata a *falso*.
Il ciclo di ricerca termina se viene trovato il valore cercato (cioè *trovato* assume il valore *vero*) oppure se si può stabilire che il valore non è presente poiché non esiste un altro intervallo di ricerca (*s>d*).

L'intervallo di ricerca viene individuato dagli indici *s* (sinistra) e *d* (destra); la prima volta vengono considerati tutti gli elementi dell'array quindi l'estremo sinistro viene impostato a

1 e l'estremo destro viene impostato a *dim* (numero degli elementi utilizzati).



Ad ogni ciclo viene determinato l'indice *c* dell'elemento centrale dell'intervallo. Si calcola la media degli indici degli estremi. Il valore cercato viene confrontato con l'elemento centrale dell'intervallo. Se il confronto ha esito positivo la variabile *trovato* viene impostata a *vero* e il ciclo di ricerca termina. Altrimenti l'esito del confronto permette di stabilire se continuare la ricerca nella metà destra dell'intervallo considerato, oppure nella metà sinistra.

Se il valore da cercare è maggiore dell'elemento centrale dell'intervallo, visto che l'array è ordinato, si possono scartare tutti gli elementi della metà sinistra e continuare la ricerca solo nella metà destra; per ridurre l'intervallo bisogna modificare l'indice dell'estremo sinistro, ponendolo uguale a *c + 1*.
Se il valore da cercare è minore dell'elemento centrale dell'intervallo si continua la ricerca solo nella metà sinistra, ponendo l'indice dell'estremo destro uguale a *c -1*.

```
void ricerca(const int dati[], short dim, int valore, bool& trovato,
short& pos){
    short s, d, c;
    trovato = false;
    s = 0;
    d = dim-1;
    while((s <= d) && (!trovato)){
        c = (s + d) / 2;
        if(dati[c] == valore){
            pos = c;
            trovato = true;
        }//if
        else
            if(dati[c] < valore)
                s = c + 1;
            else
                d = c - 1;
    }//while
}//ricerca
```

Figura 7.15 *Esecuzione dell'esempio*

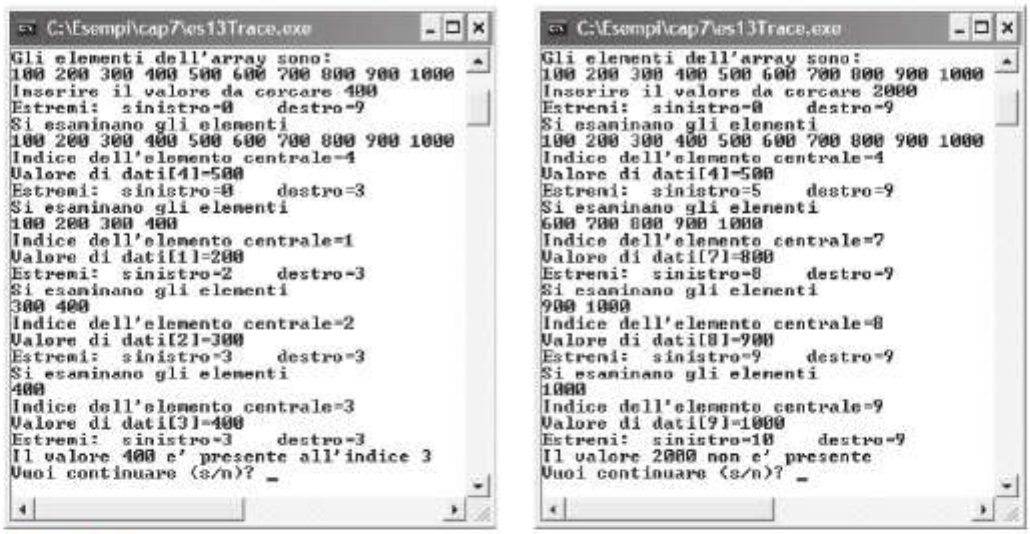


Figura 7.16 *Traccia di esecuzione del programma*

7.6 ORDINAMENTO DI UN ARRAY

Ordinare un array significa disporre in ordine (crescente o decrescente) i suoi elementi. Ci sono diversi **metodi** per eseguire l'ordinamento dell'array; uno dei metodi più semplici è quello per sostituzione o scambio. Un metodo migliore è quello a bolle o bubble sort.

Nota

Gli algoritmi presentati ordinano l'array in modo crescente; per ottenere l'ordinamento decrescente basta cambiare il verso del confronto.

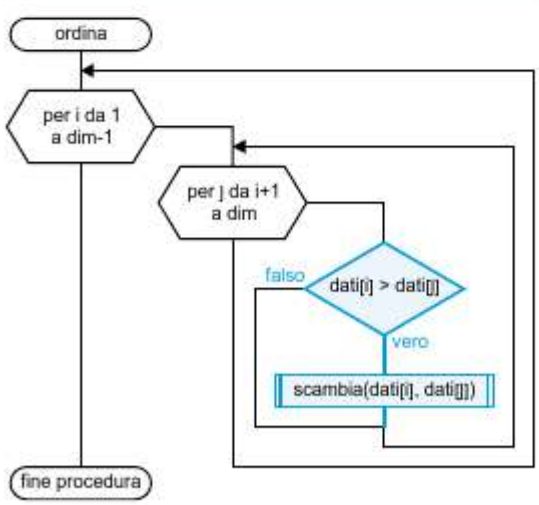
ORDINAMENTO PER SOSTITUZIONE O SCAMBIO

Si considera una posizione e si confronta il valore presente con tutti i successivi, scambiandolo quando risulta maggiore di quello con cui viene confrontato.

ESEMPIO 7.14

Ordinamento per sostituzione o scambio.

```
I/O  dati  array di max elementi
I    dim   intero      dimensione dell'array

ordina(rif dati, val dim)
   i, j    interi  usati come indici dell'array
      per i da 1 a dim-1
          per j da i+1 a dim
              se dati[i] > dati[j] allora
                  scambia(dati[i], dati[j])
              fine se
          fine per
      fine per
fine
```

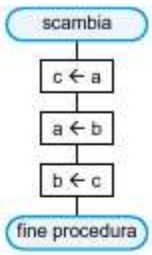


I/O a, b valori da scambiare

scambia(rif a, rif b)

c temporanea per lo scambio

```
    c ← a
    a ← b
    b ← c
fine
```



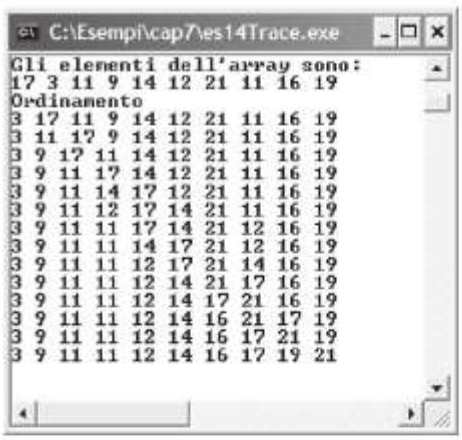


Figura 7.17 *Traccia di esecuzione*

```
#include <iostream>
using namespace std;

const short Max = 100;

short leggiDimensione(short massimo);
void leggiArray(int dati[], short dim);
void scriviArray(const int dati[], short dim);
void ordina(int dati[], short dim);
void scambia(int& a, int& b);

int main(void){
    short n;
    int numeri[Max];
    n = leggiDimensione(Max);
    leggiArray(numeri, n);
    scriviArray(numeri, n);
    ordina(numeri, n);
    cout << "Dopo l'ordinamento\n";
    scriviArray(numeri, n);
    fflush(stdin);
    getchar();
    return 0;
}//main

void scambia(int& a, int& b){
    int c;
    c = a;
    a = b;
    b = c;
}//scambia

void ordina(int dati[], short dim){
    for(short i = 0; i < dim-1; i++)
        for(short j = i+1; j < dim; j++)
            if(dati[i] > dati[j])
                scambia(dati[i], dati[j]);
}//ordina

short leggiDimensione(short massimo){ ... }//leggiDimensione

void leggiArray(int dati[], short dim){ ... }//leggiArray

void scriviArray(const int dati[], short dim){ ... }//scriviArray
```

Figura 7.18 *Esecuzione dell'esempio*

Ordinamento a bolle o bubble sort

Con questo metodo di ordinamento si confrontano a due a due gli elementi dell'array, scambiandoli se non sono in ordine. Si controlla più volte l'array terminando quando, durante l'ultimo controllo, non sono stati effettuati scambi (ciò significa che l'array è ordinato).
Per ridurre il numero dei confronti, poiché dopo la prima volta una parte dell'array risulta già ordinata, basta tener conto della posizione dell'elemento su cui è stato effettuato l'ultimo scambio e utilizzare questo valore come valore finale del ciclo di confronto.

Esempio 7.15

Ordinamento di un array con il metodo bubble sort.

Il ciclo per l'ordinamento dell'array termina quando non è più necessario effettuare scambi tra gli elementi, cioè quando la variabile *scambi* ha il valore *falso*.
Ad ogni ciclo la variabile *scambi* viene impostata a *falso*; se avviene almeno uno scambio il valore della variabile *scambi* viene modificato in *vero*.
Nel ciclo vengono confrontati a due a due gli elementi dell'array per vedere se sono necessari degli scambi per ordinare gli elementi; si cerca di portare all'inizio i valori più piccoli.
Se il valore di un elemento è più grande di quello che lo segue, i due elementi vengono scambiati.

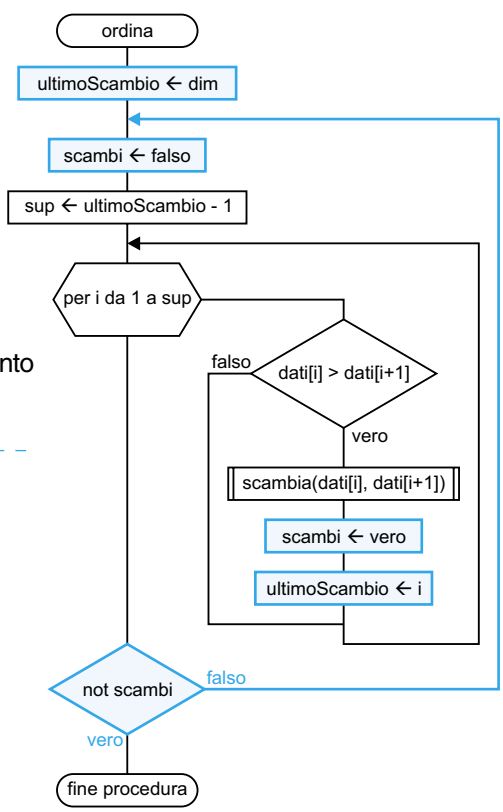
Nota Si potrebbe anche lavorare nel modo inverso portando verso il fondo i valori più grandi.

I/O	dati	array di *max* elementi	
I	dim	intero	dimensione dell'array

ordina(rif dati, val dim)

scambi	booleano	assume il valore *vero* se sono stati effettuati scambi, *falso* se l'array è ordinato
i	intero	usato come indice dell'array
ultimoScambio	intero	indice dell'ultimo elemento scambiato
sup	intero	indice dell'ultimo elemento della parte dell'array ancora da ordinare

```
    ultimoScambio ← dim
    ripeti
        scambi ← falso
        sup ← ultimoScambio - 1
        per i da 1 a sup
            se dati[i] > dati[i+1] allora
                scambia(dati[i], dati[i+1])
                scambi ← vero
                ultimoScambio ← i
            fine se
        fine per
    finché not scambi
fine
```



```
void ordina(int dati[], short dim){
    short ultimoScambio, sup;
    bool scambi;
    ultimoScambio = dim - 1;
    do{
        scambi = false;
        sup = ultimoScambio - 1;
        for(short i = 0; i <= sup; i++)
            if(dati[i] > dati[i+1]){
                scambia(dati[i], dati[i+1]);
                scambi = true;
                ultimoScambio = i;
            }//if
    }while(scambi);
}//ordina
```

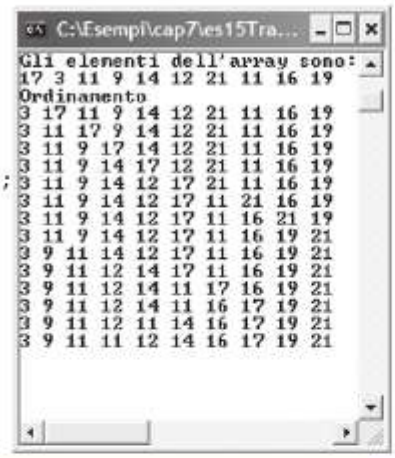


Figura 7.19 *Traccia di esecuzione*

7.7 FUSIONE O MERGE DI DUE ARRAY ORDINATI

Il merge è la **fusione** di due array **ordinati**; il risultato è un terzo array contenente tutti i valori dei due array di partenza, posti ancora in ordine.

Ogni array può contenere un numero diverso di elementi; il numero di elementi dell'array risultante è pari alla somma di quelli utilizzati dal primo più quelli utilizzati dal secondo.

Per ottenere il merge si prendono via via i valori minori dai due array di partenza, scandendo tali array un elemento alla volta; terminati gli elementi di un array, bisogna inserire nel nuovo array i restanti elementi dell'altro array.

ESEMPIO 7.16

Procedura per il merge di due array ordinati.

Se ci sono valori uguali nei due array vengono presi entrambi; nel caso particolare che i due array iniziali siano uguali, l'array risultante presenta ogni valore raddoppiato.

Si può modificare l'algoritmo in modo da prendere una sola volta eventuali valori comuni.

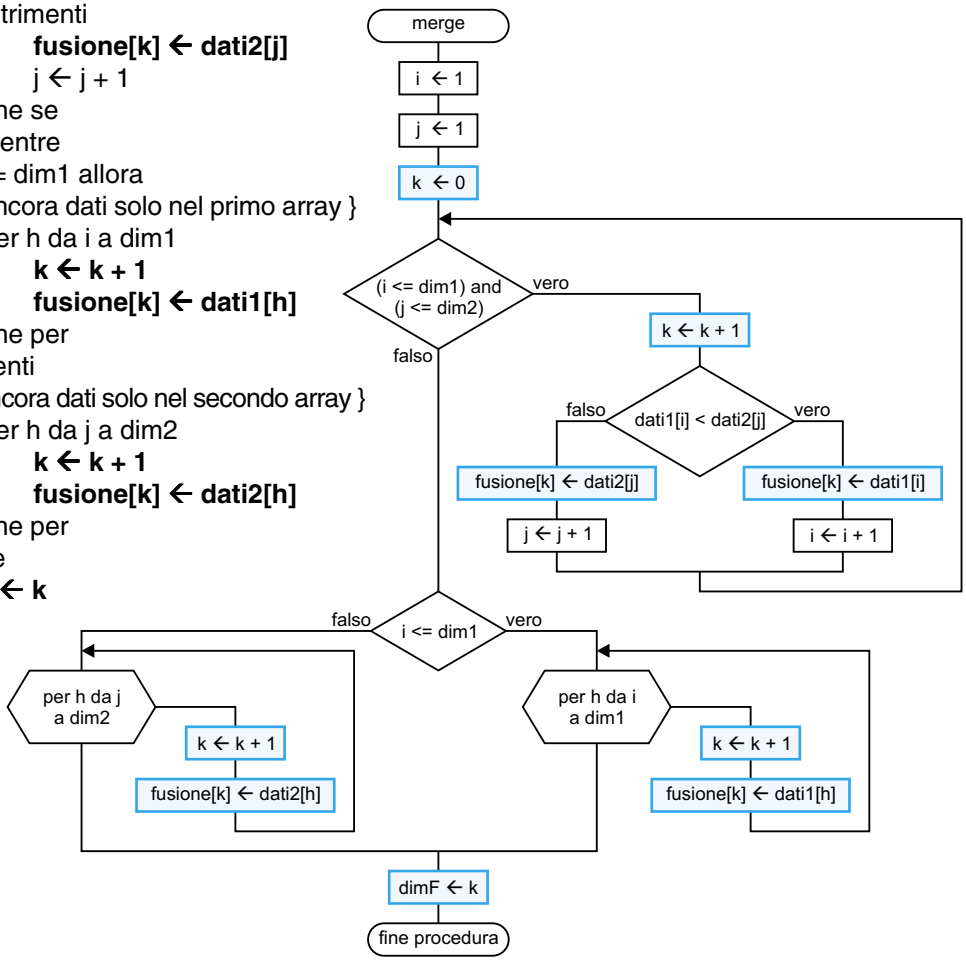
I	dati1, dati2	array	gli array da fondere	
I	dim1, dim2	interi	dimensioni degli array	
O	fusione	array di elementi dello stesso tipo di v1 e v2		array fusione
O	dimF	intero	dimensione dell'array risultante	

merge(val dati1, val dati2, val dim1, val dim2, rif fusione, rif dimF)

i, j, k, h	interi	usati come indici degli array

```
i ← 1
j ← 1
k ← 0
```

```
        mentre (i <= dim1) and (j <= dim2)
            k ← k + 1
            se dati1[i] < dati2[j] allora
                fusione[k] ← dati1[i]
                i ← i + 1
            altrimenti
                fusione[k] ← dati2[j]
                j ← j + 1
            fine se
        fine mentre
        se i <= dim1 allora
{ ci sono ancora dati solo nel primo array }
            per h da i a dim1
                k ← k + 1
                fusione[k] ← dati1[h]
            fine per
        altrimenti
{ ci sono ancora dati solo nel secondo array }
            per h da j a dim2
                k ← k + 1
                fusione[k] ← dati2[h]
            fine per
        fine se
        dimF ← k
fine
```



Gli indici *i* e *j* utilizzati rispettivamente per il primo e per il secondo array vengono inizializzati a 1 per partire dal primo elemento di ciascuno. L'indice *k* usato per l'array risultante viene inizializzato a 0 (l'array inizialmente è vuoto).
Il ciclo principale per la fusione dei due array viene ripetuto finché ci sono ancora elementi in ciascuno dei due array iniziali; quando tutti i valori di uno dei due array sono stati copiati nell'array risultante, questo ciclo termina e i dati rimasti nell'altro array vengono completati in modo diverso.
Nel ciclo viene incrementato l'indice dell'array risultante in modo che si riferisca alla prima posizione libera. Viene effettuato il confronto tra il primo valore non ancora copiato di ciascun array. Se è più piccolo il valore considerato nel primo array, l'elemento del primo array viene copiato nella prima posizione libera dell'array risultante; viene inoltre incrementato l'indice *i*, in modo che si riferisca al primo valore non ancora copiato del primo array. Altrimenti viene copiato l'elemento del secondo array e viene incrementato l'indice *j*, in modo che si riferisca al primo valore non ancora copiato del secondo array.
Quando gli elementi di uno dei due array sono stati completamente copiati nell'array risultante, basta copiare tutti i valori rimanenti dell'array non ancora completato.

Nella codifica l'indice *k* dell'array *fusione* viene inizializzato al valore -1 (array vuoto) perché l'indice dell'array parte da 0; viene incrementato prima del caricamento di ogni elemento (operatore ++ prefisso). Gli indici i e j, invece, vengono incrementati dopo il caricamento (operatore ++ postfisso).

Nota

La parola *merge* identifica una risorsa predefinita del namespace *std*. Per usarla come nome di una funzione definita dal programmatore non si deve includere l'intero namespace *std* ma solo le singole risorse necessarie.

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

const short Max = 100;

short leggiDimensione(short massimo);

void leggiArray(int dati[], short dim);

void scriviArray(const int dati[], short dim);

void merge(const int dati1[], const int dati2[], short dim1, short
dim2, int fusione[], short& dimF);

int main(void){
    short n1, n2, n3;
    int a[Max], b[Max], c[Max];
    n1 = leggiDimensione(Max);
    leggiArray(a, n1);
    n2 = leggiDimensione(Max);
    leggiArray(b, n2);
    cout << "Primo array:\n";
    scriviArray(a, n1);
    cout << "Secondo array:\n";
    scriviArray(b, n2);
    merge(a, b, n1, n2, c, n3);
    cout << "Merge:\n";
    scriviArray(c, n3);
    fflush(stdin);
    getchar();
    return 0;
}//main
```

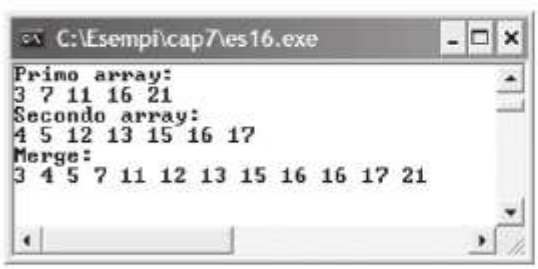


Figura 7.20 *Esecuzione dell'esempio*

```
void merge(const int dati1[], const int dati2[], short dim1, short
dim2, int fusione[], short& dimF){
    short i = 0, j = 0, k = -1;
    while((i < dim1) && (j < dim2))
        if(dati1[i] < dati2[j])
            fusione[++k] = dati1[i++];
        else
            fusione[++k] = dati2[j++];
    if(i < dim1)
        for(short h = i; h < dim1; h++)
            fusione[++k] = dati1[h];
```

```
    else
        for(short h = j; h < dim2; h++)
            fusione[++k] = dati2[h];
    dimF = k + 1; //qui k contiene l'ultimo valore di indice valido
}//merge

short leggiDimensione(short massimo){
    ...
}//leggiDimensione
void leggiArray(int dati[], short dim){
    ...
}//leggiArray
void scriviArray(const int dati[], short dim){
    ...
}//scriviArray
```

7.8 LE MATRICI

Una **matrice** è un insieme di valori a ciascuno dei quali sono associati più indici.
Gli elementi devono essere tutti **omogenei** tra loro, cioè tutti dello stesso tipo.

Per una matrice ad *n* dimensioni, ad ogni elemento sono associati *n* indici; ogni indice indica la posizione del valore lungo una dimensione.

L'**array** è un **caso particolare** di matrice: una matrice ad una dimensione.

Per una matrice a due dimensioni, pensabile come formata da **righe** e **colonne**, ad ogni valore sono associati **due indici**, uno dei quali indica la riga cui il valore appartiene (cioè la posizione sulla colonna), l'altro la colonna a cui il valore appartiene (cioè la posizione sulla riga).

A ciascun elemento della matrice si può accedere direttamente conoscendo gli indici che identificano l'elemento stesso.

Nel seguito del capitolo si fa riferimento soltanto a matrici a due dimensioni (ogni volta che ci si riferisce genericamente al termine matrice si intende matrice a due dimensioni). **Nota**

Gli **indici**, che possono essere valori sia costanti che variabili, devono sempre fare riferimento ad un elemento valido.

In molti linguaggi non esistono operazioni che agiscono sull'intera matrice; si lavora sempre su un elemento alla volta; per eseguire un'operazione su tutti gli elementi si utilizzano due cicli nidificati.

7.9 CARICAMENTO E STAMPA DI MATRICI

Lettura e stampa sequenziale

La lettura e la stampa di una matrice possono essere eseguite elemento per elemento, in modo **sequenziale**.
Per scandire in modo sequenziale tutti gli elementi di una matrice si possono usare due cicli enumerativi, uno interno all'altro, che fanno variare gli indici di riga e colonna.

La scansione sequenziale degli elementi può essere eseguita per righe o per colonne.

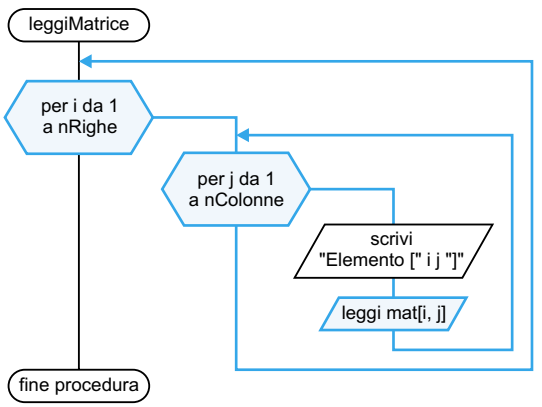
ESEMPIO 7.17

Lettura e stampa sequenziale di una matrice.

Lettura per righe

O	mat	matrice	
I	nRighe, nColonne	interi	numero di righe e di colonne della matrice

```
leggiMatrice(rif mat, val nRighe, val nColonne)
    i, j      interi  usati come indici di
                      riga e di colonna
        per i da 1 a nRighe
            per j da 1 a nColonne
                scrivi "Elemento [" i j "]"
                leggi mat[i, j]
            fine per
        fine per
fine
```



Lettura per colonne

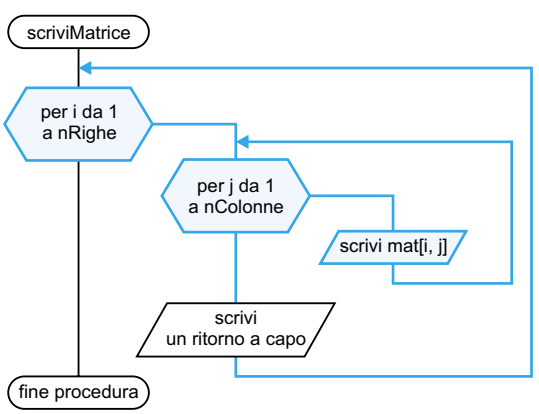
O	mat	matrice	
I	nRighe, nColonne	interi	numero di righe e di colonne della matrice

```
leggiMatrice(rif mat, val nRighe, val nColonne)
    i, j      interi  usati come indici di riga e di colonna
        per j da 1 a nColonne
            per i da 1 a nRighe
                scrivi "Elemento [" i j "]"
                leggi mat[i, j]
            fine per
        fine per
fine
```

Stampa per righe

I	mat	matrice	
I	nRighe, nColonne	interi	numero di righe e di colonne della matrice

```
scriviMatrice(val mat, val nRighe,
val nColonne)
    i, j      interi  usati come indici di
                      riga e di colonna
        per i da 1 a nRighe
            per j da 1 a nColonne
                scrivi mat[i, j]
            fine per
            scrivi un ritorno a capo
        fine per
fine
```



Gestione di elementi in modo casuale

È possibile riferirsi ad un **elemento** qualsiasi della matrice specificandone gli **indici**.

Per esempio è possibile inserire un valore in un elemento qualsiasi della matrice, o richiedere la stampa di un elemento qualsiasi, o incrementare il valore di un elemento qualsiasi, specificando gli indici che identificano l'elemento.

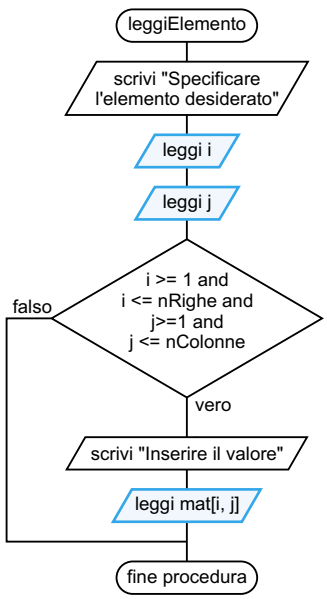
Esempio 7.18

Lettura e stampa casuale di un elemento di una matrice

Lettura casuale

I/O	mat	matrice	
I	nRighe, nColonne	interi	numero di righe e di colonne della matrice

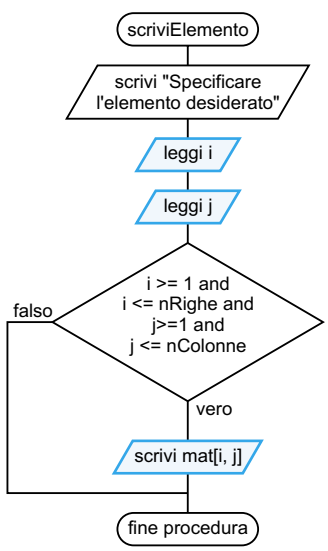
```
leggiElemento(rif mat, val nRighe, val nColonne)
    i, j    interi  usati come indici dell'elemento
    scrivi "Specificare l'elemento desiderato"
    leggi i
    leggi j
    se i ≥ 1 and i ≤ nRighe and j ≥ 1
    and j ≤ nColonne allora
        scrivi "Inserire il valore"
        leggi mat[i, j]
    fine se
fine
```



Stampa casuale

I	mat	matrice	
I	nRighe, nColonne	interi	numero di righe e di colonne della matrice

```
scriviElemento(val mat, val nRighe, val nColonne)
    i, j    interi  usati come indici dell'elemento
    scrivi "Specificare l'elemento desiderato"
    leggi i
    leggi j
    se i ≥ 1 and i ≤ nRighe
    and j≥ 1 and j ≤ nColonne allora
        scrivi mat[i, j]
    fine se
fine
```



C++ 7.3

Array a due dimensioni

La definizione di una **matrice** a due dimensioni di dimensioni fisse, in C++ ha il formato:

```
tipoElementi nomeMatrice[dim1][dim2];
```

dove *dim1* indica il numero di righe della matrice, *dim2* indica il numero di colonne e *tipoElementi* indica il tipo di ciascun elemento della matrice.
Il tipo degli elementi può essere uno qualsiasi dei tipi utilizzabili in C++.

```
int mdati[10][20];
```
indica una matrice di 200 elementi interi suddivisi in 10 righe individuate da un indice che varia da 0 a 9 e da 20 colonne individuate da un indice che varia da 0 a 19.

Per accedere ad un elemento di una matrice si usa la notazione:

```
nomeMatrice[indice1][indice2]
```

dove *indice1* corrisponde al numero di riga e *indice2* al numero di colonna dell'elemento considerato.

Non è possibile copiare una matrice in un'altra dello stesso tipo con una sola operazione di **assegnazione**, senza usare i cicli. Per copiare una matrice è possibile utilizzare la procedura *copy*.

```
copy(&mdati1[0][0], &mdati1[0][0]+nRighe*nCol, &mdati2[0][0])
```

Una matrice usata come parametro di una funzione viene sempre passata **per riferimento**.
Nella definizione della funzione, se un parametro è un array a più dimensioni, bisogna specificare tra coppie di parentesi quadre le dimensioni di tutte le componenti successive alla prima (la prima è facoltativa).
Al momento del richiamo della funzione il parametro attuale è indicato solo col nome.
Se si desidera che una matrice passata come parametro non sia modificabile in una funzione, si deve far precedere la definizione del parametro dal qualificatore ***const*** nella dichiarazione della funzione.

```
void leggiMatrice(int mdati[][dim2], int nRighe, int nCol);
void scriviMatrice(const int mdati[][dim2], int nRighe, int nCol);
```

In C una matrice è sempre vista come un array di array: il primo array ha tanti elementi quante sono le righe e ciascun elemento è un array avente tanti elementi quante le colonne.

Anche gli array multidimensionali sono memorizzati in **locazioni contigue** di memoria, una riga di seguito all'altra (l'indice della dimensione più a destra viene incrementato per primo).

L'elemento di posto *i*, *j* di una matrice *n* x *m* occupa la posizione *(i-1)m + j* (*i-1* sono le righe che precedono la riga *i*, ognuna di *m* elementi; sull'ultima riga gli elementi precedenti sono *j*).

Esempio 7.19

Lettura e stampa per righe di una matrice.

```
#include <iostream>
using namespace std;

const short Max = 100;

short leggiDimensione(string s, short massimo);

void leggiMatrice(int mat[][Max], short nRighe, short nColonne);

void scriviMatrice(const int mat[][Max], short nRighe, short nColonne);
```

```
short leggiDimensione(string s, short massimo){
    short dim;
    do{
        cout << "Inserire il numero di " << s
             << " (da 1 a " << massimo << ") ";
        cin >> dim;
    }while(!((dim >= 1) && (dim <= massimo)));
    return dim;
}//leggiDimensione
void leggiMatrice(int mat[][Max], short nRighe, short nColonne){
    cout << "Inserire gli elementi della matrice" << endl;
    for(short i = 0; i < nRighe; i++)
        for(short j = 0; j < nColonne; j++){
            cout << "Elemento di indici " << i << ", " << j << ": ";
            cin >> mat[i][j];
        }//for
}//leggiMatrice
void scriviMatrice(const int mat[][Max], short nRighe, short nColonne){
    cout << "Gli elementi della matrice sono:" << endl;
    for(short i = 0; i < nRighe; i++){
        for(short j = 0; j < nColonne; j++)
            cout << mat[i][j] << " ";
        cout << endl;
    }//for
}//scriviMatrice
int main(void){
    short m, n;
    int a[Max][Max];
    m = leggiDimensione("righe", Max);
    n = leggiDimensione("colonne", Max);
    leggiMatrice(a, m, n);
    scriviMatrice(a, m, n);
    fflush(stdin);
    getchar();
    return 0;
}//main
```

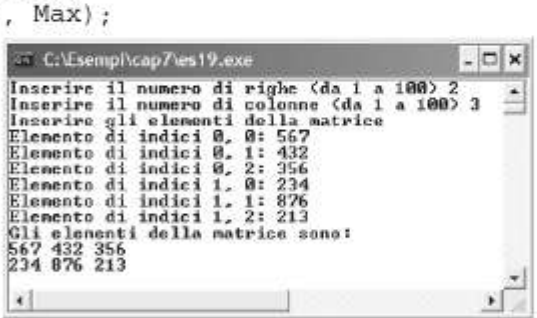


Figura 7.21 *Esecuzione dell'esempio*

Array di stringhe e matrici di caratteri

Un **array di stringhe** in realtà è una **matrice di caratteri**.
Il C permette la definizione di array di stringhe (della medesima lunghezza) solo come matrice di caratteri:

```
char arrayStringhe[numeroStringhe][lunghezzaStringa];
```

È lecito passare ad una procedura la i-esima stringa.

```
faiQualcosa(arrayStringhe[i]);
```

arrayStringhe[i] è l'indirizzo del primo carattere della i-esima riga.

7.10 UTILIZZO DI MATRICI

Matrici di contatori o totalizzatori

Si possono usare gli elementi di una matrice come **contatori** o **totalizzatori**.

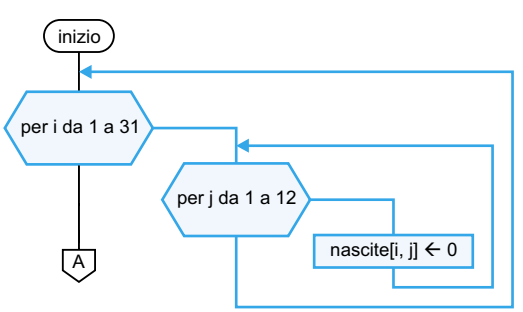
Esempio 7.20

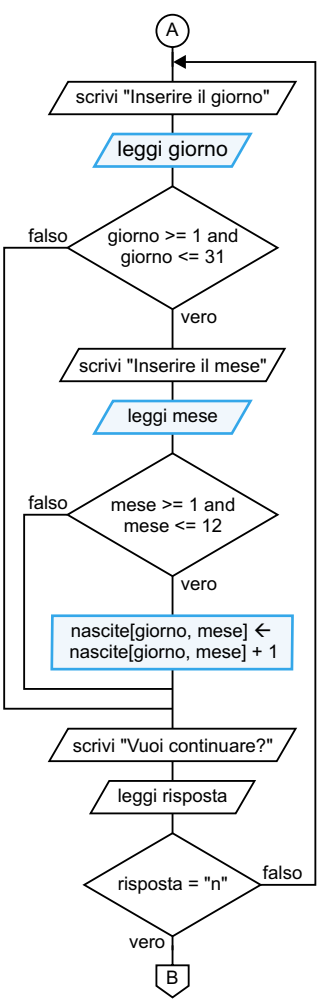
Statistica del numero degli individui nati in ogni giorno dell'anno.

Inserendo le date di nascita di ogni persona si vogliono conteggiare le nascite avvenute ogni giorno.
Gli elementi della matrice vengono usati come contatori; ogni elemento deve essere azzerato all'inizio del programma.
Il giorno e il mese identificano la riga e la colonna dell'elemento della matrice da incrementare.

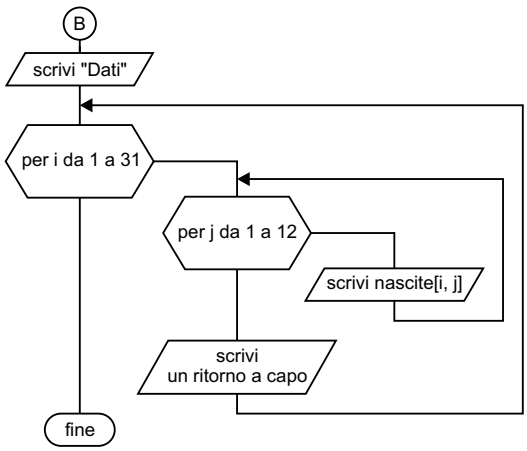
I/O	nascite	matrice di 31 righe e 12 colonne di interi	matrice dei contatori delle nascite
I	giorno	intero	giorno (da 1 a 31) della data di nascita o valore di terminazione (0)
I	mese	intero	mese (da 1 a 12) della data di nascita
	i, j	interi	usati come indici della matrice

```
inizio
    per i da 1 a 31
        per j da 1 a 12
            nascite[i, j] ← 0
        fine per
    fine per
    ripeti
        scrivi "Inserire il giorno"
        leggi giorno
        se giorno >= 1 and giorno <= 31 allora
            scrivi "Inserire il mese"
            leggi mese
            se mese >= 1 and mese <= 12 allora
                nascite[giorno, mese] ←
                nascite[giorno, mese] + 1
            fine se
        fine se
        scrivi "Vuoi continuare?"
        leggi risposta
    finché risposta = "n"
```





```
    scrivi "Dati"
    per i da 1 a 31
        per j da 1 a 12
            scrivi nascite[i, j]
        fine per
        scrivi un ritorno a capo
    fine per
fine
```



Nota

Il controllo effettuato sul giorno e mese assicura che venga effettivamente individuato un elemento della tabella, ma non garantisce che le date siano effettivamente corrette (30 febbraio, 31 aprile ecc.).

La matrice trasposta

La matrice che si ottiene stampando ogni colonna su una riga (o viceversa) si dice matrice **trasposta**.
Se la matrice è **quadrata**, cioè il numero di righe è uguale a quello delle colonne è possibile ottenere la matrice trasposta sulla matrice iniziale, scambiando in modo simmetrico gli elementi rispetto alla **diagonale** (cioè rispetto agli elementi in cui l'indice di riga è uguale all'indice di colonna).

Esempio 7.21

Stampa per colonne di una matrice (si ottiene la trasposta).

I	mat	matrice	
I	nRighe, nColonne	interi	numero di righe e di colonne della matrice

```
scriviMatrice(val mat, val nRighe, val nColonne)
    i, j    interi  usati come indici di riga e di colonna
    per j da 1 a nColonne
        per i da 1 a nRighe
            scrivi mat[i, j]
        fine per
        scrivi un ritorno a capo
    fine per
fine
```

Matrice unitaria

Una matrice **unitaria** è una matrice quadrata in cui solo gli elementi che si trovano sulla diagonale principale sono uguali a 1, tutti gli altri sono uguali a 0.

Operazioni su matrici numeriche

Sulle matrici con elementi di tipo numerico sono definite alcune operazioni, come la somma di due matrici, il prodotto per uno scalare (cioè per un numero) e il prodotto.

Somma di due matrici

La **somma** di due matrici può essere effettuata soltanto tra matrici con le stesse dimensioni: stesso numero di righe e stesso numero di colonne; la matrice somma mantiene le stesse dimensioni.
La somma è definita componente per componente; ogni elemento della matrice somma è il risultato della somma dei **corrispondenti** elementi delle matrici di partenza.

```
per I da 1 a nRighe
    per j da 1 a nColonne
        matSomma[i, j] ← mat1[i, j] + mat2[i, j]
    fine per
fine per
```

Prodotto di una matrice per uno scalare

Il **prodotto** di una matrice per uno **scalare** viene eseguito moltiplicando ciascun elemento della matrice per un valore scalare.

```
per I da 1 a nRighe
    per j da 1 a nColonne
        mat[i, j] ← mat[i, j] * valore
    fine per
fine per
```

Prodotto di due matrici

Il **prodotto** è definito tra due matrici *A* e *B* se il numero delle colonne di *A* è uguale al numero delle righe di *B*.
Sia *A* una matrice di dimensione *nRighe* x *k* e *B* una matrice di dimensione *k* x *nColonne*. Il prodotto delle due matrici dà come risultato una matrice di dimensione *nRighe* x *nColonne* in cui ogni elemento *i, j* è il risultato del prodotto scalare della *i*-esima riga della matrice *A* per la *j*-esima colonna della matrice *B* (ogni riga e colonna, fissato un indice, può essere vista come un array).

Il prodotto scalare della *i*-esima riga di *A* per la *j*-esima colonna di *B*, fissati *i* e *j*, diventa:

```
somma ← 0
per h da 1 a k
    somma ← somma + a[i, h] * b[h, j]
fine per
```

Ogni prodotto di questo tipo fornisce un elemento della matrice prodotto, che ha dimensione *nRighe* x *nColonne*; quindi, per calcolare tutti gli elementi, si deve fare un ciclo sugli elementi della matrice prodotto.

```
per I da 1 a nRighe
    per j da 1 a nColonne
        c[i, j] ← 0
        per h da 1 a k
            c[i, j] ← c[i, j] + a[i, h] * b[h, j]
        fine per
    fine per
fine per
```

7.11 I RECORD

Un **record** è una struttura costituita da più dati anche **non omogenei** tra loro; ciascun dato che compone il record si chiama **campo** e può essere identificato tramite il nome; viene associato un nome globale all'intero record e nomi diversi ai singoli campi.

DEFINIZIONE DI STRUTTURE

C++ 7.4

In C++ i record sono chiamati **strutture**.
Una struttura è definita mediante l'istruzione ***struct***:

```
struct nomeStruttura{
    tipo campo1;
    ...
    tipo campoN;
};//nomeStruttura
```

Nota

Il punto e virgola dopo la parentesi graffa chiusa è necessario.

Tra le parentesi graffe è contenuta la **definizione dei campi**; la definizione di ogni campo, ciascuno con il proprio tipo, termina con un punto e virgola (per l'ultimo campo il punto e virgola è facoltativo); se ci sono più campi dello stesso tipo vicini, si possono definire sulla stessa riga separandoli con virgole.

```
tipo campo1, campo2, ... campoN;
```

Un **campo** di una struttura può essere di uno qualsiasi dei tipi definiti in precedenza, per esempio un tipo semplice, un array o a sua volta una struttura.

La struttura è un tipo; bisogna dichiarare una variabile del tipo struttura definito, con

```
nomeStruttura nomeVariabile;
```

Si può fare riferimento a ciascun campo della struttura con la **notazione puntata**: nome della variabile del tipo struttura seguito da un punto e dal nome del campo:

```
nomeVariabile.nomeCampo
```

Si possono **inizializzare** le variabili struttura durante la dichiarazione:

```
nomeStruttura nomeVariabile = {valoreCampo1, ... , valoreCampoN};
```

l'ordine dei valori dei campi deve essere lo stesso della dichiarazione, altrimenti viene generato un errore di compilazione.

Le strutture vengono memorizzate come una sequenza di variabili contigue corrispondenti ai campi.

ESEMPIO 7.22

Gestione dei dati di un libro.

```
#include <iostream>
#include <iomanip>
using namespace std;

struct tipoLibro{
    string titolo, autore, editore;
```

```
    int numPagine;
    float prezzo;
};//tipoLibro
int main(void){
    tipoLibro libro;
    cout << "Inserire il titolo:" << endl;
    getline(cin, libro.titolo);
    cout << "Inserire l'autore:" << endl;
    getline(cin, libro.autore);
    cout << "Inserire l'editore:" << endl;
    getline(cin, libro.editore);
    cout << "Inserire il numero di pagine:" << endl;
    cin >> libro.numPagine;
    cout << "Inserire il prezzo:" << endl;
    cin >> libro.prezzo;
    cout << endl;
    cout << libro.titolo << endl;
    cout << libro.autore << endl;
    cout << libro.editore << endl;
    cout << libro.numPagine << endl;
    cout << setprecision(2) << setiosflags(ios::fixed)
         << libro.prezzo << " euro\n";
    fflush(stdin);
    getchar();
    return 0;
}//main
```

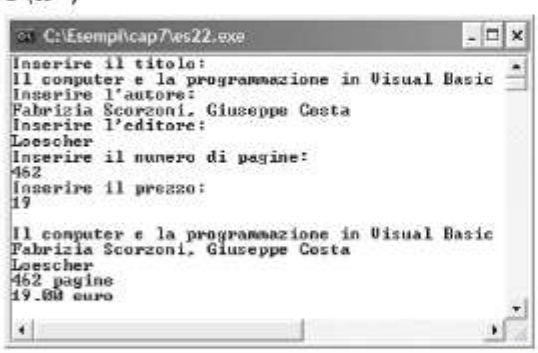


Figura 7.22 *Esecuzione dell'esempio*

7.12 LE TABELLE

Una **tabella** è un array di record; infatti un array di record si può rappresentare in forma tabellare utilizzando tante righe quanti sono i record e tante colonne quanti sono i campi di ciascun record.

ESEMPIO 7.23

Tabella di libri.

```
#include <iostream>
#include <iomanip>
using namespace std;
const short Max = 100;
struct tipoLibro{
    string titolo, autore, editore;
    int numPagine;
```

```
    float prezzo;
};//tipoLibro

short leggiDimensione(short massimo);
tipoLibro leggiLibro(void);
void scriviLibro(tipoLibro lib);
void leggiArray(tipoLibro dati[], short dim);
void scriviArray(const tipoLibro dati[], short dim);

int main(void){
    short n;
    tipoLibro libri[Max];
    n = leggiDimensione(Max);
    leggiArray(libri, n);
    scriviArray(libri, n);
    fflush(stdin);
    getchar();
    return 0;
}//main

tipoLibro leggiLibro(void){
    tipoLibro libro;
    fflush(stdin);
    cout << "Inserire il titolo:" << endl;
    getline(cin, libro.titolo);
    cout << "Inserire l'autore:" << endl;
    getline(cin, libro.autore);
    cout << "Inserire l'editore:" << endl;
    getline(cin, libro.editore);
    cout << "Inserire il numero di pagine:" << endl;
    cin >> libro.numPagine;
    cout << "Inserire il prezzo:" << endl;
    cin >> libro.prezzo;
    cout << endl;
    return libro;
}//leggiLibro

void scriviLibro(tipoLibro lib){
    cout << lib.titolo << endl;
    cout << lib.autore << endl;
    cout << lib.editore << endl;
    cout << lib.numPagine << " pagine" << endl;
    cout << setprecision(2) << setiosflags(ios::fixed)
         << lib.prezzo << " euro\n"
         << resetiosflags(ios::fixed);
}//scriviLibro

void leggiArray(tipoLibro dati[], short dim){
    cout << "Inserire gli elementi dell'array" << endl;
    for(short i = 0; i < dim; i++){
        cout << "Elemento di indice " << i << endl;
        dati[i] = leggiLibro();
    }//for
}//leggiArray
```

Figura 7.23 *Esecuzione dell'esempio*

```
void scriviArray(const tipoLibro dati[], short dim){
    cout << "Gli elementi dell'array sono:" << endl;
    for(short i = 0; i < dim; i++){
        cout << "Elemento di indice " << i << endl;
        scriviLibro(dati[i]);
        cout << endl;
    }//for
}//scriviArray

short leggiDimensione(short massimo){
    ...
}//leggiDimensione
```

Rottura di codice

Sia data una tabella in cui più record consecutivi hanno lo stesso valore per un campo (cioè ordinata in base al valore di quel campo, detto campo **chiave**); si vogliono eseguire determinate operazioni ogni volta che cambia il contenuto di tale campo; si parla in questo caso di **rottura di codice** (cambiamento del contenuto del campo).

Nota

La rottura di codice può avvenire anche su più livelli (per esempio a due livelli significa che il controllo di rottura di codice deve essere fatto su due diversi campi).

Le istruzioni da eseguire alla rottura di codice possono essere fatte quando **cominciano** i record con un certo valore, per esempio stampa di intestazioni, o quando **terminano**, per esempio stampe di totali, medie ecc.

Esempio 7.24

Tabella di libri: si vuole stampare una riga per ogni editore con l'elenco dei libri di quell'editore.

Per ogni editore si vuole stampare una intestazione con il nome dell'editore. Il controllo di rottura di codice deve essere effettuato sul campo editore (la tabella deve essere ordinata per editore).

Output: un prospetto di stampa costituito da una riga per ogni libro; il gruppo di tutti i libri che si riferiscono allo stesso editore è preceduto da una riga di intestazione con il nome dell'editore.

```
I   dati    array
I   dim     intero    numero di elementi dell'array

scriviTabella(val dati, val dim)
    i    intero    usato come indice dell'array
editorePrecedente: stringa valore del campo editore nel record precedente
    editorePrecedente ← ""
    scrivi "Elenco libri"
    per i da 1 a dim
        se dati[i].editore <> editorePrecedente allora
            scrivi "Editore: ", dati[i].editore
        fine se
        editorePrecedente ← dati[i].editore
        scrivi dati[i].titolo, dati[i].autore
    fine per
fine
```

L'elaborazione deve produrre una riga di stampa per ogni record; si deve inoltre produrre, quando necessario, la riga di intestazione con l'editore.
Per controllare se l'editore è cambiato si usa un campo che di volta in volta deve contenere il valore dell'editore relativo al record precedente.
Il valore dell'editore del record attuale viene confrontato con quello del record precedente; se è diverso si è in presenza di un caso di rottura di codice e si deve stampare l'intestazione e impostare il campo *editorePrecedente* per il confronto che si esegue sul successivo record.

(Si può pensare a una rottura di codice a due livelli stampando per ogni editore il nome di un autore e l'elenco dei titoli dei libri scritti da quell'autore, se la tabella è opportunamente ordinata).

```
#include <iostream>
using namespace std;

const short Max = 100;

struct tipoLibro{
    string titolo, autore, editore;
    int numPagine;
    float prezzo;
};//tipoLibro

short leggiDimensione(short massimo);

tipoLibro leggiLibro(void);

void leggiArray(tipoLibro dati[], short dim);

void ordina(tipoLibro dati[], short dim);

void scambia(tipoLibro& a, tipoLibro& b);

void scriviTabella(const tipoLibro dati[], short dim);

int main(void){
    short n;
    tipoLibro libri[Max];
    n = leggiDimensione(Max);
    leggiArray(libri, n);
    ordina(libri, n);
    scriviTabella(libri, n);
    fflush(stdin);
    getchar();
    return 0;
}//main
```

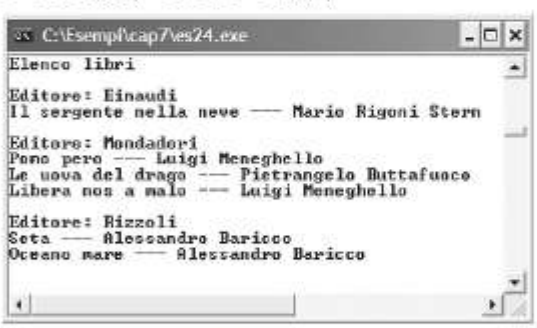


Figura 7.24 *Esecuzione dell'esempio*

```
void scambia(tipoLibro& a, tipoLibro& b){
    tipoLibro c;
    c = a;
    a = b;
    b = c;
}//scambia

void ordina(tipoLibro dati[], short dim){
    for(short i = 0; i < dim-1; i++)
        for(short j = i+1; j < dim; j++)
            if(dati[i].editore > dati[j].editore)
                scambia(dati[i], dati[j]);
}//ordina
```

```
tipoLibro leggiLibro(void){
    ...
}//leggiLibro
short leggiDimensione(short massimo){
   ...
}//leggiDimensione
void leggiArray(tipoLibro dati[], short dim){
   ...
}//leggiArray
void scriviTabella(const tipoLibro dati[], short dim){
    string editorePrecedente = "";
    cout << "Elenco libri\n";
    for(short i = 0; i < dim; i++){
        if(dati[i].editore != editorePrecedente)
            cout << "\nEditore: " << dati[i].editore << endl;
        editorePrecedente = dati[i].editore;
        cout << dati[i].titolo << " --- " << dati[i].autore << endl;
    }//for
}//scriviTabella
```

VERIFICA

QUESTIONARIO

1. Che cos'è una struttura di dati?
2. Qual è la differenza tra struttura astratta e struttura concreta?
3. Quali sono le caratteristiche principali che differenziano le strutture di dati?
4. Che cos'è un array?
5. Come viene memorizzato un array?
6. Che cos'è una matrice?
7. Che cos'è un record?
8. Che cos'è una tabella?
9. Che cosa si intende per rottura di codice?

TEST

1 ..... *Indicare se le seguenti affermazioni sono vere o false.*

V	F	
☐	☐	1) Un array può avere elementi non omogenei fra loro.
☐	☐	2) È possibile dichiarare una variabile array senza specificare il numero di elementi.
☐	☐	3) Tra gli argomenti di una procedura è possibile dichiarare un array senza specificare il numero di elementi.
☐	☐	4) In una procedura non è possibile passare un array per valore.
☐	☐	5) È sempre possibile modificare un array passato ad una procedura.

V	F	
☐	☐	6) La ricerca dicotomica può essere eseguita su qualsiasi array.
☐	☐	7) Si può definire un array a tre dimensioni.
☐	☐	8) Una matrice di stringhe è una tabella.
☐	☐	9) Una struttura può essere formata da campi non omogenei fra loro.
☐	☐	10) Si può definire un array in cui ogni elemento sia una struttura.
☐	☐	11) Si può definire una struttura in cui uno o più campi siano array.
☐	☐	12) Un array di strutture in cui le strutture hanno come campo un array è una matrice.
☐	☐	13) In una procedura non è possibile passare una struttura per valore.
☐	☐	14) Una funzione può restituire una struttura.

2 ..... *Individuare la risposta esatta.*

1) Dato il seguente frammento di programma:

```
long pongQ(int n){
    long p = 1;
    for(int i = 0; i < n; i++)
        p *= (-1);
    return p;
}//pongQ
void pingQ(long dati[], int d){
    int i = d;
    do{
        dati[--i] *= pongQ(i);
    }while(!(i == 1));
}//pingQ
```

che cosa fa la chiamata *pingQ(dati, 30)*?

- ☐ cambia di segno a tutti gli elementi;
- ☐ cambia di segno agli elementi al primo, terzo, quinto … posto;
- ☐ cambia di segno agli elementi al secondo, quarto, sesto … posto;
- ☐ nessuna delle precedenti.

2) Dato il seguente frammento di programma:

```
int pang(int n){
    return (-(n+1) % 2);
}//pang
void peng(long dati[], int d){
    for(int i = d-1; i >= 0; i--)
        dati[i] *= pang(i);
}//peng
```

che cosa fa la chiamata *peng(dati, MAX/2)*?

- ☐ azzera tutti gli elementi;
- ☐ azzera gli elementi al primo, terzo, quinto … posto e cambia di segno agli altri;
- ☐ azzera gli elementi al secondo, quarto, sesto … posto e cambia di segno agli altri;

- ☐ azzera gli elementi al secondo, quarto, sesto … posto;
- ☐ cambia di segno a tutti gli elementi.

3) Dato il seguente frammento di programma:

```
void modifica(long dati[], int n){
    for(int i = 1; i < n; i++)
        dati[i] = dati[i-1];
}//modifica
```

che cosa fa la chiamata *modifica(dati, 20)*?

- ☐ sostituisce ogni elemento con il primo elemento;
- ☐ sostituisce ogni elemento con quello alla sua destra;
- ☐ sostituisce ogni elemento con quello alla sua sinistra;
- ☐ sostituisce ogni elemento con l'ultimo elemento.

4) Che cosa fanno le procedure *elabora1* ed *elabora2*?

```
void elabora1(long dati[], int& d, int p){
    d--;
    for(int i = p; i < d; i++)
        dati[i] = dati[i+1];
}//elabora1
void elabora2(long dati[], int& d, int p){
    for(int i = p+1; i < d; i++)
        dati[i-1] = dati[i];
    d--;
}//elabora2
```

- ☐ entrambe eliminano l'elemento alla posizione *p*;
- ☐ entrambe eliminano l'elemento alla posizione *p+1*;
- ☐ la prima elimina l'elemento alla posizione *p*, la seconda quello alla posizione *p+1*;
- ☐ la prima elimina l'elemento alla posizione *p+1*, la seconda quello alla posizione *p*.

3 ..... *Segnare tutte le affermazioni esatte.*

1) Date le seguenti dichiarazioni:

```
struct TrecI{
    int sesto[5];
    string settimo;
};//TrecI
struct Trec{
    string primo;
    int secondo;
    int terzo[100];
    char quarto[5][6];
    TrecI quinto;
};//Trec
Trec rec;
TrecI otto;
```

individuare tutte le istruzioni che causano errori di compilazione:

```
cin >> primo;
cin >> rec.primo;
cin >> rec.primo[7];
cin >> rec.terzo;
cin >> rec.terzo['B'];
cin >> rec.quarto['C'-'A'];
cin >> rec.quarto['C'-'A']['F'-'B'];
for(char i='A'; i<'F';i++) cin >> rec.terzo[i];
cin >> rec.quinto;
cin >> rec.quinto.sesto;
cin >> rec.quinto.settimo;
cin >> rec.quinto.sesto['B'-65];
cin >> rec.quinto.settimo['B'-65];
for(char i='A'; i<'F';i++) cin >> rec.quarto[i-65];
cin >> rec.quinto.sesto[3];
cin >> rec.settimo[3];
cin >> rec.settimo;
cin >> rec.otto.settimo;
cin >> otto.settimo;
```

Esercizi

1 ..... Nell'esempio 7.8 sostituire l'inserimento dei risultati dei lanci del dado con la generazione di numeri casuali da 1 a 6 che simulino i lanci.

2 ..... Dato un array, stampare solo gli elementi di indice pari.

3 ..... Dato un valore, caricare tutti i suoi divisori in un array.

4 ..... Dati due array di numeri, calcolare l'array somma e l'array differenza.

5 ..... Dato un array di 7 giorni corrispondenti ai giorni della settimana, caricare in ogni elemento il nome del giorno (a partire da lunedì).

6 ..... Dati tre array di numeri, calcolare il prodotto scalare dei primi due e moltiplicare ogni elemento del terzo per il valore risultante.

7 ..... Inserire in un array dati tutti diversi tra di loro (controllare che il dato da inserire non sia già presente).

8 ..... Inserire in un array dati tutti diversi tra di loro, in modo che l'array risulti ordinato.

9 ..... Eseguire una ricerca sequenziale in un array ordinato (quando può terminare la ricerca?).

10 ..... Modificare le procedure di ordinamento degli esempi 7.14 e 7.15 in modo che effettuino l'ordinamento in ordine decrescente.

11 ..... Dati due array disordinati, ordinarli e farne il merge.

12 ..... Eseguire il merge di due array ordinati, prendendo una sola volta gli elementi comuni.

13 ..... Dati due array ordinati, individuare gli elementi comuni.

14 ..... Eseguire il merge di tre array ordinati.

15 ..... Sommare 4 matrici, procedendo alla somma di due matrici per volta.

16 ..... Realizzare la trasposta di una matrice quadrata.

17 ..... Descrivere il tracciato per una struttura che contenga:
- i dati relativi a un film;
- i dati relativi alla pagella di uno studente;
- i dati di un CD musicale.

18 ..... Data la tabella di libri dell'esempio 7.23 gestire ricerche in base al titolo o in base all'autore.

19 ..... Modificare l'esempio 7.24 in modo da stampare, dopo i libri di ogni editore, il numero medio di pagine e il prezzo medio.

20 ..... Data una stringa, contare quanti sono i simboli di punteggiatura e gli spazi presenti nel testo (Suggerimento: ricordare cosa definisce il file di intestazione cctype).

21 ..... Richiedere a una serie di persone se si spostano quotidianamente per raggiungere la sede di studio o lavoro; se la risposta è sì richiedere la distanza in km e il mezzo utilizzato (scegliendo tra le voci predefinite: treno, autobus, motorino, bicicletta, a piedi, altro). Contare la percentuale di utilizzo di ciascun mezzo e la distanza media percorsa, sempre per ciascun mezzo. (Usare un tipo ordinale per i mezzi di trasporto e utilizzarlo come indice di un array).

22 ..... Dato il seguente frammento di programma:

```
void cosaFai(int dati[], short d, short& cosa){
    bool s = true;
    cosa = 0;
    while(s){
        s = false;
        for(short i = 1; i < d; i++){
            if(dati[i] < dati[i-1]){
                cosa++;
                int t = dati[i-1];
                dati[i-1] = dati[i];
                dati[i] = t;
                s = true;
            }//if
        }//for
    }//while
}//cosaFai
```

A cosa serve la procedura *cosaFai* e qual è il compito del parametro *cosa*?

PROPOSTE DI LAVORO

1 ..... Dato un array di numeri, dire quanti sono gli elementi negativi, positivi e nulli.

2 ..... Dati due array, costruirne un terzo in cui ogni elemento sia il minimo dei corrispondenti elementi.

3 ..... Dati un array non ordinato e un valore di confronto, stampare gli elementi maggiori del valore.

4 ..... Dati un array ordinato e un valore di confronto, stampare gli elementi maggiori del valore.

5 ..... Dati un array e due valori di confronto, scrivere gli indici di tutti gli elementi compresi tra i due valori.

6 ..... Verificare se due array sono uguali (realizzare una funzione che restituisca un valore booleano).

7 ..... Dato un array, crearne un secondo in cui gli elementi risultino scambiati in modo simmetrico rispetto al primo.

8 ..... Scambiare gli elementi di un array in modo simmetrico (senza utilizzare un altro array).

9 ..... Verificare se una stringa è una palindrome (cioè se risulta uguale partendo da destra o da sinistra).

10 ..... Data una stringa dire quante sono le doppie presenti nel testo.

11 ..... Dato un array cancellare i dati contigui ripetuti, compattando a sinistra; visualizzare il numero di elementi cancellati e la lunghezza dell'array risultante.

12 ..... Dato un array, copiare gli elementi di indice dispari in un nuovo array.

13 ..... Dato un array di numeri, calcolare la media e costruire due array, uno con gli elementi minori e uno con gli elementi maggiori della media.

14 ..... Data una stringa, visualizzare la frequenza con cui compare ciascun carattere dell'alfabeto (per esaminare i caratteri della stringa considerarla come un array di caratteri). Determinare il carattere che compare con maggior frequenza.

15 ..... Data una stringa di testo, inserire in un array le parole che la compongono (una volta sola anche se ripetute); ordinare l'array per ottenere le parole in ordine alfabetico.

16 ..... Simulare le estrazioni della tombola generando numeri casuali tra 1 e 90 (controllare che ogni numero venga estratto una volta sola.

17 ..... Stabilire se un numero è primo provando come divisori solo i numeri primi precedenti.

18 ..... Trovare tutti i numeri primi minori d *n* utilizzando il metodo del crivello di Eratostene:
- si inseriscono i numeri da 2 a *n* in un array;
- 2 è primo; si stampa e si cancellano tutti i suoi multipli;
- il primo numero rimasto è primo; si stampa e si cancellano tutti i suoi multipli ecc.

19 ..... Dato un array contenente le temperature di una città per ogni giorno di un mese, calcolare le temperatura minima e massima e dire in quali giorni si sono verificate (se si sono verificate in più giorni, dire tutti i giorni); calcolare la temperatura media e lo scarto di ogni temperatura dalla media; dire in quale giorno la temperatura si è avvicinata di più alla media e in quali giorni è stata superiore alla media.

20 ..... Inserire una serie di dati riguardanti le vendite di un prodotto: data della vendita e quantità venduta. Creare e stampare un array che riporti per ogni mese le quantità totali di prodotto vendute. Determinare poi in quale mese si è verificata la vendita maggiore.

21 ..... Cancellare tutte le ripetizioni di valori in un array in modo che ogni valore compaia una sola volta:
- in array ordinato;
- in un array non ordinato.

22 ..... Dati due array creare (considerando i due casi, array ordinati o non ordinati):
- l'array intersezione (dato dagli elementi comuni);
- l'array unione (dato dagli elementi che appartengono a ciascun array, prendendo una sola volta i valori comuni);
- l'array differenza (dato da tutti i valori che appartengono a un array ma non all'altro).

23 ..... Dati due numeri interi, inserire ciascuna cifra in un elemento di un array ed eseguire le quattro operazioni aritmetiche procedendo cifra per cifra.

24 ..... Data una stringa che contiene un'espressione aritmetica con sole parentesi tonde, dire se è scritta correttamente (per controllare le parentesi basta sommare 1 per ogni parentesi aperta e sottrarre 1 per ogni parentesi chiusa; perché le parentesi siano corrette deve risultare 0).

25 ..... In un quiz viene posta a ciascun concorrente una serie di *n* domande la cui risposta può essere un numero da 1 a 4; controllare il numero delle risposte esatte per ciascun concorrente confrontando le risposte con quelle contenute in un array di *n* elementi introdotto all'inizio.

26 ..... Data una stringa di testo, determinare il numero delle parole e il numero di frasi, la lunghezza media delle parole e la lunghezza media delle frasi. Individuare la frase più lunga e la parola più lunga e dire dove iniziano.

27 ..... Data una stringa di testo, calcolare la frequenza delle parole che compaiono nel testo (usando due array paralleli, uno per le parole e uno per la frequenza, o un array di strutture composte dalla parola e dalla relativa frequenza).

28 ..... Scrivere il calendario di un mese, dati il nome del mese, il numero dei giorni e il giorno della settimana con cui inizia (memorizzare i nomi dei giorni della settimana in un array).

29 ..... Realizzare un diagramma relativo alle preferenze di alcune persone riguardo a una serie di libri, o film, o brani musicali. Il programma deve richiedere i titoli relativamente a cui calcolare le preferenze e memorizzarli in un array; deve poi utilizzare un altro array per contare le preferenze relative a ciascuna voce, man mano che vengono espresse. Al termine il programma deve riportare i dati e disegnare un semplice diagramma, in cui ogni barra non è altro che una serie di asterischi, tanti quante sono le preferenze espresse.

30 ..... Dati una serie di valori numerici rappresentarli con un istogramma.
Usare un array di caratteri per rappresentare una barra e inserire un numero di caratteri proporzionale al valore da rappresentare; il numero di caratteri per un valore *V* può essere calcolato come *DIM / Vmax x V* dove *DIM* è la dimensione dell'array e *Vmax* il valore massimo da rappresentare.

31 ..... Dato un array che rappresenti un byte, in cui ogni elemento rappresenti un bit, gestire un menu che permetta di:
- impostare o azzerare un determinato bit;
- eseguire shift o rotazione dei bit;
- eseguire l'operazione not, calcolando il not di ogni bit (0 se era 1, 1 se era 0);
- eseguire le operazioni or o and con un altro byte (array dello stesso tipo).

32 ..... Gestire un menu che permetta di ripetere a richiesta le seguenti operazioni su un array di parole:
- cancellazione di tutto l'array;
- inserimento di una parola nel primo elemento libero;
- stampa di tutte le parole contenute nell'array;
- stampa della parola alla posizione specificata;
- cancellazione della parola alla posizione specificata (spostando verso sinistra di una posizione tutte le parole successive);
- variazione della parola alla posizione specificata.

Esercizi da realizzare con array paralleli o con array di strutture

33 ..... Dizionario di parole in due lingue.
In base ad una scelta iniziale, ordinare le parole di una lingua e stampare il dizionario. Cercare una parola nell'array ordinato e fornire la corrispondente traduzione nell'altra lingua.

34 ..... Classifica dei partecipanti a una gara.
Dati i nomi dei partecipanti a una gara e i tempi realizzati rispettivamente da ciascun concorrente, produrre la classifica finale e individuare il vincitore.
Inserire i dati per ogni concorrente che termina la gara, in modo che la classifica sia sempre aggiornata.

35 ..... Gestione di una rubrica telefonica con nome, numero di telefono e indirizzo e-mail.
Gestire la visualizzazione di tutti i dati, la ricerca, l'inserimento di nuovi dati e la variazione di numeri di telefono e indirizzi e-mail.

36 ..... Orari degli autobus (numero autobus, destinazione, ora di partenza, ora di arrivo).
Consentire ricerche di vario tipo: per esempio orari di un autobus, numero dell'autobus per una certa destinazione, orario di partenza per arrivare a una destinazione a un certo orario ecc.

37 ..... Gestione dei dati di alcune persone: nome, sesso, età, statura, peso.
Determinare chi è il più giovane e chi il più vecchio, il numero percentuale di maschi e femmine, la statura media e il peso medio per sesso, l'età media.
Stampare a scelta i nomi di quelli con età superiore o inferiore alla media.
Ordinare i dati in base all'altezza; determinare chi ha l'altezza più vicina alla media.
Copiare i dati per gestire separatamente quelli dei maschi e quelli delle femmine.

38 ..... Ordini relativi ai prodotti in magazzino (codice, descrizione, quantità in magazzino).
A ogni ordine detrarre la quantità richiesta del prodotto indicato in base al codice; se la quantità in magazzino non è sufficiente, l'ordine non può essere soddisfatto; ad una specifica richiesta terminare il programma producendo la quantità rimasta in magazzino per i vari tipi di merce e un messaggio per la produzione con il totale degli ordini non soddisfatti per ogni tipo di prodotto.

39 ..... Dati dei diplomati: nome, punteggio, tipo di diploma.
Si vuole sapere il numero e la percentuale dei diplomati rispetto al tipo di diploma.
Si vuole avere inoltre la classifica in ordine decrescente di punteggio, totale o per un tipo di diploma, in base a una scelta.
Dato il nome di una persona si vuole sapere il punteggio e il tipo di diploma.

40 ..... Test con risposte a scelta multipla. *N* persone forniscono il nome e rispondono a 10 domande per ognuna delle quali è possibile scegliere tra 4 risposte (la risposta ad ogni domanda è un numero tra 1 e 4), di cui una sola esatta. Controllare le risposte di ogni persona e stampare per ogni persona il nome e il numero di risposte esatte ed un prospetto da cui appaia quali erano le risposte errate; le risposte esatte sono disponibili in un array. (Suggerimento: si può usare un array di strutture in cui ogni struttura contiene il nome di una persona e un array di 10 elementi per le risposte).

41 ..... Elenco degli studenti di una scuola ordinati per classe. Stampare per ogni classe: nome della classe, elenco degli studenti, totale degli studenti. Al termine, stampare numero di classi, numero totale di studenti e media per classe.

42 ..... Popolazione delle città italiane: regione, provincia, popolazione.
Ordinare i dati alfabeticamente per regione e all'interno della regione ordinare le province per popolazione decrescente. Stampare i dati fornendo la popolazione totale di ogni regione.
Avendo a disposizione i dati relativi a due anni, determinare per quali città l'incremento è stato maggiore, minore e più vicino all'incremento medio.

43 ..... Riepilogo delle fatture emesse ai clienti (cod. cliente, importo, data, in ordine di data).
Stampare una riga per ogni fattura, con i dati, e il totale del fatturato di ogni giorno e di ogni mese.

Matrici

44 ..... In una matrice di numeri eseguire le seguenti operazioni:
- cercare il valore minimo e la sua posizione;
- cercare il massimo di ogni riga;
- calcolare la media di ogni colonna e inserire i valori in un array;
- calcolare la media di tutti gli elementi e inserire in due array tutti i valori inferiori alla media e tutti i valori superiori alla media.

45 ..... Trasferire in un array tutti gli elementi di una matrice.

46 ..... Leggere un array di *n* elementi e disporre i dati in una matrice di *m* righe (calcolare il numero di colonne necessarie; procedere per righe e lasciare vuoti gli elementi dell'ultima riga eventualmente inutilizzati).

47 ..... Data una matrice quadrata, visualizzare gli elementi della diagonale principale e poi quelli della diagonale secondaria.

48 ..... Eseguire le seguenti operazioni su una matrice:
- shiftare verso il basso o verso l'alto, a scelta, ogni riga;
- shiftare verso sinistra o verso destra, a scelta, ogni colonna;
- scambiare in modo simmetrico le righe;
- scambiare in modo simmetrico le colonne;
- scambiare le posizioni di 2 colonne o di due righe scelte.

49 ..... Verificare se una matrice è simmetrica (cioè se è uguale alla trasposta).

50 ..... Creare una matrice che contenga una tavola Pitagorica di ordine *n*.

51 ..... Cercare un valore in una matrice e se presente restituirne la posizione.

52 ..... Ordinare ogni riga di una matrice.

53 ..... Ordinare tutti gli elementi di una matrice.

54 ..... Data una matrice in cui tutti gli elementi siano ordinati, inserire nuovi elementi al posto giusto, shiftando tutti gli altri in modo che la matrice rimanga ordinata.

55 ..... Date due matrici in cui tutti gli elementi siano ordinati, creare un array in cui siano presenti, in ordine, tutti i valori delle due matrici.

56 ..... Gestione delle risposte di *n* persone ad un questionario statistico composto da *m* domande per ognuna delle quali è possibile scegliere tra *k* risposte (la risposta ad ogni domanda è un numero tra 1 e *k*); calcolare il numero di risposte di ogni tipo per ogni domanda e stampare una tabella con i risultati.

57 ..... Risultati di domande a risposta multipla. Controllare i risultati di un test eseguito dagli studenti di una classe, confrontandoli con i risultati esatti presenti in un array. Stabilire anche per ogni domanda quanti studenti hanno risposto esattamente e per quante domande le risposte esatte sono state 0, 1, 2 ecc.

58 ..... Dato il numero degli alunni assenti (distintamente) per le classi di un istituto per ogni giorno della settimana, determinare:
- il totale delle assenze per ogni giorno della settimana complessivamente per tutte le classi;
- la media di assenze giornaliere per ogni classe;
- la media di assenze giornaliere per tutte le classi complessivamente.

59 ..... Si vuole fare una statistica sui consumi della popolazione.
Per ogni persona vengono registrati i dati riguardanti le spese annue per vitto, salute, vestiario, trasporti, casa, altre spese e il reddito;
- calcolare il reddito medio e il risparmio medio delle persone intervistate;

- calcolare la percentuale media di incidenza di ciascuna voce sull'ammontare delle spese.

60 ..... Vendite di un prodotto per ogni mese di 10 anni. Determinare:
- in quale anno si è avuta la vendita maggiore e di tale anno in quale mese;
- le vendite nei vari anni per un mese richiesto;
- la media delle vendite mensili per ogni anno, stabilendo anche in quale anno è stata maggiore;
- in quale mese dell'anno si è verificata più spesso la vendita massima.

61 ..... Data una matrice che riporti le distanze chilometriche tra città e un array contenente i nomi delle città:
- date due città, fornire la distanza tra di esse;
- data una città, fornire la distanza da tutte le altre.

62 ..... Data una tabella riportante le relazioni tra alberghi e servizi offerti (una X all'incrocio tra la riga dell'albergo e la colonna del servizio se l'albergo offre tale servizio) e due array con le descrizioni degli alberghi e dei servizi, fornire:
- l'elenco di tutti gli alberghi che offrono un determinato servizio;
- l'elenco di tutti i servizi e per ognuno di essi l'elenco degli alberghi che lo offrono;
- l'elenco di tutti i sevizi offerti da un albergo richiesto.

63 ..... Operazioni di voto per una elezione. Ogni votante può scegliere una lista (tra *n* liste) e due preferenze (tra un numero variabile di candidati per ogni lista).
Al termine si vuole ottenere per ciascuna lista il numero dei voti ottenuti e il numero di voti di ciascun candidato.

64 ..... Stabilire se una parola è contenuta orizzontalmente o verticalmente in una matrice alfanumerica in cui ogni elemento contiene un carattere.

65 ..... Life: è una simulazione che si svolge su una griglia rettangolare in cui ogni cella può essere occupata da un organismo vivente (cella vuota spazio, cella occupata X). Le generazioni si sviluppano secondo le seguenti regole:
1) i vicini di una cella sono le 8 celle che la toccano orizzontalmente, verticalmente e diagonalmente;
2) se una cella è vivente e nessuna o al più una cella tra i vicini lo è, nella successiva generazione la cella muore di solitudine;
3) se una cella è vivente e quattro o più celle vicine lo sono, nella generazione successiva la cella muore per sovrappopolamento;
4) una cella vivente con due o tre vicini vivi rimane viva nella generazione successiva;
5) se una cella non è vivente, diventa vivente nella generazione successiva se ha esattamente tre vicini viventi;
6) tutte le nascite e le morti avvengono nello stesso momento.

Stampare *n* generazioni o la *n*-esima generazione.

66 ..... Una figura digitalizzata è una matrice di punti ciascuno dei quali è bianco o nero (spazio o X). Si desidera:
- ingrandire la figura di un fattore 2, ingrandendo ogni punto della matrice originale ad un quadrato 2x2 nell'immagine risultante;
- ridurre la figura di un fattore 2, riducendo ogni quadrato 2x2 della matrice originale ad un solo punto nell'immagine risultante: se almeno due dei quattro punti nel quadrato originale sono neri il corrispondente punto nella riduzione è nero altrimenti è bianco.

67 ..... Data una matrice sparsa (una matrice che contiene molti elementi a zero) rappresentarla con un array di strutture in cui ogni struttura contiene il valore e gli indici della posizione.

8 Strutture di dati dinamiche e puntatori

Prerequisiti

Saper descrivere algoritmi e realizzare programmi in C++ che usano:

- strutture di controllo
- procedure e funzioni
- strutture di dati

Obiettivi

Conoscenze

- Tipi di dati astratti e concreti
- Pile e code
- Liste concatenate
- Grafi
- Alberi
- Allocazione dinamica della memoria

Abilità/capacità

- Descrivere algoritmi che utilizzano strutture di dati dinamiche
- Scrivere programmi in C++ che usano puntatori
- Progettare nuovi tipi di dati

8.1 STRUTTURE DI DATI DINAMICHE

Le strutture di dati **dinamiche** sono strutture di dati in cui il numero degli elementi non è fisso ma può variare durante l'esecuzione.

Si dicono strutture di dati **lineari** le strutture di dati in cui per ogni elemento è definito un solo successore e un solo predecessore (liste).
Le strutture di dati **non lineari** possono avere per ogni elemento più predecessori o più successori (alberi e grafi).

RAPPRESENTAZIONE DI NUOVI TIPI DI DATI

Perché sia possibile utilizzare una struttura di dati in un programma, bisogna che il linguaggio di programmazione con cui viene codificato il programma offra un tipo di dati corrispondente.

Un tipo di dati astratto (**ADT** - Abstract data type) è un oggetto matematico definito da:

- un insieme di valori (dominio);
- un insieme di operazioni;
- un insieme di costanti.

Un tipo di dati **concreto** permette di usare il tipo di dato in un particolare linguaggio; oltre alle proprietà astratte, bisogna definire come rappresentare il tipo di dati tenendo conto dei vincoli imposti dal linguaggio.
Per rappresentare un nuovo tipo di dati si possono utilizzare tipi di dati già esistenti.

Non esiste un'unica rappresentazione: possono essere possibili più rappresentazioni diverse; per scegliere la rappresentazione migliore, oltre alla correttezza della soluzione, bisogna valutare l'**efficienza** in base all'occupazione di memoria e al costo di esecuzione delle operazioni.

Esempi di nuovi tipi di dati sono **liste** (di cui pile e code sono casi particolari), **alberi** e **grafi**.

LINGUAGGI DI PROGRAMMAZIONE E STRUTTURE DI DATI DINAMICHE

Per realizzare strutture di dati dinamiche di solito i linguaggi permettono di definire dei **puntatori** e offrono istruzioni per utilizzarli.

Si può anche **simulare** l'uso di strutture dinamiche mediante **array**.
Si possono usare gli elementi dell'array come se si trattasse degli elementi della struttura desiderata, definendo le procedure per inserire o estrarre elementi, o per compiere qualsiasi altra operazione necessaria.

8.2 LISTE

Una **lista** (o sequenza) è un insieme di valori **omogenei**, cioè tutti dello stesso tipo, per cui è stabilito un **ordine** nella posizione.

Per ogni elemento della lista (**nodo**) è definito un **successore** e un **predecessore**; ovviamente il primo elemento non ha predecessore e l'ultimo non ha successore.

Non è possibile accedere direttamente ad un elemento; per esaminare un elemento all'interno della lista bisogna scorrere tutti gli elementi che lo precedono (**accesso sequenziale**).

Una lista si dice a **lunghezza variabile** se il numero di elementi varia nel tempo per effetto di inserimenti o cancellazioni di elementi (altrimenti si dice a **lunghezza fissa**).

8.3 PILE

La **pila**, chiamata anche **stack**, è un caso particolare di lista lineare in cui gli elementi si possono inserire o estrarre da un solo estremo.

Il criterio utilizzato per la gestione di una pila si dice **LIFO** (Last In First Out, l'ultimo che entra è il primo che esce), cioè viene estratto per primo l'ultimo valore che è stato inserito.

Si può pensare ad una pila di piatti appoggiati sul tavolo; ogni piatto lavato viene appoggiato sopra la pila di piatti e ogni volta che serve un piatto si prende il primo disponibile.

Realizzazione di una pila con un array

Per realizzare la pila si può utilizzare un **array** per contenere gli elementi ed una variabile per indicare la **cima** della pila (l'ultimo elemento inserito).
Oltre alla creazione della pila, si devono definire le operazioni di inserimento (**push**) ed estrazione dalla cima della pila (**pop**).
Poiché l'array ha una dimensione prefissata, bisogna gestire il caso in cui sia pieno: bisogna fissare un limite massimo per il numero di elementi che possono essere inseriti e la procedura *push* deve segnalare quando l'array è pieno.

La pila può essere implementata con un record che contiene sia l'array degli elementi della pila che la variabile utilizzata per la cima.

Esempio 8.1

Gestione di una pila con un array.

Costanti
max: dimensione dell'array utilizzato per simulare la pila.

Tipi
dimensione: tipo dell'indice dell'array (valori da 1 a *max*).
tipoPila: record formato da
  pila: array usato per simulare la pila;
  cima: indice dell'ultimo elemento inserito (all'inizio vale 0).
tipo: tipo degli elementi gestiti dalla pila.

Basta cambiare la definizione del tipo *tipo* per inserire nella pila elementi di qualsiasi tipo (ma tutti omogenei tra loro).

Le procedure e funzioni principali per la gestione di una pila sono:

procedura ***crea****(rif p: tipoPila)*	crea una pila vuota o svuota una pila esistente;
procedura ***push****(rif p: tipoPila, val n: intero, val elemento: tipo)*	inserisce un elemento sulla cima della pila;
funzione ***vuota****(val p: tipoPila): booleano*	restituisce *vero* se la pila è vuota;
procedura ***pop****(rif p: tipoPila, rif elemento: tipo)*	rimuove un elemento dalla cima della pila;
funzione ***top****(p: tipoPila): tipo*	restituisce l'elemento sulla cima della pila, senza rimuoverlo.

I/O	p	tipoPila	è la pila usata

```
procedura crea(rif p)
    p.cima ← 0
fine
```

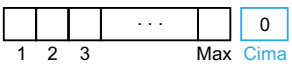


Figura 8.1 *Creazione della pila*

I/O	p	tipoPila	è la pila usata
I	n	intero	numero massimo di elementi dell'array usato per la pila
I	elemento	tipo	valore da inserire nella pila

```
procedura push (rif p, val n, val elemento)
    se (p.cima+1) = n allora
        scrivi "Pila piena"
    altrimenti
        p.cima ← p.cima+1
        p.pila[p.cima] ← elemento
    fine se
fine
```

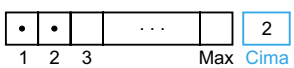


Figura 8.2 *Inserimento di due elementi*

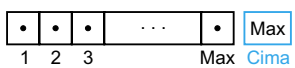


Figura 8.3 *Pila piena*

O	vuota	booleano	ha valore *vero* se la pila non contiene elementi
I	p	tipoPila	è la pila usata

```
funzione vuota(val p)
    vuota ← vero
    se p.cima > 0 allora
        vuota ← falso
    fine se
fine
```

I/O	p	tipoPila	è la pila usata
O	elemento	tipo	valore estratto dalla pila

```
procedura pop(rif p, rif elemento)
    se vuota(p) allora
        scrivi "Pila vuota"
    altrimenti
        elemento ← p.pila[p.cima]
        p.cima ← p.cima-1
    fine se
fine
```

O	top	tipo	valore presente sulla cima della pila
I	p	tipoPila	è la pila usata

```
funzione top(val p)
    se vuota(p) allora
        scrivi "Pila vuota"
    altrimenti
        top ← p.pila[p.cima]
    fine se
fine
```

Variabili usate nel programma principale:

p1	pila
v	valore di un elemento della pila

```
#include <iostream>
using namespace std;

const int Max = 10;

struct tipoPila{
    char pila[Max];
    int cima;
};//tipoPila

void crea(tipoPila& p);

void push(tipoPila& p, int n, char elemento);

bool vuota(const tipoPila& p);

void pop(tipoPila& p, char& elemento);

char top(const tipoPila& p);

void crea(tipoPila& p){
    p.cima = -1;
}//crea

void push(tipoPila& p, int n, char elemento){
    if((p.cima + 1) == n)
        cerr << "Pila piena" << endl;
    else
        p.pila[++p.cima] = elemento;
}//push

bool vuota(const tipoPila& p){
    return !(p.cima >= 0);
}//vuota

void pop(tipoPila& p, char& elemento){
    if(vuota(p))
        cerr << "Pila vuota" << endl;
    else
        elemento = p.pila[p.cima--];
}//pop

char top(const tipoPila& p){
    if(vuota(p))
        cerr << "Pila vuota" << endl;
    else
        return p.pila[p.cima];
}//top
```

Si può provare a utilizzare la pila definita per inserire una serie di parole e stamparle in ordine inverso.

```
int main(void){
    tipoPila p1;
```

```
    char v;
    crea(p1);
    do{
        cout << "Inserire un valore ";
        cin >> v;
        push(p1, Max, v);
    }while(v != '.');
    while(!vuota(p1)){
        pop(p1, v);
        cout << v << endl;
    }//while
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 8.4 *Esecuzione dell'esempio*

8.4 CODE

La **coda** è un caso particolare di lista lineare in cui gli elementi vengono inseriti da un estremo ed estratti dall'altro.

Il criterio utilizzato per la gestione di una coda è il criterio **FIFO** (First In First Out, il primo che entra è il primo che esce), cioè viene estratto per primo l'elemento che è stato inserito per primo.

Si può pensare ad una coda di persone davanti ad uno sportello che vengono servite nell'ordine in cui sono arrivate, o alle macchine in coda ad un semaforo che passano in ordine di arrivo.

Realizzazione di una coda con un array

Per realizzare la coda si può utilizzare un **array** per contenere gli elementi ed una variabile per indicare la **testa** della coda (l'ultimo elemento inserito).
Oltre alla creazione della pila, si devono definire le operazioni di **inserimento** in fondo alla coda ed **estrazione** dalla testa della coda.
Poiché l'array ha una dimensione prefissata, bisogna gestire il caso in cui sia pieno: bisogna fissare un limite massimo per il numero di elementi che possono essere inseriti e la procedura *inserisci* deve segnalare quando l'array è pieno.

Se si vuole fare in modo che il primo elemento della coda coincida sempre col primo elemento dell'array, bisogna shiftare a sinistra tutti gli elementi ogni volta che il primo viene estratto dalla coda.
Conviene usare l'**array** in modo **circolare**: una variabile (*primo*) indica il primo elemento della coda (quello che verrebbe estratto da una operazione *estrai*) e una variabile (*libero*) indica il primo elemento dell'array in cui è possibile inserire un elemento con *inserisci*.

Quando la coda è vuota si ha che:

```
primo = libero
```

quando la coda è piena si ha che:

```
restoDi((libero+1)/dimensione) = primo
```

Un elemento resta sempre vuoto altrimenti sarebbe difficile distinguere quando la coda è vuota e quando invece è piena.

La coda può essere implementata con un record che contiene sia l'array degli elementi della coda che le variabili utilizzate per la testa e per indicare il primo elemento libero.

Esempio 8.2

Gestione di una coda con un array.

Costanti
max: dimensione dell'array utilizzato per simulare la coda.

Tipi
dimensione: tipo dell'indice dell'array (valori da 1 a *max*).
tipoCoda: record formato da
    coda: array usato per simulare la coda;
    primo: testa della coda (all'inizio vale 0);
    libero: primo elemento libero dell'array.
tipo: tipo degli elementi gestiti dalla coda.

Basta cambiare la definizione del tipo *tipo* per inserire nella coda elementi di qualsiasi tipo (ma tutti omogenei tra loro).

Le procedure e funzioni principali per la gestione di una coda sono:

procedura ***crea****(rif c: tipoCoda)*	crea una coda vuota o svuota una coda esistente;
procedura ***inserisci****(rif c: tipoCoda, val n: intero, val elemento: tipo)*	inserisce un elemento in fondo alla coda;
funzione ***vuota****(val c: tipoCoda): booleano*	restituisce *vero* se la coda è vuota;
procedura ***estrai****(rif c: tipoCoda, val n: intero, rif elemento: tipo)*	rimuove il primo elemento della coda;
funzione ***primo****(val c: tipoCoda): tipo*	restituisce il primo elemento della coda, senza rimuoverlo.

I/O	c	tipoCoda è la coda usata

```
procedura crea(rif c)
    c.primo ← 1
    c.libero ← 1
fine
```

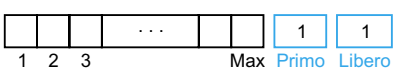


Figura 8.5 *Creazione della coda*

I/O	c	tipoCoda	è la coda usata
I	n	intero	numero massimo di elementi dell'array usato per la coda
I	elemento	tipo	valore da inserire nella coda

```
procedura inserisci (rif c, val n, val elemento)
    se (restoDi(c.libero+1)/n) = c.primo allora
        scrivi "Coda piena"
    altrimenti
        c.coda[c.libero] ← elemento
        c.libero ← restoDi((c.libero+1)/n)
    fine se
fine
```

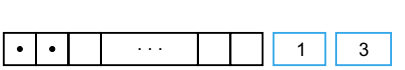


Figura 8.6 *Inserimento di due elementi*

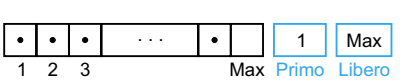


Figura 8.7 *Coda piena*

O	vuota	booleano	ha valore *vero* se la coda non contiene elementi
I	c	tipoCoda	è la coda usata

```
funzione vuota(val c)
    vuota ← vero
    se c.primo ≠ c.libero allora
        vuota ← falso
    fine se
fine
```

I/O	c	tipoCoda	è la coda usata
I	n	intero	numero massimo di elementi dell'array usato per la pila
O	elemento	tipo	valore estratto dalla coda

```
procedura estrai(rif c, val n, rif elemento)
    se vuota(c) allora
        scrivi "Coda vuota"
    altrimenti
        elemento ← c.coda[c.primo]
        c.coda[c.primo] ← ""
        c.primo ← restoDi((c.primo+1)/n
    fine se
fine
```

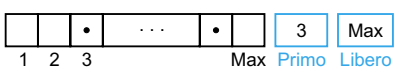


Figura 8.8 *Estrazione di due elementi*

Dopo l'inserimento di altri due elementi la coda appare così:

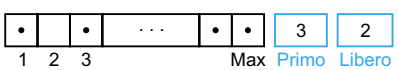


Figura 8.9 *Inserimento di altri due elementi*

O	primo	tipo	valore presente nella testa della coda
I	c	tipoCoda	è la coda usata

```
funzione primo(val c)
    se vuota(c) allora
        scrivi "Coda vuota"
    altrimenti
        primo ← c.coda[c.primo]
    fine se
fine
```

Variabili usate nel programma principale:

c1	coda
v	valore di un elemento della coda

```
#include <iostream>
using namespace std;

const int Max = 10+1;
```

```
struct tipoCoda{
    string coda[Max];
    int primo;
    int libero;
};//tipoCoda

void crea(tipoCoda& c);

void inserisci(tipoCoda& c, int n, string elemento);

bool vuota(const tipoCoda& c);

void estrai(tipoCoda& c, int n, string& elemento);

string primo(const tipoCoda& p);

void crea(tipoCoda& c){
    c.primo = 0;
    c.libero = 0;
}//crea

void inserisci(tipoCoda& c, int n, string elemento){
    if(((c.libero + 1) % n) == c.primo)
        cerr << "Coda piena" << endl;
    else{
        c.coda[c.libero] = elemento;
        c.libero = (c.libero + 1) % n;
    }//else
}//inserisci

bool vuota(const tipoCoda& c){
    return c.primo == c.libero;
}//vuota

void estrai(tipoCoda& c, int n, string& elemento){
    if(vuota(c))
        cerr << "Coda vuota" << endl;
    else{
        elemento = c.coda[c.primo];
        c.coda[c.primo] = "";
        c.primo = (c.primo + 1) % n;
    }//else
}//estrai

string primo(const tipoCoda& c){
    if(vuota(c))
        cerr << "Coda vuota" << endl;
    else
        return c.coda[c.primo];
}//primo
```

Si può provare a utilizzare la coda definita per inserire una serie di parole e stamparle successivamente.

```
int main(void){
    tipoCoda c1;
    string v;
```

```
    crea(c1);
    do{
        cout << "Inserire un valore ";
        cin >> v;
        inserisci(c1, Max, v);
    }while(v != ".");
    while(!vuota(c1)){
        estrai(c1, Max, v);
        cout << v << endl;
    }//while
    fflush(stdin);
    getchar();
    return 0;
}//main
```

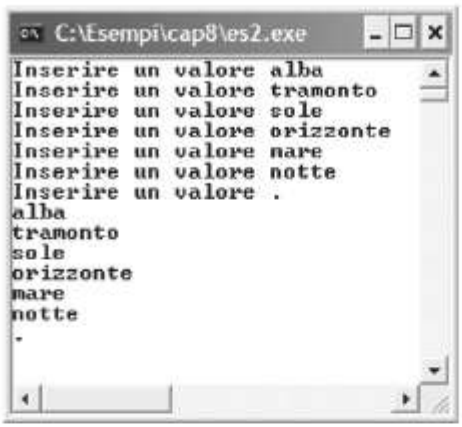


Figura 8.10 *Esecuzione dell'esempio*

8.5 LISTE CONCATENATE

Una **lista concatenata** è una lista in cui si possono inserire o estrarre elementi anche all'interno della lista.

In una lista concatenata unidirezionale gli elementi sono composti ciascuno da due parti: il dato e un valore che indica qual è l'elemento successore (questi valori in pratica sono chiamati **puntatori**, perché puntano ad un altro elemento); l'ultimo elemento non ha successore e quindi contiene un segnale che indica la fine della catena.
L'accesso alla lista è **sequenziale**, cioè per accedere ad un elemento qualsiasi bisogna esaminare tutti quelli che lo precedono.
Bisogna conoscere il primo elemento della catena, poi si passa da un elemento all'altro utilizzando le informazioni date dai puntatori.
È possibile inserire ed eliminare elementi in qualsiasi posizione, modificando i puntatori.

Altre forme di liste concatenate sono:

- la lista concatenata **circolare**, una catena unidirezionale in cui l'ultimo elemento contiene un puntatore al primo elemento;
- la lista concatenata **bidirezionale**, composta da elementi che oltre al dato contengono due puntatori, uno all'elemento precedente e uno al successivo; l'ultimo elemento contiene nel puntatore al successivo un segnale di fine catena; il primo elemento contiene nel puntatore al precedente un segnale di fine catena; anche la catena bidirezionale può essere circolare.

Il dato di ogni elemento di una lista concatenata può essere anche un puntatore ad un'altra lista.

Nota

Operazioni sulle liste concatenate

Tutte le operazioni di ricerca, inserimento o cancellazione di elementi in una lista concatenata vengono svolte sfruttando i **puntatori**.

Ricerca

Per **cercare** un dato in una lista si devono **scorrere** tutti gli elementi; per ogni elemento si confronta la parte contenente il dato con il dato cercato e, se non corrisponde, si passa all'elemento successivo attraverso il puntatore; si continua finché non si trova il dato voluto oppure finché si incontra il puntatore che contiene il segnale di fine della catena.

Inserimento

Per l'**inserimento** di un nuovo dato si devono considerare casi diversi, secondo la posizione in cui si vuole inserire l'elemento: all'inizio della lista, al centro o alla fine.

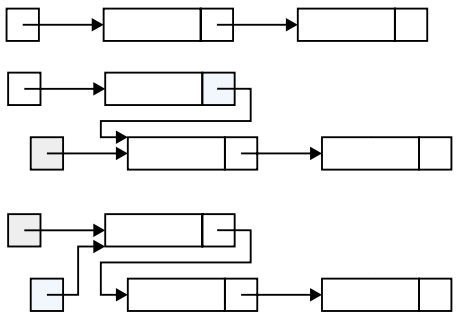


Figura 8.11 *Inserimento come primo elemento*

L'inserimento di un dato come **primo** elemento della lista è piuttosto semplice: si conosce l'indirizzo del primo elemento della lista; questo indirizzo viene inserito nel puntatore del nuovo elemento, mentre l'indirizzo del nuovo elemento inserito diventerà quello del primo elemento.

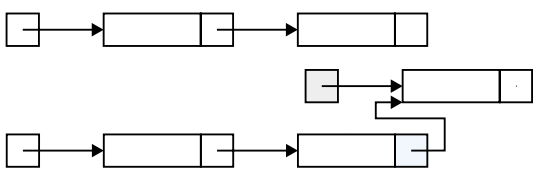


Figura 8.12 *Inserimento in fondo*

Per inserire un dato **in fondo** alla lista bisogna scorrere l'intera lista fino ad arrivare all'ultimo elemento e sostituire il segnale di fine lista con il puntatore al nuovo elemento.

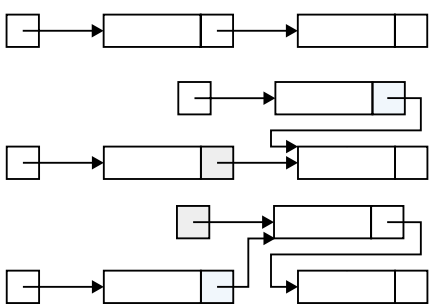


Figura 8.13 *Inserimento al centro*

Per inserire un dato **al centro** della lista (per esempio per creare una lista in cui i dati sono ordinati) bisogna scorrere l'intera lista fino a trovare l'elemento che deve precedere il nuovo; il puntatore di questo elemento deve essere copiato nel puntatore dell'elemento che si vuole inserire e poi modificato in modo che punti al nuovo elemento.

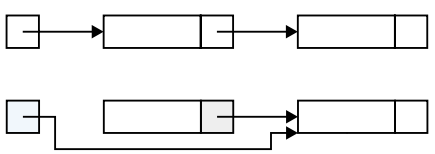


Figura 8.14 *Estrazione del primo elemento*

Estrazione di un elemento

Per **cancellare** un elemento basta fare in modo che non faccia più parte della catena, cioè che scorrendo i puntatori venga saltato; bisogna quindi fare in modo che il puntatore dell'elemento che precede quello da cancellare lo salti, puntando invece all'elemento successivo.

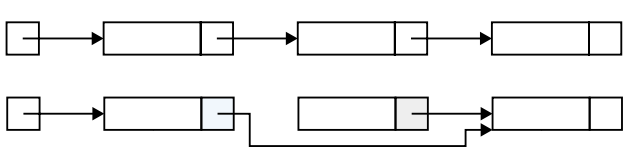


Figura 8.15 *Estrazione di un elemento interno*

Realizzazione di una lista concatenata con array

Per realizzare una lista concatenata si possono utilizzare due **array paralleli**, uno per i dati e uno per i puntatori; gli elementi corrispondenti dei due array costituiscono le due parti (dato e puntatore) di un elemento della lista.
Le operazioni sulla lista vengono effettuate sugli array, aggiornando la parte che contiene il dato e la parte che contiene il puntatore.
Invece di due array è possibile usare un **array di record**, in cui ogni record è costituito dal dato e dal puntatore al successivo elemento dell'array.

8.6 ALLOCAZIONE DINAMICA DELLA MEMORIA

Le strutture di dati descritte usano un numero **variabile** di dati.
L'implementazione con un array richiede di fissare in anticipo la dimensione dell'array e quindi limita il numero di dati utilizzabili.

Per poter realizzare effettivamente delle strutture **dinamiche** con un numero variabile di dati bisogna usare l'allocazione dinamica dei dati.
Con l'**allocazione dinamica** le variabili vengono allocate in memoria solo quando servono (**variabili dinamiche**).

Le variabili dinamiche vengono allocate in una zona della memoria detta **heap**; si possono creare variabili dinamiche finché c'è spazio nello heap; quando non c'è più spazio si verifica un errore di esecuzione. Le locazioni di memoria usate nello heap non sono necessariamente contigue.

Le variabili dinamiche, dato che devono essere allocate dinamicamente, non possono avere un nome che le identifichi e non sono dichiarate. Per poter usare le variabili dinamiche si usa il **tipo puntatore**.

I puntatori

Per riferirsi a una variabile dinamica si usa una variabile di tipo puntatore (che viene chiamata semplicemente **puntatore**).

> Un puntatore contiene l'indirizzo di memoria in cui è memorizzata la variabile dinamica associata (si dice che **punta** alla variabile e la variabile viene chiamata **variabile puntata**).

Ogni puntatore può fare riferimento solo a variabili di un tipo specifico: per riferirsi a una variabile dinamica bisogna usare un puntatore di tipo adatto.

I puntatori

C++ 8.1

Dichiarazione di puntatori

Per dichiarare un puntatore si usa il simbolo * chiamato **operatore di indirezione**.

```
tipoDominio *p;
```

definisce un puntatore *p* che punta a variabili di tipo *tipoDominio*.

Il **tipo del dominio** deve essere specificato con un identificatore di tipo e può essere qualsiasi tipo semplice o strutturato.

Nota | Spesso si usano puntatori a strutture (e array).

La variabile *nomePuntatore* può puntare solo a variabili di tipo *tipoDominio*.

Per assegnare a un puntatore l'indirizzo di una variabile si usa l'operatore & chiamato **operatore di indirizzo**.

```
tipoDominio *p;
tipoDominio nomeVar;
```

```
p = &nomeVar;
```
assegna a *p* l'indirizzo della variabile *nomeVar*

È possibile riunire definizione e assegnazione di un puntatore in un'unica istruzione.

```
char c;
char *p1 = &c;

int n;
int *p2 = &n;
```

p1 è una variabile che contiene l'indirizzo di memoria in cui è memorizzata la variabile *c* di tipo *char*;
p2 è una una variabile che contiene l'indirizzo di memoria in cui è memorizzata la variabile *n* di tipo *int*.

I puntatori sono **tipizzati**, cioè il puntatore è legato al tipo del suo dominio: un puntatore può fare riferimento solo a oggetti di questo particolare tipo di dati.
In questo modo il tipo dell'oggetto puntato è noto a tempo di compilazione.

Una variabile puntatore contiene un **indirizzo di memoria**, che indica dove è memorizzato il dato di un'altra variabile.
Un puntatore è memorizzato come un valore a 32 bit senza segno nei processori a 32 bit e come un valore a 64 bit senza segno nei processori a 64 bit.
L'operatore **sizeof** restituisce l'occupazione di memoria in byte di una variabile o di un tipo passati come argomento.

```
cout << "Dimensione di un int " << sizeof(n);
cout << "Dimensione di un int " << sizeof(int);
```

Uso di puntatori

Il puntatore contiene l'indirizzo della variabile e quindi si può usare per fare riferimento alla variabile.
Per **riferirsi** alla variabile, se *p* è il puntatore, si usa la notazione ***p**; si dice anche che si sta **dereferenziando** il puntatore.
Una variabile dinamica indirizzata in questo modo può essere usata come qualsiasi altra variabile.

```
int *pn, numero = 100;
pn = &numero;
cout << "numero=" << numero << endl;    //stampa 100
cout << "numero=" << *pn << endl;       //stampa 100
*pn = *pn / 2;
cout << "numero=" << numero << endl;    //stampa 50
cout << "numero=" << *pn << endl;       //stampa 50
```

Si può **liberare** l'area di memoria associata a un puntatore *p* con l'operatore *delete*:

`delete p;` oppure `delete(p);`

l'area di memoria viene resa disponibile e il puntatore *p* non punta più a niente.

Nota

In C l'area di memoria viene rilasciata richiamando la procedura *free(p)* definita nel file di intestazione *<cstdlib>*.

Per indicare esplicitamente che un puntatore non punta a niente si può assegnare la costante predefinita ***NULL***.

Dopo aver assegnato *NULL* a un puntatore *p* o eseguito *delete p*, *p* è **indefinito**.
Non si deve mai usare **p* se *p* è indefinito perché si possono causare **errori di esecuzione**.

Puntatore ad un elemento di un array

Un puntatore può riferirsi ad un singolo elemento di un array:

```
nomePuntatore = &nomeArray[indiceElemento];
```

in particolare con l'istruzione:

```
nomePuntatore = nomeArray;
```

nomePuntatore punta al primo elemento (di indice 0) dell'array ed è equivalente all'istruzione:

```
nomePuntatore = &nomeArray[0];
```

```
long a[]={1000, 2000, 3000, 500, 1500, 600, 400};
long *pa0, *pa3;

pa0 = a;
```

pa0 punta al primo elemento dell'array *a*, quindi l'istruzione:

```
cout << *pa0 - 100 << endl;
```

produce la stampa del valore 900.

```
pa3 = &a[3];
```

pa3 punta al quarto elemento dell'array *a*, quindi l'istruzione:

```
cout << *pa3 << endl;
```

produce la stampa del valore 500.

Esempio 8.3

Uso di puntatori.

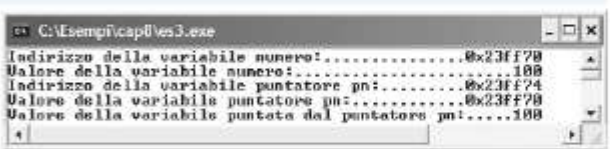


Figura 8.16 *Esecuzione dell'esempio*

```
#include <iostream>
#include <iomanip>
using namespace std;

int main(int argc, char *argv[]){
    int *pn, numero = 100;
    cout << "Indirizzo della variabile numero:"
         << setw(23) << setfill('.') << &numero << endl;
    cout << "Valore della variabile numero:"
         << setw(26) << numero << endl;
    pn = &numero;
    cout << "Indirizzo della variabile puntatore pn:"
         << setw(17) << &pn << endl;
    cout << "Valore della variabile puntatore pn:"
         << setw(20) << pn << endl;
    cout << "Valore della variabile puntata dal puntatore pn:"
         << setw(8) << *pn << endl;
   ...
}//main
```

Operazioni sui puntatori

Sui puntatori sono permesse le operazioni di **assegnazione**, **sottrazione** e **confronto**.
I puntatori devono essere dello stesso tipo.
È inoltre possibile sommare o sottrarre a un puntatore un numero intero.

Assegnazione

A un puntatore può essere assegnata:

- la costante *NULL* per indicare che il puntatore non punta a nulla;
- un altro puntatore dello stesso tipo; dopo l'assegnazione i due puntatori puntano alla stessa variabile.

```
p = q;
```
**p* e **q* sono la stessa variabile e non due variabili uguali;
p e *q* sono la stessa variabile; se si esegue *delete p*, anche *q* diventa indefinito.

Nota

Alcuni compilatori (come il GNU C++), quando si hanno più puntatori alla stessa variabile e si esegue un *delete* su uno di essi, non rendono indefiniti tutti gli altri ma annullano il valore della variabile puntata finché non si esegue un *delete* su tutti i puntatori.

Addizione con un intero

Siano *p* un puntatore che punta ad un qualsiasi elemento di un array *dati* e *n* un qualsiasi intero (positivo o negativo).

```
p = &dati[i];
```

p + n è ancora un **puntatore** che punta all'elemento dell'array *dati* di **indice *i + n***.

Nota

Se *n + i* non è più un valore valido per l'indice dell'array si verifica un errore di esecuzione.

Poiché *p + n* è un puntatore, per accedere al contenuto della variabile puntata si usa l'operatore *, quindi ***(p+n)*** è il valore di *dati[i+n]*.

In particolare le notazioni ***p++***, ***++p***, ***p--***, ***--p*** permettono di riferirsi al successivo o precedente elemento dell'array.

Nota

È sempre possibile utilizzare l'operatore * dando vita a notazioni come ***--p** oppure ***p++**.

Sottrazione tra due puntatori

Siano *p* e *q* due puntatori che puntano a due elementi qualsiasi dello stesso array *dati*.

```
p = &dati[i];
q = &dati[j];
```

p - q è un **numero intero** risultato della sottrazione ***i - j***.

La sottrazione tra due puntatori permette di capire quale dei due elementi puntati ha indice inferiore: se la sottrazione è positiva allora il secondo puntatore punta ad un elemento di indice maggiore rispetto a quello puntato dal primo, cioè *i > j*.

Confronto

Il risultato di un confronto tra due puntatori è un valore **booleano**.

Con l'operatore di uguaglianza == (e con il rispettivo operatore di disuguaglianza !=) si possono controllare le condizioni:

- se il valore di un puntatore è *NULL*;
- se due puntatori sono uguali, cioè puntano alla stessa variabile;

Sono lecite le seguenti condizioni:

```
p == NULL    p != NULL    p == q    p != q
```

Si possono usare anche gli operatori <, <=, >, >=.

Nota

In realtà l'esito di un confronto tra due puntatori è una conseguenza della differenza tra i due: se $p < q$ dà esito *true* (valore positivo) allora $p - q < 0$, quindi p punta ad un elemento di indice inferiore rispetto a quello puntato da q.

Esempio 8.4

Operazioni sui puntatori.

```
#include <iostream>
using namespace std;

int main(void){
    char a[]="SalveMondo";
    char *p, *q;
    p = &a[4];
    q = a;
    cout << "Valore: " << *p << endl;
    cout << "Valore: " << *(p+3) << endl;
    cout << "Valore: " << *(p-2) << endl;
    cout << "Valore: " << *++q << endl;
    cout << "Valore: " << *--p << endl;
    cout << "Differenza: " << p-q << endl;
    cout << "Confronto: " << (p == q) << endl;
    //le parentesi sono necessarie
    getchar();
    return 0;
}//main
```

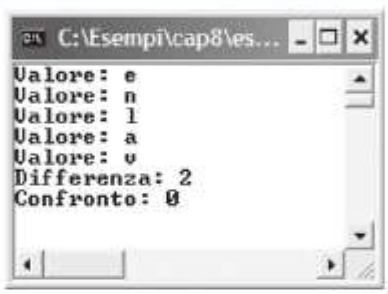


Figura 8.17 *Esecuzione dell'esempio*

Puntatori e array

Il legame tra puntatori e array è molto stretto.
È possibile utilizzare la notazione e l'aritmetica dei puntatori per accedere a un array; il nome dell'array è infatti un indirizzo.

La dichiarazione di una funzione avente come parametro un array può utilizzare un puntatore.

```
Tipo1 nomeFunzione(Tipo2 *nomeA, Tipo3 dimensione)
```

equivale a:

```
Tipo1 nomeFunzione(Tipo2 nomeA[], Tipo3 dimensione)
```

Nel corpo della funzione si può utilizzare un puntatore per scorrere l'array: all'inizio il puntatore assume l'indirizzo del primo elemento, successivamente viene incrementato (operatore ++); la conclusione del ciclo avviene con un confronto tra il puntatore e la somma tra l'indirizzo del primo elemento e la dimensione dell'array.

```
for(Tipo2 *p = nomeA; p < (nomeA + dimensione); p++)
```

Se l'array è un parametro di input e usa il qualificatore **const**, è necessario un cast per inizializzare correttamente il puntatore in quanto l'array è di tipo *const Tipo2** e il puntatore (il cui contenuto deve cambiare) è di *Tipo2**.

```
Tipo1 nomeFunzione(const Tipo2 *nomeA, Tipo3 dimensione){
    for(Tipo2 *p = (Tipo2 *)nomeA; p < (nomeA + dimensione); p++)
...
```

L'indice del ciclo (come numero intero) è dato dalla differenza tra il puntatore e l'indirizzo del primo elemento, mentre il valore dell'elemento dell'array è dato dall'operatore * applicato a *p*.

Esempio 8.5

Modifica della procedura *scriviArray()* dell'esempio 7.4 con la sintassi dei puntatori.

```
#include <iostream>
using namespace std;

void scriviArray(const int *dati, short dim){
    cout << "Gli elementi dell'array sono:" << endl;
    for(int *p = (int *)dati; p < (dati+dim); p++)
        cout << "Elemento di indice "
             << p-dati << ": " << *p << endl;
}//scriviArray

int main(void){
    int numeri[]={1000, 2000, 3000, 500, 1500, 600, 400, 200,700};
    scriviArray(numeri, 9);
    getchar();
    return 0;
}//main
```

Puntatori come valori di ritorno

Una funzione può avere come valore di ritorno un puntatore; ciò è particolarmente utile quando il puntatore punta ad un array, oppure ad una struttura.

Una funzione avente un puntatore come valore di ritorno ha la forma:

```
Tipo* nomeFunzione(listaParametri);
```

Puntatori a strutture

Un puntatore a struttura è una variabile che può contenere l'indirizzo di una variabile struttura. È possibile dichiarare un puntatore a qualsiasi struttura.

Si può inizializzare un puntatore affinché punti a una variabile di tipo struttura con l'operatore *&*.

```
struct nomeStruttura{
    Tipo campo1;
    ...
    Tipo campoN;
};//nomeStruttura

nomeStruttura nomeVar, *nomePuntS;
nomePuntS = &nomeVar;
```

I campi della struttura sono accessibili sia attraverso la variabile di tipo struttura con l'operatore . (punto), sia attraverso il puntatore con l'operatore **->** (**freccia**):

```
nomeVar.nomeCampo;
nomePuntS -> nomeCampo;
```

Se una struttura deve essere passata come parametro a una funzione e occupa molta memoria (perché ha tanti campi, o array o altre strutture), è preferibile passare invece della struttura un puntatore, cioè effettuare un passaggio per indirizzo.

L'OPERATORE NEW

Si può allocare in memoria una struttura mediante l'operatore ***new***.

```
nomeStruttura *nomePuntS;
nomePuntS = new nomeStruttura;
```

alloca lo spazio in memoria necessario alla struttura *nomeStruttura*.

L'accesso ai campi si ha con l'operatore -> sul puntatore *nomePuntS*.

ESEMPIO 8.6

Allocazione di punti attraverso puntatori.

Figura 8.18 *Esecuzione dell'esempio*

```
#include <iostream>
using namespace std;

struct Punto {
    float x, y;
};//Punto

//ritorna un puntatore di tipo Punto
Punto* creaPunto(float ascissa, float ordinata);

int main(void){
    float cx, cy;
    Punto *primoP, *simmO;
    cout << "Inserisci l'ascissa: ";
    cin >> cx;
    cout << "Inserisci l'ordinata: ";
    cin >> cy;
    primoP = creaPunto(cx, cy);
    simmO = creaPunto(-cx, -cy);
    cout << "Punto: (" << primoP->x << ","
         << primoP->y <<")\n";
    cout << "Simmetrico rispetto O: ("
         << simmO->x << "," << simmO->y <<")\n";
    fflush(stdin);
    getchar();
    return 0;
}//main

Punto* creaPunto(float ascissa, float ordinata){
    Punto* p = new Punto;
    p->x = ascissa;
    p->y = ordinata;
    return p; //puntatore all'area allocata
}//creaPunto
```

Allocazione dinamica della memoria in C

In C non esiste l'operatore *new* ma è comunque possibile allocare un'area di memoria costituita da locazioni contigue attraverso una delle funzioni ***malloc()***, ***calloc()*** e ***realloc()***. Queste funzioni, definite nel file di libreria ***<cstdlib>***, restituiscono un puntatore all'area di memoria allocata.
Il puntatore universale ***void**** permette di allocare un'area in cui è possibile memorizzare dati di tipo diverso (come interi, reali o strutture).
Il puntatore *void** non è utilizzabile direttamente, quindi diventa necessario un cast per convertirlo nell'effettivo tipo desiderato.

Nota Terminata l'elaborazione, si può rilasciare l'area di memoria con la procedura ***free()***.

Se l'allocazione non va a buon fine viene restituito il valore *NULL*.

La procedura assert

```
void assert(int condizione);
```

La procedura ***assert()*** definita nel file di libreria ***<cassert>*** controlla la condizione che riceve come parametro; se è falsa termina l'esecuzione comunicando un opportuno messaggio.

```
assert(pArea != NULL);
```
controlla il valore del puntatore *pArea* e se è *NULL* (cioè la condizione specificata è falsa) termina l'esecuzione.

La funzione malloc

```
void* malloc(size_t numeroByte);
```

size_t è un tipo numerico intero positivo e *numeroByte* indica la dimensione in byte della memoria da allocare, solitamente specificata con l'operatore ***sizeof*** nella forma *sizeof(tipo)*;

```
Punto* uno = static_cast<Punto *>(malloc(sizeof(Punto)));
```
oppure
```
Punto* uno = (Punto *)malloc(sizeof(Punto));
```
alloca lo spazio di memoria necessario per la struttura *Punto* e assegna il suo indirizzo al puntatore *uno*.

Esempio 8.7

Uso della funzione *malloc()*. Modifica della funzione *creaPunto()* dell'esempio 8.6.

```
Punto* creaPunto(float ascissa, float ordinata){
    Punto* p = static_cast<Punto*>(malloc(sizeof(Punto)));
    assert(p != NULL);
    p->x = ascissa;
    p->y = ordinata;
    return p;
}//creaPunto
```

La funzione calloc

La funzione ***calloc()*** può essere vista come un'evoluzione della funzione *malloc()*, in quanto permette l'allocazione di array, cioè di un certo numero di elementi dello stesso tipo.

```
void* calloc(size_t numeroElementi, size_t numeroByte);
```

Il puntatore restituito individua il primo elemento dell'array; gli elementi successivi possono essere individuati con l'aritmetica dei puntatori.

Nota

La funzione *malloc()* può essere vista come un caso particolare della funzione *calloc()*, con *numeroElementi = 1*.

```
Punto* tanti = static_cast<Punto *>(calloc(n, sizeof(Punto)));
```
alloca un array di *n* elementi di tipo *Punto* individuato dal puntatore *tanti*.
```
float ascissa3 = (tanti + 2)->x;
```
memorizza in *ascissa3* l'ascissa del terzo punto dell'array.

Esempio 8.8

Allocazione e stampa di un array di punti.

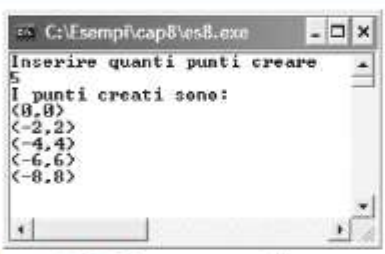


Figura 8.19 *Esecuzione dell'esempio*

```
#include <iostream>
#include <cstdlib>
#include <cassert>
using namespace std;

struct Punto {
    float x, y;
};//Punto

Punto* creaPunti(int);
void mostraPunti(const Punto*, int);

int main(void){
    int n;
    Punto *punti;
    cout << "Inserire quanti punti creare\n";
    cin >> n;
    punti = creaPunti(n);
    cout << "I punti creati sono:\n";
    mostraPunti(punti, n);
    fflush(stdin);
    getchar();
    return 0;
}//main

Punto* creaPunti(int num){
    Punto* p = static_cast<Punto*>(calloc(num, sizeof(Punto)));
    assert(p != NULL);
    for(Punto *q = p; q < (p+num); q++){
        q->x = 2 * (p-q);
        q->y = 2 * (q-p);
    }//for
    return p;
}//creaPunto

void mostraPunti(const Punto* nPunti, int num){
    for(int i = 0; i < num; i++)
        cout << "(" << (nPunti+i)->x << ","
             << (nPunti+i)->y << ")\n";
}//mostraPunti
```

LA FUNZIONE REALLOC

La funzione ***realloc()*** permette di riallocare un'area di memoria precedentemente allocata, sia per aumentarne la dimensione, che per diminuirla.

```
void* realloc(void* pAreaPrec, size_t numeroByte)
```

pAreaPrec è il puntatore all'area precedentemente allocata.

Poiché l'allocazione avviene utilizzando locazioni contigue, se alla richiesta di ulteriore memoria le locazioni vicine sono occupate, la funzione *realloc()* alloca altrove l'area complessiva e vi trasferisce il contenuto precedente. Riallocare ripetutamente può rallentare l'esecuzione.

```
Punto* tanti = (Punto*)realloc(tanti, 3*n*sizeof(Punto));
```

alloca un'area di memoria di dimensione tripla rispetto a quella allocata precedentemente; anche questa area sarà identificata dal puntatore *tanti*.

ESEMPIO 8.9

Modifica dell'esempio 8.8 usando la funzione *realloc()*.
Sono riportate solo le funzioni *creaPunti()* e *main()*.
Il parametro della funzione *creaPunti()* può cambiare all'interno della funzione quindi è passato per indirizzo.

```
Punto* creaPunti(int *nu){
    int num = *nu;
    Punto* p = static_cast<Punto*>(calloc(num, sizeof(Punto)));
    assert(p != NULL);
    for(Punto *q = p; q < (p+num); q++){
        q->x = 2 * (p-q);
        q->y = 2 * (q-p);
    }//for
    char scelta;
    cout << "Vuoi raddoppiare il numero di punti (s/n)?\n";
    cin >> scelta;
    if(scelta == 's'){
       p = static_cast<Punto*>(realloc(p, 2*num*sizeof(Punto)));
        assert(p != NULL);
        for(Punto *q = (p+num); q < (p+2*num); q++){
            q->x = 3 * (p-q);
            q->y = 3 * (q-p);
        }//for
        *nu = 2*num;
    }//if
    return p;
}//creaPunto

int main(void){
    int n;
    Punto *punti;
    cout << "Inserire quanti punti creare\n";
    cin >> n;
    punti = creaPunti(&n);
    cout << "I punti creati sono:\n";
```

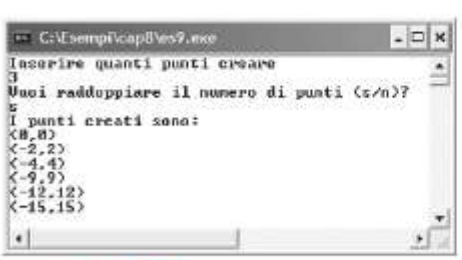


Figura 8.20 *Esecuzione dell'esempio*

```
    mostraPunti(punti, n);
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Strutture dinamiche

Per creare strutture dinamiche si usano **strutture** che contengono uno o più **puntatori** ad altre strutture, creando una definizione ricorsiva.

```
struct Nodo {
    char elemento;
    Nodo *successivo;
};//Nodo
```

Alcuni compilatori non accettano questa definizione: in tal caso si definisce un tipo per la struttura con *typedef*:

```
typedef struct Nodo {
    char elemento;
    struct Nodo *successivo;
} Nodo;
```

In entrambi i casi è possibile dichiarare sia variabili che puntatori del tipo della struttura:

```
Nodo *pNodo, varNodo;

pNodo -> elemento;    equivale a    (*pNodo).elemento;
pNodo -> successivo -> elemento;
pNodo -> successivo -> successivo -> elemento;
```

Puntatori passati per indirizzo

A volte in una funzione è necessario modificare un puntatore passato come parametro.

Se il puntatore è passato come parametro con la notazione

```
tipo1 nomeFunzione(tipo2* nomePunt, ...);
```

viene passato **per valore**.

Per definizione di puntatore, *nomePunt* contiene un indirizzo che permette di accedere alla variabile puntata e di modificarla; se si modifica il puntatore però, la modifica viene perduta all'uscita della funzione.

Nota

Affinché sia possibile modificare il puntatore, è necessario passarlo per indirizzo; la dichiarazione della funzione diventa:

```
tipo1 nomeFunzione(tipo2**nomePunt, ...);
```

dove il primo dei due asterischi indica che vi è un passaggio per indirizzo, il secondo che *nomePunt* è un puntatore.
All'interno della funzione, poiché è un parametro passato per indirizzo, il puntatore è identificato dalla notazione **nomePunt*;

La presenza di spazi tra il tipo, i due asterischi e il puntatore è ininfluente.

Nota

Nella chiamata della funzione si deve usare l'operatore & per indicare l'indirizzo del puntatore.

```
void push(Nodo **cima, Nodo *nuovo);
```

in questa dichiarazione *cima* è un puntatore passato per indirizzo e *nuovo* è un puntatore passato per valore. Una possibile chiamata è:

```
push(&n1, n2);
```

8.7 REALIZZAZIONE DI UNA PILA CON I PUNTATORI

ESEMPIO 8.10

Gestione di una pila con i puntatori.

Tipi

pNodo: puntatore a un nodo.
nodo: record formato da
    elemento: tipo valore di un elemento della pila;
    successivo: pNodo puntatore all'elemento successivo.
tipo: tipo degli elementi gestiti dalla pila.

Basta cambiare la definizione del tipo *tipo* per inserire nella pila elementi di qualsiasi tipo.

Le procedure e funzioni principali per la gestione di una pila sono:

procedura ***crea****(rif p: tipoPila)*	crea una pila vuota o svuota una pila esistente;
procedura ***push****(rif p: tipoPila, val n: intero, val elemento: tipo)*	inserisce un elemento sulla cima della pila;
funzione ***vuota****(val p: tipoPila): booleano*	restituisce *vero* se la pila è vuota;
procedura ***pop****(rif p: tipoPila, rif elemento: tipo)*	rimuove un elemento dalla cima della pila;
funzione ***top****(val p: tipoPila): tipo*	restituisce l'elemento sulla cima della pila, senza rimuoverlo.

I/O	cima	pNodo	puntatore alla cima della pila

```
procedura crea(rif cima)
    cima ← niente
fine
```

I/O	cima	pNodo	puntatore alla cima della pila
I	nuovo	pNodo	puntatore al nuovo elemento da inserire

```
procedura push (rif cima, val nuovo)
    se cima = niente allora
        cima ← nuovo
    altrimenti
        nuovo^.successivo ← cima
        cima ← nuovo
    fine se
fine
```

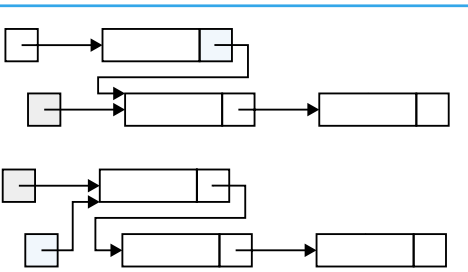


Figura 8.21 *Inserimento di un elemento*

O	vuota	booleano	ha valore *vero* se la pila non contiene elementi
I	cima	pNodo	puntatore alla cima della pila

```
funzione vuota(val cima)
    vuota ← vero
    se cima ≠ niente allora
        vuota ← falso
    fine se
fine
```

I/O	cima	pNodo	puntatore alla cima della pila
O	elemento	tipo	valore estratto dalla pila

```
procedura pop(rif cima, rif elemento)
    se vuota(cima) allora
        scrivi "Pila vuota"
    altrimenti
        elemento ← cima^.elemento
        cima ← cima^.successivo
    fine se
fine
```

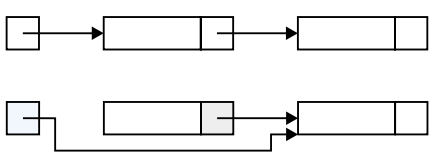


Figura 8.22 *Estrazione di un elemento*

O	top	tipo	valore presente sulla cima della pila
I	cima	pNodo	puntatore alla cima della pila

```
funzione top(val cima)
    se vuota(cima) allora
        scrivi "Pila vuota"
    altrimenti
        top ← cima^.elemento
    fine se
fine
```

Variabili usate nel programma principale:

p1	pNodo	puntatore alla cima della pila usata
v	tipo	valore di un elemento della pila

La funzione *creaNodo* prende il valore da inserire nella pila e crea un nodo contenente il valore e un puntatore a niente; restituisce il puntatore al nodo creato.

O	creaNodo	pNodo	puntatore al nodo creato
I	v	tipo	valore da inserire nella pila

```
funzione creaNodo(val v)
    n  pNodo   puntatore a un nodo generico
    new(n)
    n^.elemento ← v
    n^.successivo ← niente
    creaNodo ← n
fine
```

```
#include <iostream>
using namespace std;
```

```
struct Nodo {
    char elemento;
    Nodo *successivo;
};//Nodo

Nodo* creaNodo(char v);

void crea(Nodo **cima);
void push(Nodo **cima, Nodo *nuovo);
bool vuota(Nodo *cima);
void pop(Nodo **cima, char& elemento);
char top(Nodo *cima);

Nodo* creaNodo(char v){
    Nodo* n = new Nodo;
    n->elemento = v;
    n->successivo = NULL;
    return n;
}//creaNodo

void crea(Nodo **cima){
    *cima = NULL;
}//crea

char top(Nodo *cima){
    if(vuota(cima))
        cerr << "Pila vuota" << endl;
    else
        return cima->elemento;
}//top

void push(Nodo **cima, Nodo *nuovo){
    if(*cima == NULL)
        *cima = nuovo;
    else{
        nuovo->successivo = *cima;
        *cima = nuovo;
    }//else
}//push

bool vuota(Nodo *cima){
    return cima == NULL;
}//vuota

void pop(Nodo **cima, char& elemento){
    if(vuota(*cima))
        cerr << "Pila vuota" << endl;
    else{
        elemento = (*cima)->elemento;
        *cima = (*cima)->successivo;
    }//else
}//pop

int main(int argc, char *argv[]){
    Nodo *p1;
    char v;
    crea(&p1);
```

Figura 8.23 *Esecuzione dell'esempio*

```
    do{
        cout << "Inserire un valore ";
        cin >> v;
        push(&p1, creaNodo(v));
    }while(v != '.');
    while(!vuota(p1)){
        pop(&p1, v);
        cout << v << endl;
    }//while
    fflush(stdin);
    getchar();
    return 0;
}//main
```

8.8 REALIZZAZIONE DI UNA CODA CON I PUNTATORI

ESEMPIO 8.11

Gestione di una coda con i puntatori.

Tipi
pNodo: puntatore a un nodo;
nodo: record formato da
elemento: *tipo* valore di un elemento della pila;
successivo: *pNodo* puntatore all'elemento successivo.
tipo: tipo degli elementi gestiti dalla pila.

Basta cambiare la definizione del tipo *tipo* per inserire nella pila elementi di qualsiasi tipo.

Le procedure e funzioni principali per la gestione di una coda sono:

procedura ***crea****(rif c: tipoCoda)*	crea una coda vuota o svuota una coda esistente;
procedura ***inserisci****(rif c: tipoCoda, val n: intero, val elemento: tipo);*	inserisce un elemento in fondo alla coda;
funzione ***vuota****(val c: tipoCoda): booleano*	restituisce *vero* se la coda è vuota;
procedura ***estrai****(rif c: tipoCoda, val n: intero, rif elemento: tipo)*	rimuove il primo elemento della coda;
funzione ***primo****(val c: tipoCoda): tipo*	restituisce il primo elemento della coda, senza rimuoverlo.

I/O			
I/O	testa	pNodo	puntatore alla testa della coda

```
procedura crea(rif testa)
    testa ← niente
fine
```

I/O			
I/O	testa	pNodo	puntatore alla testa della coda
I	nuovo	pNodo	puntatore all'elemento da inserire nella coda

```
procedura inserisci (rif testa, val nuovo)
```

succ	pNodo	puntatore temporaneo

```
    se testa = niente allora
```

```
        testa ← nuovo
    altrimenti
        succ ← testa
        mentre succ^.successivo ≠ niente
            succ ← succ^.successivo
        fine mentre
        succ^.successivo ← nuovo
    fine se
fine
```

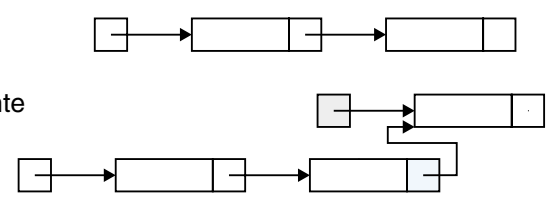


Figura 8.24 *Inserimento di un elemento*

O	vuota	booleano	ha valore *vero* se la coda non contiene elementi
I	testa	pNodo	puntatore alla testa della coda

```
funzione vuota(val testa)
    vuota ← vero
    se testa ≠ niente allora
        vuota ← falso
    fine se
fine
```

I/O	testa	pNodo	puntatore alla testa della coda
O	elemento	tipo	valore estratto dalla coda

```
procedura estrai(rif testa, rif elemento)
    se vuota(testa) allora
        scrivi "Coda vuota"
    altrimenti
        elemento ← testa^.elemento
        testa ← testa^.successivo
    fine se
fine
```

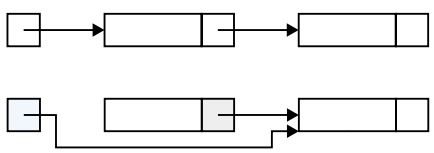


Figura 8.25 *Estrazione di un elemento*

O	primo	tipo	valore presente nella testa della coda
I	testa	pNodo	puntatore alla testa della coda

```
funzione primo(val testa)
    se vuota(testa) allora
        scrivi "Coda vuota"
    altrimenti
        primo ← testa^.elemento
    fine se
fine
```

Variabili usate nel programma principale:

c1	pNodo	puntatore alla testa della coda usata
v	tipo	valore di un elemento della coda

La funzione *creaNodo* prende il valore da inserire nella coda e crea un nodo contenente il valore e un puntatore a niente; restituisce il puntatore al nodo creato.

O	creaNodo	pNodo	puntatore al nodo creato
I	v	tipo	valore da inserire nella coda

```
funzione creaNodo(val v)
   n   pNodo    puntatore a un nodo generico
    new(n)
    n^.elemento ← v
    n^.successivo ← niente
    creaNodo ← n
fine
```

```
#include <iostream>
using namespace std;

struct Nodo {
    char elemento;
    Nodo *successivo;
};//Nodo

Nodo* creaNodo(char v);

void crea(Nodo **testa);
void inserisci(Nodo **testa, Nodo *nuovo);
bool vuota(Nodo *testa);
void estrai(Nodo **testa, char& elemento);
char primo(Nodo *testa);

Nodo* creaNodo(char v){
    Nodo* n = new Nodo;
    n->elemento = v;
    n->successivo = NULL;
    return n;
}//creaNodo

void crea(Nodo **testa){
    *testa = NULL;
}//crea

char primo(Nodo *testa){
    if(vuota(testa))
        cerr << "Coda vuota" << endl;
    else
        return testa->elemento;
}//primo

void inserisci(Nodo **testa, Nodo *nuovo){
    Nodo* succ;
    if(*testa == NULL)
        *testa = nuovo;
    else{
        succ = *testa;
        while(succ->successivo != NULL)
            succ = succ->successivo;
        succ->successivo = nuovo;
    }//else
}//inserisci

bool vuota(Nodo *testa){
```

```
    return testa == NULL;
}//vuota

void estrai(Nodo **testa, char& elemento){
    if(vuota(*testa))
        cerr << "Coda vuota" << endl;
    else{
        elemento = (*testa)->elemento;
        *testa = (*testa)->successivo;
    }//else
}//estrai

int main(int argc, char *argv[]){
    Nodo *c1;
    char v;
    crea(&c1);
    do{
        cout << "Inserire un valore ";
        cin >> v;
        inserisci(&c1, creaNodo(v));
    }while(v != '.');
    while(!vuota(c1)){
        estrai(&c1, v);
        cout << v << endl;
    }//while
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 8.26 *Esecuzione dell'esempio*

8.9 REALIZZAZIONE DI UNA LISTA CONCATENATA CON I PUNTATORI

Per realizzare una lista concatenata si può partire definendo un record *nodo* che contiene un valore e un riferimento al *nodo* successivo della lista.

Per gestire la lista concatenata serve un riferimento alla testa della lista e procedure e funzioni per la gestione della lista.

ESEMPIO 8.12

Gestione di una lista concatenata con i puntatori.

Tipi
pNodo: puntatore a un nodo.
nodo: record formato da
  elemento: *tipo* valore di un elemento della pila;
  successivo: *pNodo* puntatore all'elemento successivo.
tipo: tipo degli elementi gestiti dalla pila.

Basta cambiare la definizione del tipo *tipo* per inserire nella lista elementi di qualsiasi tipo.

Le procedure e funzioni principali per la gestione di una lista concatenata sono:

procedura ***crea****(rif testa: pNodo)* crea una lista vuota o svuota una lista esistente;
procedura ***inserisciInTesta****(rif testa: pNodo, val nuovo: pNodo)*
inserisce un elemento in testa alla lista;
procedura ***inserisciInCoda****(rif testa: pNodo, val nuovo: pNodo)*
inserisce un elemento in fondo alla lista;
funzione ***cerca****(val testa: pNodo, val valore: tipo): pNodo* cerca un oggetto nella lista e restituisce il riferimento al nodo in cui è contenuto;
procedura ***inserisciDopo****(rif testa: pNodo, val nuovo: pNodo, val valore: tipo)*
inserisce un elemento nella lista dopo il nodo che contiene un certo valore;
procedura ***stampa****(val testa: pNodo)* stampa tutti gli elementi contenuti nella lista.

I/O	testa	pNodo	puntatore alla testa della lista

```
procedura crea(rif testa)
    testa ← niente
fine
```

I/O	testa	pNodo	puntatore alla testa della lista
I	nuovo	pNodo	puntatore all'elemento da inserire nella lista

```
procedura inserisciInTesta(rif testa, val nuovo)
    se testa = niente allora
        testa ← nuovo
    altrimenti
        nuovo^.successivo ← testa
        testa ← nuovo
    fine se
fine
```

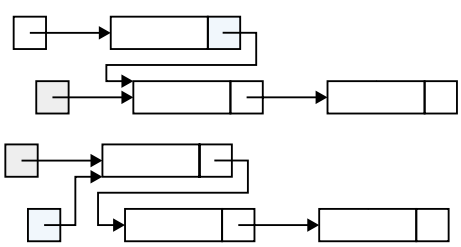


Figura 8.27 *Inserimento di un elemento in testa*

I/O	testa	pNodo	puntatore alla testa della lista
I	nuovo	pNodo	puntatore all'elemento da inserire nella lista

```
procedura inserisciInCoda(rif testa, val nuovo)
    succ      pNodo    puntatore temporaneo
    se testa = niente allora
        testa ← nuovo
    altrimenti
        succ ← testa
        mentre succ^.successivo ≠ niente
            succ ← succ^.successivo
        fine mentre
        succ^.successivo ← nuovo
    fine se
fine
```

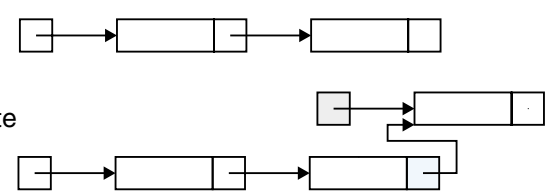


Figura 8.28 *Inserimento di un elemento in fondo*

O	cerca	pNodo	puntatore al nodo della lista contenente il valore cercato
I	testa	pNodo	puntatore alla testa della lista
I	valore	tipo	valore da cercare nella lista

```
funzione cerca(val testa, al valore)
    succ      pNodo    puntatore temporaneo
      succ ← testa
      mentre (succ ≠ niente) and (succ^.elemento ≠ valore)
            succ ← succ^.successivo
      fine mentre
  cerca ← succ
fine
```

I/O	testa	pNodo	puntatore alla testa della lista
I	nuovo	pNodo	puntatore all'elemento da inserire nella lista
I	valore	tipo	valore di riferimento

```
procedura inserisciDopo(rif testa, val nuovo, val valore)
    n   pNodo    puntatore temporaneo
      n ← cerca(testa, valore)
      se n ≠ niente allora
            nuovo^.successivo ← n^.successivo
            n^.successivo ← nuovo
      altrimenti
            inserisciInCoda(testa, nuovo)
      fine se
fine
```

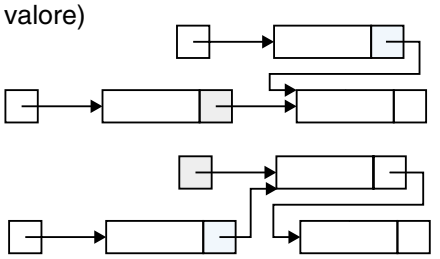


Figura 8.29 *Inserimento di un elemento dopo quello che contiene un certo valore*

I	testa	pNodo	puntatore alla testa della lista

```
procedura stampa(val testa)
    succ      pNodo    puntatore temporaneo
      succ ← testa
      mentre succ ≠ niente
            writeln succ^.elemento
            succ ← succ^.successivo
      fine mentre
fine
```

Variabili usate nel programma principale:

lis1	pNodo	puntatore alla testa della lista usata
scelta	char	operazione da effettuare

La funzione *creaNodo* prende il valore da inserire nella lista e crea un nodo contenente il valore e un puntatore a niente; restituisce il puntatore al nodo creato.

O	creaNodo	pNodo	puntatore al nodo creato
I	v	tipo	valore da inserire nella lista

```
funzione creaNodo(val v)
    n   pNodo    puntatore a un nodo generico
      new(n)
      n^.elemento ← v
```

```
    n^.successivo ← niente
    creaNodo ← n
fine
```

Le altre procedure realizzano operazioni di prova.

```
#include <iostream>
using namespace std;

struct Nodo {
    string elemento;
    Nodo *successivo;
};//Nodo

Nodo* creaNodo(string v);
void crea(Nodo **testa);
void inserisciInTesta(Nodo **testa, Nodo *nuovo);
void inserisciInCoda(Nodo **testa, Nodo *nuovo);
bool vuota(Nodo *testa);
void estrai(Nodo **testa, string& elemento);
Nodo* cerca(Nodo *testa, string valore);
void inserisciDopo(Nodo **testa, Nodo *nuovo, string valore);
void stampa(Nodo* testa);
void provaCrea(Nodo **lis);
void provaCerca(Nodo* lis);
void provaInserisciInTesta(Nodo **lis);
void provaInserisciDopo(Nodo **lis);

Nodo* creaNodo(string v){
    Nodo* n = new Nodo;
    n->elemento = v;
    n->successivo = NULL;
    return n;
}//creaNodo

void crea(Nodo **testa){
    *testa = NULL;
}//crea

Nodo* cerca(Nodo *testa, string valore){
    Nodo* succ = testa;
    while((succ != NULL) && (succ->elemento != valore))
        succ = succ->successivo;
    return succ;
}//cerca

void inserisciDopo(Nodo **testa, Nodo *nuovo, string valore){
    Nodo* n = cerca(*testa, valore);
    if(n != NULL){
        nuovo->successivo = n->successivo;
        n->successivo = nuovo;
    }//if
    else
        inserisciInCoda(testa, nuovo);
}//inserisciDopo
```

```
void inserisciInTesta(Nodo **testa, Nodo *nuovo){
    Nodo* succ;
    if(*testa == NULL)
        *testa = nuovo;
    else{
        nuovo->successivo = *testa;
        *testa = nuovo;
    }//else
}//inserisciInTesta
void inserisciInCoda(Nodo **testa, Nodo *nuovo){
    Nodo* succ;
    if(*testa == NULL)
        *testa = nuovo;
    else{
        succ = *testa;
        while(succ->successivo != NULL)
            succ = succ->successivo;
        succ->successivo = nuovo;
    }//else
}//inserisciInCoda
void stampa(Nodo* testa){
    Nodo* succ = testa;
    while(succ != NULL){
        cout << succ->elemento << endl;
        succ = succ->successivo;
    }//while
}//stampa
void provaCrea(Nodo **lis){
    string v;
    cout << "Inserisci una parola alla volta, . per terminare\n";
    do{
        cin >> v;
        inserisciInCoda(lis, creaNodo(v));
    }while(v != ".");
}//provaCrea
void provaCerca(Nodo* lis){
    Nodo* n;
    string v;
    cout << "Inserisci la parola da cercare\n";
    cin >> v;
    n = cerca(lis, v);
    if(n != NULL)
        cout << n->elemento << endl;
    else
        cout << "Non trovato\n";
}//provaCerca
void provaInserisciInCoda(Nodo **lis){
    string v;
    cout << "Inserisci una parola\n";
    cin >> v;
```

```
    inserisciInCoda(lis, creaNodo(v));
}//provaInserisciInCoda

void provaInserisciInTesta(Nodo **lis){
    string v;
    cout << "Inserisci una parola\n";
    cin >> v;
    inserisciInTesta(lis, creaNodo(v));
}//provaInserisciInTesta

void provaInserisciDopo(Nodo **lis){
    Nodo* n;
    string v;
    cout << "Inserisci una parola\n";
    cin >> v;
    n = creaNodo(v);
    cout << "Dopo quale valore?\n";
    cin >> v;
    inserisciDopo(lis, n, v);
}//provaInserisciDopo
```

Figura 8.30 *Esecuzione dell'esempio*

Per provare la lista concatenata si può creare un menu che permetta di eseguire diverse operazioni sulla lista.

```
int main(int argc, char *argv[]){
    Nodo *lis1;
    char scelta;
    crea(&lis1);
    do{
        cout << "0 - Uscita\n";
        cout << "1 - Crea lista\n";
        cout << "2 - Cerca\n";
        cout << "3 - Inserisci in testa\n";
        cout << "4 - Inserisci in coda\n";
        cout << "5 - Inserisci dopo\n";
        cout << "6 - Stampa\n";
        cin >> scelta;
        switch(scelta){
            case '1':
                provaCrea(&lis1);
                break;
            case '2':
                provaCerca(lis1);
                break;
            case '3':
                provaInserisciInTesta(&lis1);
                break;
            case '4':
                provaInserisciInCoda(&lis1);
                break;
            case '5':
                provaInserisciDopo(&lis1);
                break;
```

```
            case '6':
                stampa(lis1);
        }//switch
    }while(scelta != '0');
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Confronto tra uso di array e di strutture dinamiche con i puntatori

Entrambe usano elementi omogenei.

	Svantaggi	Vantaggi
Array	■ dimensione prefissata; ■ difficoltà di inserimento ed eliminazione, perché ogni operazione richiede lo shift degli altri elementi.	■ facilità di accesso attraverso l'indice.
Strutture dinamiche con puntatori	■ maggiore occupazione di ogni elemento perché contiene anche un puntatore; ■ accesso solo sequenziale scorrendo i puntatori a partire da un valore iniziale esterno alla struttura.	■ dimensioni variabili; ■ facilità di inserimento ed eliminazione dei dati perché basta modificare il puntatore di un elemento.

8.10 GRAFI

Un **grafo** è un insieme di elementi detti **nodi** e di **archi** che congiungono coppie di nodi.

Una successione di archi che congiunge due nodi in un grafo si dice **cammino**.
Si chiama **cammino semplice** un cammino che non passa due volte per uno stesso nodo; un cammino semplice che inizia e finisce sullo stesso nodo si dice **ciclo**.
Un **ciclo Hamiltoniano** è un ciclo che passa per tutti i nodi del grafo una ed una sola volta.
Un grafo si dice **orientato** se per ogni arco è definito un senso in cui l'arco è percorribile; (altrimenti si dice non orientato).

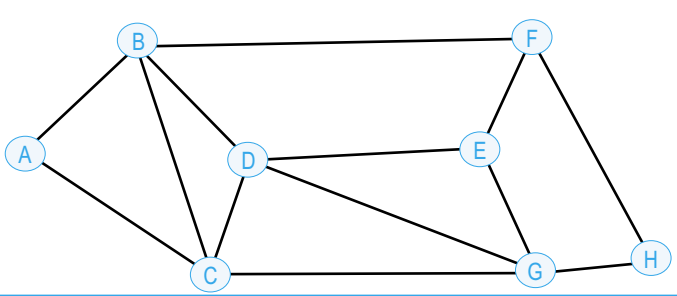


Figura 8.31 *Grafo con 8 nodi e 13 archi, in cui sono presenti molti cicli.*

Uso dei grafi per la soluzione del problema del commesso viaggiatore

Il problema noto come problema del commesso viaggiatore consiste nel cercare un percorso che sia il più **breve** o il più **economico** possibile tra quelli che toccano un certo numero di località.
Il problema può essere rappresentato con un grafo in cui i nodi rappresentano delle località e gli archi le vie di comunicazione con associate le relative distanze o i costi (o tempi) di percorrenza.

Uso dei grafi per la soluzione di problemi di localizzazione di servizi in un territorio

Nella programmazione territoriale si presenta spesso la necessità di dover **localizzare** il luogo più conveniente dove collocare un servizio (un ospedale, una scuola ecc.) in funzione di un certo **obiettivo** che tiene conto dei costi per la realizzazione e per l'utilizzo del servizio, delle distanze che dovranno essere percorse dagli utenti per accedere al servizio ecc.
La regione in cui devono essere localizzati i servizi può essere rappresentata con un grafo, in cui ogni possibile utente coincide con un nodo e le vie di comunicazione con gli archi; agli archi sono associati dei valori che rappresentano le distanze. I servizi possono essere localizzati sia nei nodi che lungo gli archi.

8.11 ALBERI

Un **albero** è un particolare tipo di grafo, connesso e aciclico.

Il grafo di figura 8.31 è connesso ma non è aciclico. **Nota**

Un grafo si dice **connesso** se, presa una coppia qualsiasi di nodi, esiste sempre un cammino che li congiunge.
Un grafo si dice **aciclico** se non esistono cicli, cioè se non ci sono cammini che partono da un nodo e terminano sullo stesso nodo.
Se un grafo gode di queste proprietà, si può disegnare in un modo particolare, in cui un nodo rappresenta la **radice** e tutti gli altri nodi vengono rappresentati lungo dei rami (come appunto un albero, anche se rovesciato, dato che la radice viene rappresentata in alto e i rami verso il basso). I nodi terminali dei rami vengono chiamati **foglie**.
Si dice **livello** di un nodo il numero di archi che è necessario percorrere per raggiungere dal nodo la radice.
Si può anche dire che un nodo è **padre** dei nodi di livello inferiore e che questi sono i suoi **figli**; ogni nodo può avere più figli ma ogni figlio può avere soltanto un padre.

Uso di alberi per la rappresentazione di classificazioni gerarchiche

Gli alberi si prestano ad essere utilizzati in schemi di classificazione e rappresentazione di dati tra i quali c'è un ordinamento gerarchico.

Alberi binari

Un caso particolare di albero è l'**albero binario** in cui ogni nodo ha sempre al **massimo due** figli.

Qualsiasi albero può essere ridisegnato come albero binario seguendo delle opportune regole. **Nota**

Gli alberi binari sono particolarmente importanti; su questi tipi di alberi sono definiti dei metodi per esaminare ordinatamente tutti gli elementi (**visita** dell'albero).

Visita di un albero binario

La **visita** di un albero binario è un metodo per esaminare i nodi dell'albero; ci sono diversi metodi per eseguirla: in ordine anticipato, cominciando dalla radice, posticipato (o differito), cominciando dalle foglie, e simmetrico.

La descrizione dei metodi di visita viene fatta di solito in modo ricorsivo, cioè il metodo viene spiegato facendo ancora riferimento a se stesso; l'insieme di dati a cui si applica la procedura diventa ogni volta più piccolo, limitandosi sempre ad un sottoalbero dell'albero considerato, finché si arriva alla radice, che permette di terminare la descrizione ricorsiva.

Visita in ordine **anticipato**:

- visita la radice;
- visita il sottoalbero sinistro in ordine anticipato;
- visita il sottoalbero destro in ordine anticipato.

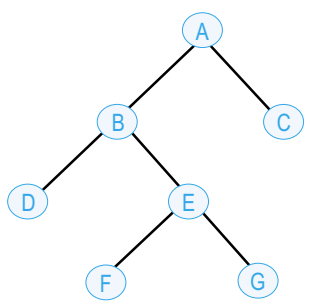
Visita in ordine **posticipato**:

- visita il sottoalbero sinistro in ordine posticipato;
- visita il sottoalbero destro in ordine posticipato;
- visita la radice.

Visita in ordine **simmetrico**:

- visita il sottoalbero sinistro in ordine simmetrico;
- visita la radice;
- visita il sottoalbero destro in ordine simmetrico.

Visita dell'albero



In ordine anticipato:
A, B, D, E, F, G, C

in ordine posticipato:
D, F, G, E, B, C, A

in ordine simmetrico:
D, B, F, E, G, A, C

8.12 REALIZZAZIONE DI UN ALBERO BINARIO CON I PUNTATORI

Per realizzare un albero binario si può partire definendo un record *nodo* che contiene un valore e due riferimenti ai nodi figli, sinistro e destro.

Nota

Il record *nodo* definito in questo modo può essere utilizzato anche per realizzare una lista concatenata bidirezionale, considerando i riferimenti come relativi all'elemento precedente e successivo.

Per gestire un albero serve un riferimento alla radice e procedure e funzioni per la gestione dell'albero.
In un albero binario si possono inserire gli elementi in modo che, per ogni nodo, il sottoalbero sinistro contenga solo elementi minori e il sottoalbero destro contenga solo elementi maggiori; l'attraversamento dell'albero in modo simmetrico restituisce gli elementi in ordine ascendente.

Esempio 8.13
Gestione di un albero binario con i puntatori.

Tipi
pNodo: puntatore a un nodo.
nodo: record formato da
    elemento: tipo valore di un elemento della pila;
    sinistro: pNodo puntatore al sottoalbero sinistro;
    destro: pNodo puntatore al sottoalbero destro.
tipo: tipo degli elementi gestiti dall'albero.

Basta cambiare la definizione del tipo *tipo* per inserire nell'albero elementi di qualsiasi tipo.

Le procedure e funzioni principali per la gestione di un albero binario sono:

procedura ***crea****(rif radice: pNodo)*	crea un albero vuoto o svuota un albero esistente;
procedura ***inserisci****(rif radice: pNodo, val nuovo: pNodo)*	inserisce un elemento nell'albero;
procedura ***inserisciDopo****(val nuovo: pNodo, val n: pNodo)*	inserisce un elemento nell'albero dopo il nodo che contiene un certo valore;
procedura ***stampa****(val n: pNodo)*	stampa il valore dei nodi dell'albero di radice *n*.

I/O	radice	pNodo	puntatore alla radice dell'albero

```
procedura crea(rif radice)
    radice ← niente
fine
```

I	nuovo	pNodo	puntatore all'elemento da inserire nell'albero
I	n	pNodo	puntatore a un elemento, usato per scorrere l'albero

```
procedura inserisciDopo(val nuovo, val n)
    se nuovo^.elemento < n^.elemento allora
        se n^.sinistro=niente allora
            n^.sinistro ← nuovo
        altrimenti
            inserisciDopo(nuovo, n^.sinistro)
        fine se
    fine se
    se nuovo^.elemento > n^.elemento allora
        if n^.destro=niente allora
            n^.destro ← nuovo
        altrimenti
            inserisciDopo(nuovo, n^.destro)
        fine se
    fine se
fine
```

I/O	radice	pNodo	puntatore alla radice dell'albero
I	nuovo	pNodo	puntatore all'elemento da inserire nell'albero

```
procedura inserisci(rif radice, val nuovo)
    se radice=niente allora
        radice ← nuovo
    altrimenti
```

```
            inserisciDopo(nuovo, radice)
        fine se
fine
```

I	n	pNodo	nodo di partenza

```
procedura stampa(n: pNodo);
        se n <> niente allora
                stampa(n^.sinistro)
                scrivi n^.elemento
                stampa(n^.destro)
        fine se
fine
```

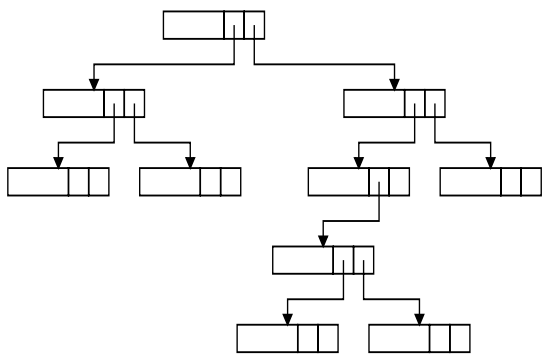


Figura 8.32 *Struttura di un albero binario*

Variabili usate nel programma principale:

a	pNodo	puntatore alla radice dell'albero

La funzione *creaNodo* prende il valore da inserire nell'albero e crea un nodo contenente il valore e un puntatore a niente; restituisce il puntatore al nodo creato.

O	creaNodo	pNodo	puntatore al nodo creato
I	v	tipo	valore da inserire nell'albero

```
funzione creaNodo(val v)
    n   pNodo   puntatore a un nodo generico
        new(n)
        n^.elemento ← v
        n^.sinistro ← niente
        n^.destro ← niente
        creaNodo ← n
fine
```

Si può provare a utilizzare l'albero binario definito per inserire una serie di parole in disordine e ottenerne una stampa ordinata.

```
#include <iostream>
using namespace std;

struct Nodo {
    string elemento;
    Nodo *sinistro;
    Nodo *destro;
};//Nodo

Nodo* creaNodo(string v);
void crea(Nodo **radice);
void inserisciDopo(Nodo *nuovo, Nodo *n);
void inserisci(Nodo **radice, Nodo *nuovo);
void stampa(Nodo* n);

Nodo* creaNodo(string v){
    Nodo* n = new Nodo;
```

```
    n->elemento = v;
    n->sinistro = NULL;
    n->destro = NULL;
    return n;
}//creaNodo

void crea(Nodo **radice){
    *radice = NULL;
}//crea

void inserisciDopo(Nodo *nuovo, Nodo *n){
    if(nuovo->elemento < n->elemento)
        if(n->sinistro == NULL)
            n->sinistro = nuovo;
        else
            inserisciDopo(nuovo, n->sinistro);
    if(nuovo->elemento > n->elemento)
        if(n->destro == NULL)
            n->destro = nuovo;
        else
            inserisciDopo(nuovo, n->destro);
}//inserisciDopo

void inserisci(Nodo **radice, Nodo *nuovo){
    if(*radice == NULL)
        *radice = nuovo;
    else
        inserisciDopo(nuovo, *radice);
}//inserisci

void stampa(Nodo* n){
    if(n != NULL){
        stampa(n->sinistro);
        cout << n->elemento << endl;
        stampa(n->destro);
    }//if
}//stampa

int main(int argc, char *argv[]){
    Nodo *a;
    crea(&a);
    inserisci(&a, creaNodo("G"));
    inserisci(&a, creaNodo("A"));
    inserisci(&a, creaNodo("S"));
    inserisci(&a, creaNodo("H"));
    inserisci(&a, creaNodo("Z"));
    stampa(a);
    getchar();
    return 0;
}//main
```

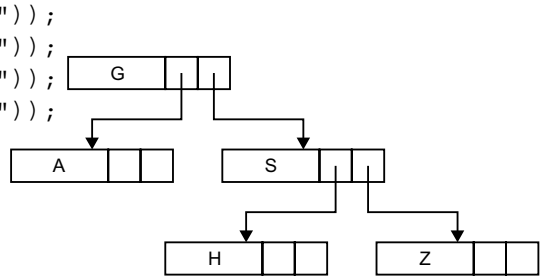


Figura 8.33 *Albero creato dall'esecuzione dell'esempio*

VERIFICA

QUESTIONARIO

1. Che cosa sono le strutture di dati dinamiche?
2. Qual è la differenza tra strutture lineari e non lineari?
3. Che cos'è un tipo di dato astratto (ADT)?
4. Che cos'è un tipo di dato concreto?
5. Che cos'è una lista?
6. Che cos'è una pila?
7. Che cos'è una coda?
8. Che cos'è una lista concatenata?
9. Che cosa significa allocazione dinamica della memoria?
10. Che cos'è lo heap?
11. Che cosa sono i puntatori?
12. Che cos'è un grafo?
13. Che cos'è un albero?
14. Che cos'è un albero binario?

TEST

1 ..... *Individuare la risposta esatta.*

1) Una variabile di tipo puntatore a stringa contiene:
   - ☐ il primo carattere di una stringa;
   - ☐ una stringa;
   - ☐ un numero;
   - ☐ nessuna delle precedenti.

2) L'istruzione *p1 = p2*, con *p1* e *p2* puntatori a stringhe, ha il significato:
   - ☐ *p1* assume il valore della stringa puntata da *p2*;
   - ☐ *p1* assume il valore di *p2*; entrambi puntano alla stessa stringa;
   - ☐ la variabile puntata da *p1* assume il valore della stringa puntata da *p2*;
   - ☐ la variabile puntata da *p1* assume il valore del puntatore *p2*;
   - ☐ nessuno; causa un errore di compilazione.

3) L'istruzione **p1 = p2*, con *p1* e *p2* puntatori a stringhe, ha il significato:
   - ☐ *p1* assume il valore della stringa puntata da *p2*;
   - ☐ *p1* assume il valore di *p2*; entrambi puntano alla stessa stringa;
   - ☐ la variabile puntata da *p1* assume il valore della stringa puntata da *p2*;
   - ☐ la variabile puntata da *p1* assume il valore del puntatore *p2*;
   - ☐ nessuno; causa un errore di compilazione.

4) L'istruzione *p1 = *p2*, con *p1* e *p2* puntatori a stringhe, ha il significato:
   - ☐ *p1* assume il valore della stringa puntata da *p2*;
   - ☐ *p1* assume il valore di *p2*; entrambi puntano alla stessa stringa;
   - ☐ la variabile puntata da *p1* assume il valore della stringa puntata da *p2*;

- ☐ la variabile puntata da *p1* assume il valore del puntatore *p2*;
- ☐ nessuno; causa un errore di compilazione.

5) L'istruzione **p1 = *p2*, con *p1* e *p2* puntatori a stringhe, ha il significato:
- ☐ *p1* assume il valore della stringa puntata da *p2*;
- ☐ *p1* assume il valore di *p2*; entrambi puntano alla stessa stringa;
- ☐ la variabile puntata da *p1* assume il valore della stringa puntata da *p2*;
- ☐ la variabile puntata da *p1* assume il valore del puntatore *p2*;
- ☐ nessuno; causa un errore di compilazione.

6) Le istruzioni *p = &a[4]; p += 5; cout << *p;* con *p* puntatore a int e *a* array di 20 interi permettono la stampa:
- ☐ di *a[9]*;
- ☐ dell'indirizzo di *a[9]*;
- ☐ del contenuto della variabile *p*;
- ☐ del contenuto della variabile *p* incrementato di 5;
- ☐ nessuna stampa; causano un errore di compilazione.

Esercizi

1 ..... Trasferire i dati da una coda realizzata con array a una coda realizzata con una struttura concatenata.

2 ..... Trasferire i dati da una pila a una coda.

3 ..... Aggiungere alla gestione della pila e della coda degli esempi 8.1 - 8.11 procedure e funzioni per:
- ■ contare il numero di elementi;
- ■ invertire gli elementi (il primo elemento deve diventare l'ultimo e l'ultimo il primo);
- ■ concatenare un'altra struttura dello stesso tipo;
- ■ cercare la posizione di un valore;
- ■ fondere due strutture ordinate dello stesso tipo.

4 ..... Aggiungere alla gestione della lista concatenata dell'esempio 8.12 procedure e funzioni per:
- ■ contare il numero di elementi della lista;
- ■ concatenare alla lista un'altra lista;
- ■ restituire il riferimento al nodo che contiene il valore che precede immediatamente un valore specificato;
- ■ inserire un elemento prima di un nodo che contiene un valore specificato;
- ■ eliminare il nodo che contiene un valore specificato;
- ■ sostituire un nuovo valore in un nodo che contiene un valore specificato.

5 ..... Aggiungere alla gestione dell'albero binario dell'esempio 8.13 procedure e funzioni per:
- ■ stampare i nodi in modo anticipato e differito;
- ■ restituire un sottoalbero;
- ■ cancellare un elemento;
- ■ modificare un elemento (e quindi spostare il sottoalbero).

6 ..... Effettuare la visita di un albero senza utilizzare procedure ricorsive.
Si deve ricorrere all'uso di una pila (con la definizione ricorsiva l'uso della pila è implicito). La pila viene usata per contenere i riferimenti ai nodi dell'albero che devono essere stampati in un momento successivo.

PROPOSTE DI LAVORO

1 ..... Realizzare una coda con un array in cui il primo elemento della coda coincida sempre col primo elemento dell'array.

2 ..... Progettare una struttura di dati *DoppiaCoda* che permetta di inserire ed estrarre elementi dai due estremi.

3 ..... Realizzare una implementazione di una lista concatenata bidirezionale.

4 ..... Realizzare una lista concatenata con due array paralleli o con un array di strutture. Gestire all'interno dell'array anche una lista dello spazio libero che colleghi tutti gli elementi liberi dell'array, in modo che gli elementi liberati in seguito a cancellazioni possano essere riutilizzati.
(All'inizio tutto l'array contiene elementi disponibili; bisogna inizializzare i puntatori in modo da formare una catena che colleghi tutti gli elementi ed inizializzare il puntatore alla prima posizione libera in modo che punti al primo elemento dell'array).

5 ..... Rappresentare un albero binario tramite un array di strutture; ogni struttura rappresenta un elemento ed è costituito dal dato, da un valore che indica l'indice corrispondente al figlio di sinistra e da un valore che indica l'indice corrispondente al figlio di destra.

6 ..... Progettare una struttura di dati *Insieme* che permetta almeno di:
- aggiungere un elemento;
- togliere un elemento;
- stampare il contenuto dell'insieme;
- effettuare l'unione con un altro insieme;
- effettuare l'intersezione con un altro insieme;

Realizzare varie implementazioni, per esempio con:
- un array;
- una lista concatenata.

Aggiungere il prodotto cartesiano con un altro insieme.

7 ..... Un grafo può essere rappresentato con una lista di liste in cui ogni elemento della lista principale contiene un elemento del grafo e ciascuna lista secondaria collega tutti i nodi discendenti di ciascun elemento, cioè i nodi a cui si può arrivare con un arco.
Un grafo può essere rappresentato anche con una matrice *n*x*n*, se i nodi sono *n*, dove

a(i, j) = 1 se esiste l'arco che collega il nodo *i* al nodo *j*;
0 in caso contrario.

Matrici di questo tipo vengono anche chiamate mappe di bit. Spesso molti elementi della matrice sono a 0, si tratta cioè di matrici sparse.
Realizzare procedure e funzioni per:
- passare dalla rappresentazione come lista di liste a quella in forma di matrice;
- passare dalla rappresentazione in forma di matrice a quella come lista di liste;
- contare quanti archi escono da un nodo dato, utilizzando la rappresentazione in forma di matrice;
- contare quanti archi escono da un nodo dato, utilizzando la rappresentazione come lista di liste.

8 ..... Inserire in un albero la classificazione di alcuni animali (gli animali rappresentano le foglie dell'albero). Il programma deve:
- stampare usando una opportuna indentazione la classificazione di tutti i dati presenti;
- dato un animale, dire come viene classificato (bisogna mantenere l'elenco delle foglie in una opportuna struttura).

File 9

Prerequisiti

Saper descrivere algoritmi e realizzare programmi in C++ che usano:

- strutture di controllo
- procedure e funzioni
- strutture di dati

Obiettivi

Conoscenze

- File di record e file di testo
- Acesso sequenziale e diretto
- Operazioni sui file

Abilità/capacità

- Descrivere algoritmi che utilizzano file
- Scrivere programmi in C++ che usano file

9.1 I FILE

I **file di dati** permettono di memorizzare dati su memoria di massa.

La maggior parte dei linguaggi permette di gestire

- **file tipizzati** o **strutturati** che contengono sequenze di dati dello stesso tipo semplice o strutturato (di solito record);
- **file di testo** che contengono sequenze di caratteri.

File di record

Un **file di record** è un insieme di **record** logici (suddivisi in campi).
L'operazione di lettura di un file permette di accedere soltanto ad un record per volta; i dati del record letto vengono resi disponibili tramite un'area di memoria riservata al file detta **area record**. L'area record viene utilizzata per eseguire tutte le operazioni di input/output logiche sul file.
Il file può essere pensato come una successione di record; non esiste un limite al numero di record che possono essere contenuti nel file, se non dovuto alla capacità del supporto su cui il file è registrato.

Gestione di file

La gestione di un file comprende delle operazioni che agiscono su tutto il file e delle operazioni che agiscono sul contenuto del file.
Operazioni che agiscono su tutto il file sono la **creazione** del file, l'**apertura** per rendere il file disponibile ad un programma, la **cancellazione** del file, la **copia** e la **ridenominazione**.

Un file definito nel programma è un **file logico**; un file memorizzato sul disco è un **file fisico**; per poter utilizzare file su disco è necessario stabilire un **collegamento** tra file logico e file fisico; serve una procedura di **assegnazione**:

assegna *nomeFile* a *nomeFileFisico*

Apertura dei file

I file sono risorse che i processi chiedono e rilasciano con le operazioni di **apertura** e **chiusura**.

Ogni processo può aprire più file. L'istruzione di **apertura** rende disponibile al programma l'area record associata al file (senza dati). L'apertura avviene specificando il modo; le modalità possibili possono essere lettura (input), scrittura (output), lettura/scrittura (input/output), scrittura in coda al file (append o extend).

crea *nomeFile* (apri in output)
apri *nomeFile* in input
apri *nomeFile* in input/output
apri *nomeFile* in append

Tra l'apertura e la chiusura di un file si ha una sessione di operazioni sulle informazioni contenute nel file. Le operazioni che agiscono sui dati del file sono quelle di **lettura** e **aggiornamento** (inserimento, variazione e cancellazione di dati). Le operazioni ammesse per ogni file usato nel programma dipendono dal tipo di apertura richiesto.

Per ogni file aperto viene assegnato al processo un **puntatore** che viene utilizzato per mantenere la posizione corrente all'interno del file, cioè un riferimento al dato su cui ha effetto l'operazione richiesta.

Le operazioni di lettura e scrittura agiscono sull'elemento a cui si riferisce il puntatore; dopo l'esecuzione dell'operazione il puntatore viene incrementato.

Quando il file non viene più utilizzato deve essere **chiuso** mediante un'istruzione che provvede a rilasciare correttamente il file.

chiudi *nomeFile*

Metodi di accesso

Le modalità di lettura e scrittura dipendono dalla struttura logica e dall'organizzazione fisica del file e dai metodi di accesso.

Secondo la struttura logica le operazioni di lettura e scrittura riguardano sequenze di **caratteri** o **record**.

I **metodi di accesso** possono essere sequenziale o diretto.

Con l'**accesso sequenziale** i dati si possono registrare e rileggere soltanto in modo ordinato, in sequenza; per esaminare l'ultimo dato, per esempio, è necessario scorrere tutti i dati che lo precedono. Si possono solo aggiungere dati in fondo al file.
Il metodo sequenziale è basato su un puntatore alla posizione corrente, modificabile dalle operazioni di lettura e scrittura. L'apertura posiziona il puntatore all'inizio del file.
Una operazione di lettura parte dalla posizione attuale e sposta il puntatore. Il file termina con una etichetta **EOF** che segnala la fine del file.
Analogamente la scrittura parte dalla posizione attuale e sposta il puntatore; se non si è alla fine del file, in caso di scrittura si perdono le informazioni preesistenti.

Operazioni di lettura e scrittura:
leggi da *nomeFile nomeVariabile*
scrivi su *nomeFile nomeVariabile*

Condizione di riconoscimento della fine del file:
fine del file *nomeFile*

Con l'**accesso diretto** (**random** o **casuale**) il file viene considerato in modo simile a un array di dati; ogni dato è individuato da un numero che esprime la posizione relativa all'interno del file; l'accesso diretto permette il posizionamento sul dato desiderato. Per l'accesso diretto oltre alle operazioni di lettura e scrittura si introduce l'operazione di **posizionamento** del puntatore (**seek**).

Posizionamento:
posizionati su *nomeFile* alla posizione *n*

In genere ai file di testo si può accedere solo in modo sequenziale, ai file tipizzati, quindi in particolare ai file di record, si può accedere sia in modo sequenziale che in modo diretto.

Sui **file di caratteri** le operazioni avvengono su un carattere alla volta; l'accesso può essere solo sequenziale; si possono aggiungere caratteri solo alla fine del file.

Sui **file di record** le operazioni vengono effettuate su un record per volta.
Con l'accesso **sequenziale** si può accedere ai record solo in modo ordinato.
Con l'accesso **diretto** ogni record è individuato da un numero che esprime la posizione relativa all'interno del file; è possibile il posizionamento sul record desiderato.
Si possono aggiungere record solo alla fine del file.

9.2 OPERAZIONI SUI FILE

Accesso sequenziale

Scrittura

Il file deve essere aperto in output o in input/output.

Schema generale per la creazione di un file con accesso sequenziale, introducendo i dati da tastiera:

```
inizio
    apri il file in scrittura
    scrivi "Vuoi continuare?"
    leggi risposta
    mentre risposta = "s"
        richiedi dati
        leggi dati
        scrivi dati nel file
        scrivi "Vuoi continuare?"
        leggi risposta
    fine mentre
    chiudi il file
fine
```

Lettura

Il file deve essere aperto in input o in input/output.
Poiché una lettura oltre la fine del file causa un errore di esecuzione, prima della lettura bisogna controllare se è stata raggiunta la fine del file.

Schema generale per la lettura di un file con accesso sequenziale:

```
inizio
    apri il file in lettura
    mentre non è la fine del file
        leggi i dati dal file {se è un file di record viene letto un record}
        elabora i dati
    fine mentre
    chiudi il file
fine
```

Accesso random

Lettura

Il file deve essere aperto in input o in input/output.

Schema generale per la lettura in modo random di un record:

```
inizio
    apri il file in lettura
    scrivi "Vuoi continuare?"
    leggi risposta
    mentre risposta = "s"
        scrivi "Inserire il numero del record"
```

```
            leggi nRecord
            leggi il record nRecord dal file
            se il record esiste
                elabora il record
            altrimenti
                operazioni in caso di record non esistente
            fine se
            scrivi "Vuoi continuare?"
            leggi risposta
        fine mentre
        chiudi il file
fine
```

VARIAZIONE

Il file deve essere aperto in input o in input/output.

Schema per la variazione di record in modo random:

```
inizio
    apri il file in lettura/scrittura
    scrivi "Numero di record da variare (0 per finire)"
    leggi nRecord
    mentre nRecord > 0
        se nRecord > dimensioneFile allora
            scrivi "Record inesistente"
        altrimenti
            posizionati sul record nRecord-1
            leggi il record
            mostra i dati e richiedi i nuovi dati
            leggi dati
            prepara i nuovi dati del record
            posizionati di nuovo sul record nRecord-1
            {dopo la lettura il puntatore si è spostato sul record successivo}
            scrivi il record
        fine se
        scrivi "Numero di record da variare (0 per finire)"
        leggi n
    fine mentre
    chiudi il file
fine
```

INSERIMENTO E CANCELLAZIONE DI RECORD

Per **aggiungere dati** in fondo a un file esistente bisogna posizionarsi in fondo al file (se possibile aprendo il file in modalità append, oppure leggendo tutto il file in modo sequenziale, o sfruttando l'accesso diretto) e poi scrivere i nuovi record.

Per **inserire o cancellare record** all'interno del file bisogna ricopiare il file inserendo o cancellando i record desiderati.

Nei problemi di aggiornamento di file in genere le modifiche da apportare al file originale vengono preparate in un secondo file; elaborando il file originale e il file delle variazioni si può ottenere il nuovo file contenente tutti i dati aggiornati (modalità **batch**).

C++ 9.1

I FILE

In C/C++ si possono usare **file binari**, in particolare **file di record**, a cui è possibile accedere in modo sequenziale o diretto, e **file di testo** a cui è possibile accedere solo in modo sequenziale.

Nota

In C++ è possibile gestire i file associandoli ai flussi con la programmazione a oggetti, come descritto nel capitolo 16.

Per utilizzare un file (binario o di testo) è necessaria la dichiarazione di una **variabile di tipo puntatore a struttura**:

```
FILE* pF;
```

La struttura ***FILE*** è definita nel file di libreria *<stdio.h>* (incluso automaticamente con *<iostream>*); è costituita da sei campi contenenti le informazioni necessarie al sistema operativo per accedere al file.

Nota

Tutte le costanti e le funzioni relative ai file sono definite nel file di libreria *<stdio.h>*.

La variabile *pF* (di tipo **puntatore a *FILE***) viene inizializzata all'apertura del file e identifica all'interno del programma il file fisico presente sul disco. Ogni accesso al file avviene attraverso *pF*.

Nota

Il tipo *FILE** non ammette l'aritmetica dei puntatori; non sono pertanto possibili notazioni quali *pF++* oppure *(pF+intero)*.

C++ 9.2

I FILE BINARI

I **file binari** sono una sequenza di byte. Quando un dato è scritto sul file, viene scritta la rappresentazione (in byte) del dato secondo il suo tipo. Se il dato è una variabile di tipo struttura vengono scritti i valori dei suoi campi così come prevede la rappresentazione del singolo tipo a cui appartiene il campo. Ogni operazione di lettura o scrittura sul file avviene per blocchi di byte.

Per **aprire** un file binario si usa la funzione ***fopen()***:

```
FILE* fopen(const char *nomeFile, const char* modo);
```

che restituisce una variabile di tipo puntatore a *FILE* associato al file *nomeFile* se l'apertura è riuscita, *NULL* altrimenti.
Il parametro *modo* stabilisce la modalità di apertura del file:

`rb`	in sola lettura, per leggere i dati presenti nel file; se il file non esiste l'apertura fallisce;
`wb`	in sola scrittura, per creare un file in cui scrivere i dati; se il file esiste viene svuotato;
`ab`	in append, per scrivere i dati alla fine del file; se il file non esiste viene creato;
`rb+`	in lettura e scrittura, per leggere o scrivere i dati nel file; se il file non esiste l'apertura fallisce;
`wb+`	in lettura e scrittura, per creare il file in cui scrivere o leggere i dati; se il file esiste viene svuotato;
`ab+`	in lettura e append, per leggere i dati o per scrivere i dati alla fine del file; se il file non esiste viene creato;

Per **leggere** dal file si usa la funzione ***fread()*** che restituisce l'effettivo numero di byte letti:

```
size_t fread(void* var, size_t numByte, size_t num, FILE* pF);
```

Significato dei parametri

var	variabile in cui vengono messi i dati letti dal file;
numByte	numero di byte letti dal file; coincide con *sizeof(var)*;

num numero di blocchi di *numByte* letti;
pF variabile che identifica il file su cui avviene la lettura.

Nota

Il valore di ritorno di *fread()* è spesso usato per verificare se è stata raggiunta la fine del file; se è stata raggiunta la fine del file il valore di ritorno è 0 e non viene letto nessun byte.

Per **scrivere** sul file si usa la funzione ***fwrite()*** che restituisce l'effettivo numero di byte scritti:

```
size_t fwrite(const void* var, size_t numByte, size_t num, FILE* pF);
```

Significato dei parametri:

var è la variabile contenente i dati da scrivere sul file;
numByte è il numero di byte che saranno scritti sul file e coincide con *sizeof(var)*;
num è il numero di blocchi di *numByte* scritti;
pF è la variabile che identifica il file su cui avviene la scrittura.

Per **chiudere** il file si usa la funzione ***fclose()*** che restituisce 0 se la chiusura è riuscita, -1 altrimenti:

```
int fclose(FILE* pF);
```

Le funzioni fprintf e fscanf

Se i dati sono di tipo fondamentale per scrivere e leggere il file è possibile utilizzare rispettivamente le funzioni ***fprintf()*** e ***fscanf()***, generalizzazione delle funzioni *printf()* e *scanf()*:

```
int fprintf(FILE* pF, const char* stringaDiControllo, ...);
int fscanf(FILE* pF, const char* stringaDiControllo, ...);
```

Per leggere un dato di tipo *int* dal file identificato da *pFIn* si può scrivere:

```
fscanf(pFIn, "%d", &varInt);
```

Per scrivere un dato di tipo *float* e un dato di tipo *unsigned int* sul file identificato da *pFOut* si può scrivere:

```
fprintf(pFOut, "%f%u", varFloat, varIntPos);
```

Accesso sequenziale

La funzione ***feof()*** permette di verificare se si è raggiunta la fine del file:

```
int feof(FILE* pF);
```

restituisce 0 se il file non è finito, un intero non nullo in caso contrario.

Accesso diretto

L'*accesso diretto* permette di accedere ai dati indicando il **numero relativo** di posizione; i dati sono numerati a partire da 0.
In realtà i dati sono sequenze di byte, quindi la posizione di un dato è formata dal numero di byte complessivo occupato dai dati precedenti.

La funzione ***fseek()*** permette di effettuare un posizionamento sull'elemento desiderato del file restituendo 0 se il posizionamento è riuscito, 1 altrimenti:

```
int fseek(FILE* pF, long offset, int partenza);
```

Significato dei parametri

offset indica lo spostamento relativo (quindi anche negativo) rispetto alla posizione espressa da *partenza*.

partenza indica la posizione da cui effettuare lo spostamento; è espressa dalle costanti:

	Valore	Significato
SEEK_SET	0	inizio del file;
SEEK_CUR	1	posizione corrente;
SEEK_END	2	fine del file

L'operazione di lettura o scrittura successiva agisce sull'elemento specificato.

La funzione ***ftell()*** restituisce il numero di byte presenti nel file fino alla corrente posizione:

```
long ftell(FILE* pF);
```

Nota

Per ottenere il numero di byte totali del file basta posizionarsi alla fine del file con la funzione *fseek()* e richiamare la funzione *ftell()*.

Si può ottenere il numero di dati (di tipo *tipoDati*) presenti nel file (dopo aver effettuato un posizionamento alla fine del file) con la formula:

```
ftell(pF) / sizeof(tipoDati);
```

I FILE DI RECORD

Se i dati del file sono campi di una struttura bisogna definire il tipo della struttura; le operazioni di lettura e scrittura agiscono su variabili di quel tipo.

Per registrare in un file dati relativi a dei libri bisogna definire una struttura per i dati di un libro.

```
struct tipoLibro{
    char titolo[50], autore[50], editore[50];
    int numPagine;
    float prezzo;
};//tipoLibro
```

SCRITTURA CON ACCESSO SEQUENZIALE

ESEMPIO 9.1

Creazione di un file con i dati di alcuni libri.

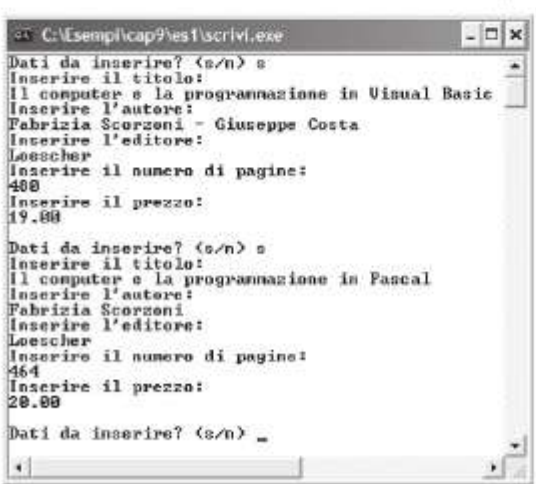
```
#include <iostream>
#include <cstdlib>
using namespace std;

struct tipoLibro{
    char titolo[50], autore[50], editore[50];
    int numPagine;
    float prezzo;
};//tipoLibro

tipoLibro leggiLibro(void){
    tipoLibro libro;
    string s;
    cout << "Inserire il titolo:" << endl;
```

```
    getline(cin, s);
    strcpy(libro.titolo, s.c_str());
    cout << "Inserire l'autore:" << endl;
    getline(cin, s);
    strcpy(libro.autore, s.c_str());
    cout << "Inserire l'editore:" << endl;
    getline(cin, s);
    strcpy(libro.editore, s.c_str());
    cout << "Inserire il numero di pagine:" << endl;
    cin >> libro.numPagine;
    cout << "Inserire il prezzo:" << endl;
    cin >> libro.prezzo;
    return libro;
}//leggiLibro

bool scriviFile(string nomeF, string apert){
    char risposta;
    FILE* pFOut;
    tipoLibro lib;
    if((pFOut=fopen(nomeF.c_str(),apert.c_str())) == NULL)
        return false; //fallita l'apertura
    cout << "Dati da inserire? (s/n) ";
    cin >> risposta;
    while(risposta == 's'){
        fflush(stdin);
        lib = leggiLibro();
        fwrite(&lib, sizeof(tipoLibro), 1, pFOut);
        cout << "\nDati da inserire? (s/n) ";
        cin >> risposta;
    }//while
    fclose(pFOut);
    return true;
}//scriviFile

void scriviNuovo(string nomeF){
    if(!scriviFile(nomeF, "wb"))
        cout << "Errore nell'apertura del file "
             << nomeF << endl;
}//scriviNuovo

int main(void){
    scriviNuovo("libri.dat");
    getchar();
    return 0;
}//main
```

Per aggiungere altri dati si può procedere per esempio in questo modo:

```
void aggiungiFile(string nomeF){
    if(!scriviFile(nomeF, "ab"))
        cout << "Errore "
             << nomeF << endl;
}//aggiungiFile
```

Figura 9.1 *Esecuzione dell'esempio*



Lettura con accesso sequenziale

È importante verificare se si è raggiunta la fine del file prima di una operazione di lettura in modo che non si verifichino errori di esecuzione.

Esempio 9.2

Lettura dei dati dei libri inseriti nell'esempio 9.1.

```
#include <iostream>
#include <iomanip>
using namespace std;

struct tipoLibro{
    char titolo[50], autore[50], editore[50];
    int numPagine;
    float prezzo;
};//tipoLibro

void scriviLibro(const tipoLibro& libro){
    cout << libro.titolo << endl;
    cout << libro.autore << endl;
    cout << libro.editore << endl;
    cout << libro.numPagine << " pagine" << endl;
    cout << setprecision(2) << setiosflags(ios::fixed)
         << libro.prezzo << " euro\n";
}//scriviLibro

bool leggiFile(string nomeF){
    char risposta;
    FILE* pFIn;
    tipoLibro lib;
    if((pFIn=fopen(nomeF.c_str(),"rb")) == NULL)
        return false; //fallita l'apertura in lettura
    cout << "Dati\n";
    while(fread(&lib, sizeof(tipoLibro), 1, pFIn)){
        scriviLibro(lib);
        cout << endl;
    }//while
    fclose(pFIn);
    return true;
}//leggiFile

int main(void){
    string nF = "libri.dat";
    if(!leggiFile(nF))
        cout << "Errore nell'apertura del file "
             << nF << endl;
    getchar();
    return 0;
}//main
```

Figura 9.2 *Esecuzione dell'esempio*

Variazione con accesso diretto

Esempio 9.3

Variazione del prezzo dei libri inseriti negli esempi 9.1 e 9.2.

```
#include <iostream>
#include <iomanip>
using namespace std;

struct tipoLibro{
    char titolo[50], autore[50], editore[50];
    int numPagine;
    float prezzo;
};//tipoLibro

void leggiPrezzo(tipoLibro& lib){
    cout << "Inserire il nuovo prezzo\n";
    cin >> lib.prezzo;
}//leggiPrezzo

void scriviLibro(const tipoLibro& libro){
    cout << libro.titolo << endl;
    cout << libro.autore << endl;
    cout << libro.editore << endl;
    cout << libro.numPagine << " pagine" << endl;
    cout << setprecision(2) << setiosflags(ios::fixed)
         << libro.prezzo << " euro\n";
}//scriviLibro

bool variaFile(string nomeF){
    long n, numRec;
    FILE* pFInOut;
    tipoLibro lib;
    if((pFInOut=fopen(nomeF.c_str(),"rb+")) == NULL)
        return false; //fallita l'apertura in lettura/scrittura
    cout << "Numero del record da variare (0 per finire) ";
    cin >> n;
    while(n > 0){
        fseek(pFInOut, 0, SEEK_END);
        numRec = ftell(pFInOut)/sizeof(tipoLibro);
        if(n > numRec)
            cout << "Record inesistente!\nIl numero"
                 << " massimo e' " << numRec << endl;
        else{
            fseek(pFInOut, (n-1)*sizeof(tipoLibro), SEEK_SET);
            fread(&lib, sizeof(tipoLibro), 1, pFInOut);
            scriviLibro(lib);
            leggiPrezzo(lib);
            fseek(pFInOut, -sizeof(tipoLibro), SEEK_CUR);
            fwrite(&lib, sizeof(tipoLibro), 1, pFInOut);
        }//else
        cout << "Numero del record da variare (0 per finire) ";
        cin >> n;
    }//while
    fclose(pFInOut);
    return true;
}//scriviFile
```

```
int main(void){
    string nF = "libri.dat";
    if(!variaFile(nF))
       cout << "Errore nell'apertura"
        <<" del file " << nF << endl;
    getchar();
    return 0;
}//main
```

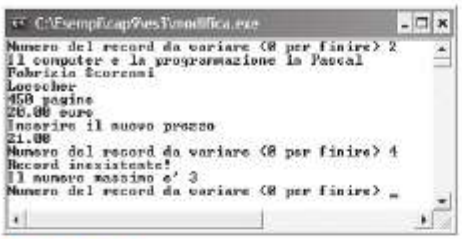


Figura 9.3 *Esecuzione dell'esempio*

C++ 9.3

FILE DI TESTO

Per **aprire** un file di testo si usa la funzione ***fopen()***:

```
FILE* fopen(const char *nomeFile, const char* modo);
```

che restituisce una variabile di tipo puntatore a *FILE* associato al file *nomeFile* se l'apertura è riuscita, *NULL* altrimenti.
Il parametro *modo* stabilisce la modalità di apertura del file:

`r`	in sola lettura, per leggere i dati presenti nel file; se il file non esiste l'apertura fallisce;
`w`	in sola scrittura, per creare il file in cui scrivere i dati; se il file esiste viene svuotato;
`a`	in append, per scrivere i dati alla fine del file; se il file non esiste viene creato;
`r+`	in lettura e scrittura, per leggere o scrivere i dati nel file; se il file non esiste l'apertura fallisce;
`w+`	in lettura e scrittura, per creare il file in cui scrivere o leggere i dati; se il file esiste viene svuotato;
`a+`	in lettura e append, per leggere i dati o per scrivere i dati alla fine del file; se il file non esiste viene creato.

Le istruzioni di **lettura** dal file e **scrittura** sul file sono:

`int fgetc(FILE* pF)`
: lettura di un carattere; restituisce il carattere letto o EOF se è raggiunta la fine del file;

`char* fgets(char* stringa, int dim, FILE* pF)`
: lettura di una riga di dimensione massima *dim*; la riga letta viene posta in *stringa*; restituisce *NULL* se è raggiunta la fine del file;

`int fputc(FILE* pF)`
: scrittura del carattere *c*; restituisce -1 se la scrittura fallisce;

`int fputs(const char* stringa, FILE* pF)`
: scrittura di *stringa;* restituisce -1 se la scrittura fallisce.

È possibile solo l'**accesso sequenziale**.

La funzione ***feof()*** permette di verificare se si è raggiunta la fine del file:

```
int feof(FILE* pF);
```

restituisce 0 se il file non è finito, un intero non nullo in caso contrario.

Per **chiudere** il file si usa la funzione ***fclose()*** che restituisce 0 se la chiusura è riuscita, -1 altrimenti:

```
int fclose(FILE* pF);
```

ESEMPIO 9.4

Creazione di un file di testo.

```
#include <iostream>
using namespace std;
bool scriviFile(string nomeF, string apert){
    string riga;
    FILE* pFOut;
    if((pFOut=fopen(nomeF.c_str(),apert.c_str())) == NULL)
        return false; //fallita l'apertura
    cout << "Scrivi una riga alla volta (. per terminare)\n";
    getline(cin, riga);
    while(riga != "."){
        fputs(riga.c_str(), pFOut);
        fputc('\n', pFOut);
        getline(cin, riga);
    }//while
    fclose(pFOut);
    return true;
}//scriviFile
void scriviNuovo(string nomeF){
    if(!scriviFile(nomeF, "w"))
        cout << "Errore nell'apertura del file "
             << nomeF << endl;
}//scriviNuovo
int main(void){
    scriviNuovo("testo.txt");
    ...
}//main
```

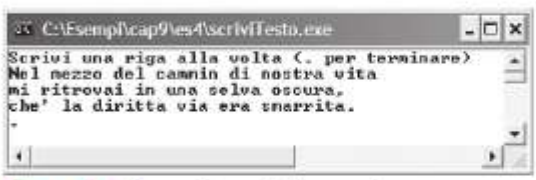


Figura 9.4 *Esecuzione dell'esempio*

ESEMPIO 9.5

Lettura di un file di testo una riga per volta.

```
#include <iostream>
using namespace std;
const int NumCarRiga = 80;
bool leggiFile(string nomeF){
    char riga[NumCarRiga];
    FILE* pFIn;
    if((pFIn=fopen(nomeF.c_str(),"r")) == NULL)
        return false; //fallita l'apertura in lettura
    while(fgets(riga, NumCarRiga, pFIn))
        cout << riga;
    fclose(pFIn);
    return true;
}//leggiFile
int main(void){
    string nF = "testo.txt";
    if(!leggiFile(nF))
        cout << "Errore "
          <<"nell'apertura del file " << nF << endl;
    ...
}//main
```

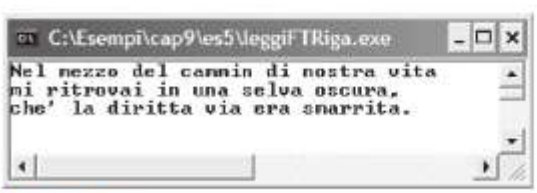


Figura 9.5 *Esecuzione dell'esempio*

Nella lettura per carattere si usa la costante intera ***EOF*** (di valore -1) per verificare se è stata raggiunta la **fine del file**. Per questo il tipo di ritorno di *fgetc()* è *int* e non *char*.

ESEMPIO 9.6

Lettura di un file di testo un carattere per volta. È riportata solo la funzione *leggiFile()*.

```
bool leggiFile(string nomeF){
    int car;
    FILE* pFIn;
    if((pFIn=fopen(nomeF.c_str(),"r")) == NULL)
        return false; //fallita l'apertura in lettura
    while((car=fgetc(pFIn)) != EOF)
        cout << (char)car;
    fclose(pFIn);
    return true;
}//leggiFile
```

VERIFICA

QUESTIONARIO

1. Che cos'è un file binario?
2. Che cos'è un file di testo?
3. In che cosa consiste l'operazione di apertura di un file?
4. A che cosa serve il puntatore al file?
5. In che cosa consiste l'accesso sequenziale?
6. In che cosa consiste l'accesso diretto?

TEST

1 ..... *Indicare se le seguenti affermazioni sono vere o false.*

V	F	
☐	☐	1) Un file binario è sempre una sequenza di dati di tipo struttura.
☐	☐	2) Un file di testo è una sequenza di byte in cui ogni byte rappresenta un carattere.
☐	☐	3) Il puntatore al file viene modificato da ogni operazione di lettura e scrittura.
☐	☐	4) Ai file binari è consentito sia l'accesso sequenziale che random.
☐	☐	5) Ai file di testo è consentito sia l'accesso sequenziale che random.
☐	☐	6) In un file di testo è possibile inserire una nuova riga all'inizio del file.
☐	☐	7) In un file binario è possibile inserire un nuovo dato all'inizio del file.
☐	☐	8) In un file binario è possibile inserire un nuovo dato alla fine del file.
☐	☐	9) In un file di testo è possibile inserire un nuova riga alla fine del file.
☐	☐	10) In un file binario è possibile variare un dato.
☐	☐	11) In un file binario è possibile cancellare un dato.

PROPOSTE DI LAVORO

1 ..... Inserire in un file dei dati anagrafici (nome, indirizzo, data di nascita, professione). Deve essere possibile visualizzare i dati:
- delle persone che abitano in una città richiesta;
- delle persone nate in un certo anno;
- delle persone che svolgono una certa professione;
- delle persone il cui nome inizia con una certa stringa.

A scelta deve essere possibile vedere i dati di una persona alla volta; dopo ogni persona deve essere possibile scegliere se continuare o interrompere la visualizzazione.

2 ..... Gestire una rubrica telefonica in cui ogni record contenga nome, numero di telefono e indirizzo e-mail.
Permettere di:
- aggiungere nuovi dati;
- visualizzare tutti i dati;
- effettuare ricerche per nome;
- modificare il numero di telefono o l'indirizzo e-mail di una persona;
- caricare i dati in un array, effettuare l'ordinamento per nome e memorizzare i dati ordinati.

3 ..... Inserire in un file i dati relativi ai prodotti di una azienda: cod. prodotto, descrizione, prezzo, scorta minima, quantità in magazzino.
Effettuare sul file le seguenti operazioni:
- stampare i dati relativi ai prodotti sottoscorta (i prodotti per cui la quantità in magazzino è minore della scorta minima stabilita), indicando anche qual è la minima quantità da riordinare per ripristinare la scorta minima in magazzino;
- determinare quali sono i prodotti con la scorta di maggiore e minor valore (il valore è dato dalla quantità in magazzino per il prezzo di vendita);
- determinare il valore medio delle scorte dei diversi articoli;
- permettere di variare il prezzo e la quantità in magazzino dei prodotti.

4 ..... Dati due file con i nomi degli studenti di due classi in ordine alfabetico, creare il file fusione che contiene tutti i nomi, ancora in ordine alfabetico.

5 ..... Creare un file per la registrazione dei dati di un'agenda mensile in cui ogni record rappresenta un giorno:
- consentire di registrare un solo impegno per ogni giorno;
- gestire gli impegni di ogni giorno in modo che sia possibile registrare un impegno per ogni ora.

Permettere di memorizzare, variare, annullare o visualizzare gli impegni (tutti o per un periodo richiesto).

6 ..... Realizzare un programma per l'inserimento di dati relativi alla richiesta di iscrizione ad una scuola. I dati devono essere registrati in tre file, ognuno con cognome, nome e indirizzo, in base alla lingua scelta (tra tre possibilità).
Permettere di:
- contare il numero degli iscritti ad una lingua;
- aggiungere altre iscrizioni anche successivamente;
- modificare la lingua scelta; i dati di un iscritto vanno cancellati dal file originale (annullando il record o ricopiando il file, senza il record eliminato) e aggiunti al file corrispondente alla nuova lingua.

7 ..... Dato un file originario (con un codice e altri dati) in ordine di codice e un file variazioni, con il codice, dati dello stesso tipo del file originario e un tipo operazione

(I = inserimento, V = variazione, C= cancellazione), creare un file dello stesso tipo del primo, effettuando tutte le modifiche indicate nel secondo. Eseguire tutti i controlli relativi alle operazioni richieste (per esempio se è richiesto un inserimento controllare che il codice non sia già presente e controllare invece che esista per una cancellazione) e segnalare gli eventuali messaggi di errore.

8 ..... Inserire in un file i dati relativi agli alunni di una scuola (classe e nome); i dati sono ordinati per classe e all'interno della classe in ordine alfabetico.
Deve essere possibile:
- dato il nome di un alunno, stabilire a che classe appartiene;
- data una classe, ottenere l'elenco degli alunni della classe;
- stampare tutti i dati suddivisi per classe (rottura di codice sul campo classe).

9 ..... Gestione del televideo. Le pagine del televideo sono memorizzate in un file in cui il numero di record corrisponde al numero di pagina; ogni record contiene una stringa di caratteri che rappresenta la pagina.
Realizzare un programma che permetta (dopo aver memorizzato le pagine):
- di visualizzare una determinata pagina;
- di avanzare alla pagina successiva;
- di ritornare all'ultima pagina vista.

10 ..... Dato un file di testo che contiene un programma in C++, produrre l'elenco di tutte le variabili usate nel programma, indicando se si tratta di variabili globali o locali (e in tal caso dire a che cosa sono locali).

11 ..... Dato un file di testo, visualizzarne il contenuto in pagine di 10 righe alla volta; dopo ogni pagina deve essere possibile decidere se passare alla pagina successiva, riprendere dall'inizio o interrompere la visualizzazione.

12 ..... Dato un file di testo, ricopiarlo correggendo l'uso degli spazi (non ci devono essere due o più spazi consecutivi, non ci deve essere uno spazio prima di un segno di punteggiatura, ma ci deve essere dopo, se il testo prosegue).

13 ..... Dato un file di testo, determinare il numero delle parole e il numero di frasi, la lunghezza media delle parole e la lunghezza media delle frasi; calcolare la frequenza delle parole che compaiono nel testo e al termine scrivere in un altro file di testo una serie di righe con i risultati (per ogni riga una parola e dopo uno spazio la relativa frequenza).

14 ..... Memorizzare in un file di testo una serie di domande del tipo vero/falso. Ogni domanda comincia ad una nuova riga che contiene nel primo carattere la risposta (V o F). Presentare le domande a una persona e conteggiare le risposte giuste e quelle sbagliate. Deve essere possibile, a scelta, vedere subito per ogni domanda se la risposta data era corretta.

15 ..... Memorizzare in un file di testo una serie di domande a risposta multipla: le righe che contengono le domande iniziano con il numero della risposta esatta (da 1 a 3) e sono seguite dalle tre risposte possibili. Presentare le domande a una persona e conteggiare le risposte giuste e quelle sbagliate. Se la risposta è sbagliata si può permettere un secondo tentativo.

16 ..... Dato un file di testo che contenga in ogni riga il nome di una nazione seguito da uno spazio e dal nome della capitale, deve essere possibile:
- visualizzare i dati, uno alla volta, per studiare i nomi;
- presentare a un utente, uno alla volta, il nome di una nazione, chiedere il nome della capitale e controllare se la risposta è esatta;
- presentare a un utente, uno alla volta, il nome di una capitale, chiedere il nome della nazione e controllare se la risposta è esatta.

Modulo 3

La programmazione a oggetti

10 CLASSI E OGGETTI

Prerequisiti

- Saper compilare ed eseguire applicazioni suddivise in più file
- Conoscere gli elementi di base del linguaggio C++

Obiettivi

Conoscenze

- Vantaggi della programmazione a oggetti
- Concetti fondamentali della programmazione a oggetti: classe e oggetto (istanza), attributi e metodi, stato e comportamento di un oggetto, metodi costruttori, chiamate di metodi come invio di messaggi, overloading, riferimento *this,* incapsulamento e information hiding
- Diagrammi UML per la rappresentazione delle classi
- Concetto di spazio degli stati legale e comportamento corretto di una classe
- Definizione di classi, attributi e metodi con modificatori di accesso
- Suddivisione delle classi in file distinti
- Funzioni friend
- Overloading di operatori
- Asserzioni
- Uso del modificatore *static*
- Definizione di classi interne e friend

Abilità/capacità

- Progettare classi con attributi e metodi
- Rappresentare una classe con un diagramma UML
- Suddividere le classi in file distinti
- Definire e usare funzioni friend, anche per l'overloading di operatori

10.1 OGGETTI E CLASSI

Un **oggetto** è un insieme di dati (**attributi** o proprietà) e di codice (**metodi**); gli attributi definiscono le caratteristiche dell'oggetto, cioè lo **stato**, e i metodi definiscono le operazioni eseguibili sull'oggetto, cioè il **comportamento**.

Ciò che un oggetto può fare è stabilito dal suo comportamento, cioè dai suoi metodi; i metodi possono modificare lo stato dell'oggetto ma l'oggetto conserva lo stato tra una chiamata di metodo e l'altra (cioè mantiene le informazioni per tutta la durata del programma).

Gli oggetti vengono creati in base a una definizione di oggetti dello stesso tipo, chiamata **classe**.

La **classe** è un modello o prototipo che definisce un **tipo** di oggetto, cioè è un modello per tutti gli oggetti dello stesso tipo; definisce la struttura e il comportamento degli oggetti appartenenti alla classe.

Per **usare degli oggetti** di un certo tipo si deve **definire una classe** che descriva gli attributi e i metodi comuni agli oggetti della classe, poi bisogna **creare un oggetto** appartenente alla classe assegnando specifici valori agli attributi, cioè definendo lo stato iniziale dell'oggetto.
Da una classe possono essere generati molti oggetti; ogni oggetto ha la stessa struttura e comportamento della classe, ma ha il proprio stato.

La creazione di un oggetto a partire da una classe si dice **istanziazione** o creazione di un'istanza della classe.
La **classe** è il **codice**, è lo "stampo" per istanziare oggetti.
Un **oggetto** è un'**istanza** (o esemplare) della classe.

Dopo aver creato un oggetto si possono usare o modificare i valori degli **attributi** ed eseguire delle operazioni richiamandone i **metodi**.

Nella programmazione a oggetti un **programma** è costituito da varie classi; un metodo principale crea uno o più oggetti i cui metodi possono a loro volta creare altri oggetti.
Per far fare qualcosa a un oggetto bisogna eseguire una parte del codice dell'oggetto cioè chiamare uno dei suoi metodi.
Gli oggetti interagiscono, comunicano e si scambiano informazioni solo tramite chiamate di metodi.

10.2 VANTAGGI DELLA PROGRAMMAZIONE A OGGETTI

La programmazione a oggetti presenta molti vantaggi rispetto alla programmazione imperativa:

- il **mondo** reale è **orientato agli oggetti**: i programmatori possono progettare software allo stesso modo in cui percepiscono il mondo reale;
- la programmazione orientata agli oggetti distingue nettamente tra **che cosa** deve essere fatto e **come** deve essere fatto; per usare un oggetto basta sapere che cosa fa, cioè quali sono i metodi disponibili e come devono essere richiamati, e non come lo fa, cioè come sono implementati i metodi; l'implementazione è nascosta e potrebbe anche essere modificata senza problemi;
- l'orientamento agli oggetti permette di avere:
  - **riusabilità** del codice e **facilità di manutenzione**: le classi definite si possono riutilizzare, eventualmente estendendole per creare nuove classi (quindi è più facile anche apportare modifiche);

- **robustezza** cioè capacità di continuare l'esecuzione in presenza di errori; i casi anomali vengono gestiti con il meccanismo delle **eccezioni**; ogni volta che si verifica un problema viene lanciata un'eccezione; il codice può intercettare l'eccezione e fornire delle operazioni da eseguire in quel caso;
- **affidabilità**: si può verificare che siano soddisfatte le condizioni per cui un oggetto appartiene a una classe (invarianti di classe).

> Per esempio se una classe *Triangolo* ha come attributi i tre punti che costituiscono i vertici si può verificare nel costruttore che tre punti individuino veramente un triangolo controllando che la somma di due lati sia maggiore del terzo.

10.3 IMPLEMENTAZIONE DI CLASSI E OGGETTI

Dal punto di vista del **codice** una **classe** è composta da variabili (**variabili di istanza** o campi) che rappresentano gli attributi e da **metodi**.

Attributi e **variabili di istanza** in realtà non sempre corrispondono esattamente.
Gli attributi sono le proprietà osservabili dall'esterno, mentre le variabili di istanza sono i valori che l'oggetto memorizza; alcuni attributi possono essere calcolati a partire da variabili di istanza diverse. Nell'uso comune però i termini attributi e variabili di istanza in genere vengono considerati sinonimi.

> Una classe *Cerchio* che ha come attributi le coordinate del centro e il raggio, nell'implementazione potrebbe memorizzare le coordinate del vertice superiore sinistro del quadrato in cui è inscritto il cerchio, invece di quelle del centro.
>
> Una classe *Parallelepipedo* potrebbe memorizzare le misure degli spigoli e calcolare l'attributo volume solo quando richiesto.

I **metodi** sono funzioni che possono essere eseguite sugli oggetti; un metodo definito in una classe cioè può essere eseguito solo su un oggetto di quella classe.

La differenza tra metodo e funzione "classica" è che il metodo è sempre definito in una classe e opera sui dati contenuti nell'oggetto su cui è richiamato. **Nota**

Le classi devono avere anche dei metodi **costruttori** (almeno uno, anche se a volte può essere implicito).
Un metodo costruttore viene richiamato quando viene creata una istanza della classe; di solito il metodo costruttore inizializza le variabili di istanza dell'oggetto.

Gli oggetti mantengono i propri dati in aree di memoria separate: ogni oggetto ha un'area di memoria privata per le sue variabili di istanza.
Per questioni di efficienza però le istanze di una classe condividono le implementazioni dei metodi, cioè si ha una sola copia del codice per tutti gli oggetti della stessa classe.

La **creazione degli oggetti** avviene in fase di esecuzione e gli oggetti creati esistono solo per la durata del programma.

Perciò quando il programma termina si perdono gli oggetti. **Nota**

Quando si crea un oggetto, richiamando il metodo costruttore, si ottiene un **riferimento** all'oggetto (maniglia, handle o puntatore all'oggetto). Il riferimento permette di usare l'oggetto all'interno del programma.

Un oggetto mantiene sempre lo stesso riferimento per tutta la sua vita.
Due oggetti sono diversi se hanno riferimenti diversi, anche se sono identici, cioè hanno identico stato.
Il riferimento a un oggetto di solito viene memorizzato in una variabile.
Se si assegna lo stesso riferimento a due variabili, le variabili puntano allo stesso oggetto.

Attraverso il **riferimento** all'oggetto si possono usare i suoi **attributi** e **metodi**, con la notazione puntata.

Si può fare riferimento ad un attributo con:

```
oggetto.attributo
```

(dove *oggetto* è un riferimento all'oggetto).
Il valore di un attributo può essere usato come vengono usate le variabili; per esempio si può assegnare un valore a un attributo con un'istruzione di assegnazione:

```
oggetto.attributo = valore
```

o si può assegnare il valore di un attributo a una variabile:

```
variabile = oggetto.attributo
```

Si può richiamare un metodo con:

```
oggetto.metodo()
```

Eventuali parametri vanno indicati tra le parentesi.
I metodi in genere restituiscono un valore che si può assegnare a una variabile o utilizzare direttamente in una espressione:

```
variabile = oggetto.metodo()
```

In genere si limita la possibilità di eseguire operazioni direttamente sulle variabili di istanza rendendo disponibili dei metodi per effettuare solo le operazioni che si considerano lecite in un certo contesto.
I metodi di una classe hanno sempre pieno accesso alle variabili di istanza della classe, cioè ai dati all'interno degli oggetti della classe, ma si può modificare la **visibilità** dei dati e dei metodi degli oggetti al mondo esterno; per visibilità di un oggetto si intende la parte a cui ha accesso un altro oggetto.
In genere si fa in modo che i dati siano invisibili, cioè inaccessibili agli altri oggetti, e che le interazioni tra oggetti avvengano solo tramite metodi.

Variabili e metodi visibili all'esterno vengono chiamati **pubblici** e costituiscono l'interfaccia dell'oggetto. Variabili e metodi visibili solo all'interno vengono chiamati **privati**.

I metodi possono utilizzare il valore delle variabili di istanza o modificarle.

Un metodo che restituisce il valore di una variabile di istanza è detto **accessore**.
Un metodo che modifica il valore di una variabile di istanza è detto **modificatore**.

Nota

Alcune classi, dette immodificabili, hanno solo metodi accessori (o costanti); gli oggetti di queste classi non possono essere modificati. Un esempio è la classe *string* del C++: una volta costruita una stringa il suo contenuto non può cambiare.

Nota

In C++ un metodo che non modifica il valore delle variabili di istanza viene chiamato **metodo costante** e il qualificatore *const* è usato nella sua dichiarazione.

In genere sono disponibili metodi per impostare (***set()***) o ottenere (***get()***) il valore di ciascun attributo.
Si può rendere un attributo di **sola lettura** fornendo solo il metodo *get()* e non il metodo *set()*.

Nota

Non è detto che gli attributi di sola lettura siano costanti; potrebbero essere modificati automaticamente o derivare da un calcolo.

Per esempio per l'oggetto *persona* si potrebbe rendere disponibile un metodo per modificare l'indirizzo ma non il nome o la data di nascita.

Oltre agli attributi e ai metodi di istanza esistono anche attributi e metodi di classe (chiamati anche **statici**) che sono sempre unici per tutti gli oggetti della classe; gli attributi statici rappresentano proprietà comuni a tutte le istanze della classe; i metodi statici non operano su un oggetto ma per conto di una classe.
Ad attributi e metodi statici si accede con l'operatore :: (**operatore di risoluzione della visibilità**) attraverso il nome della classe e non attraverso un riferimento a un oggetto.

```
Classe::metodo()
```

Un esempio di attributo statico può essere un contatore degli oggetti creati. Si usa una variabile di classe che viene incrementata dal metodo costruttore ogni volta che viene creato un oggetto.

10.4 LA PROGRAMMAZIONE A OGGETTI

- La programmazione a oggetti parte dalla definizione delle classi.
- Per poter usare una classe bisogna istanziarla cioè creare un oggetto della classe (a meno che si debbano usare solo metodi di classe).
- Per eseguire delle operazioni si richiamano i metodi degli oggetti (o, a volte, metodi di classe).
- Il programma viene avviato da un metodo principale (un metodo di classe, cioè statico).

L'esecuzione di un programma consiste nell'**interazione tra oggetti**, attraverso chiamate di metodi.

Per poter usare una classe, e quindi i suoi oggetti, bisogna sapere quali sono i metodi disponibili e qual è la loro semantica (cioè come devono essere richiamati: numero e tipo di argomenti e tipo di valore restituito).
La combinazione di dati, metodi e semantiche rappresenta il contratto tra il progettista della classe e il programmatore che la usa: definisce cosa accade quando si invocano i metodi su un oggetto.

Chiamate di metodi

Per far fare qualcosa a un oggetto bisogna eseguire una parte del codice dell'oggetto cioè chiamare uno dei suoi metodi.
Gli oggetti interagiscono, comunicano e si scambiano informazioni solo tramite chiamate di metodi.

Le chiamate di metodi sono composte da tre elementi:

- l'**oggetto** su cui viene richiamato il metodo;
- il **nome** del metodo;
- i **parametri** necessari.

I parametri possono essere oggetti (o meglio riferimenti a oggetti).

Nota

In un linguaggio a oggetti **puro**, cioè che non ha dati, ma solo oggetti, i parametri possono essere solo oggetti.

L'espressione **chiamare un metodo** può essere sostituita da **inviare un messaggio;** sono espressioni equivalenti: metodo e messaggio nella programmazione a oggetti sono sinonimi.
Gli elementi che compongono l'invio di un messaggio si possono interpretare come:

- l'oggetto che riceve il messaggio;
- il nome dell'azione da eseguire;
- i parametri, cioè altre informazioni che riguardano il messaggio.

Oggetti e metodi

Un oggetto rispetto a un metodo può essere:

- l'oggetto nel cui codice viene richiamato un metodo;
- l'oggetto di cui viene eseguito il codice.

Usando la terminologia del messaggio possiamo dire che un oggetto può essere:

- mittente del messaggio;
- destinatario del messaggio.

Inoltre un oggetto può essere:

- riferito da una variabile all'interno di un altro oggetto;
- riferito da un parametro.

I metodi possono essere **informativi**, **interrogativi** o **imperativi**.
Il significato è più chiaro usando il termine messaggio.

Un messaggio **informativo** fornisce all'oggetto informazioni per aggiornarsi (messaggio di aggiornamento, inoltro o push); è orientato al passato, informa di una cosa che è avvenuta.

```
persona.matrimonio(data)
```
può essere usato per aggiornare la data di matrimonio.

Un messaggio **interrogativo** chiede informazioni all'oggetto (messaggio di lettura, retrospettivo o pull); è orientato al presente, chiede lo stato attuale dell'oggetto.

```
persona.getIndirizzo()
```
chiede l'indirizzo attuale.

Un messaggio **imperativo** dice all'oggetto di compiere un'azione su se stesso; è orientato al futuro.

```
auto.avvia()
```
indica di avviare il motore.

Overloading

Ogni metodo ha una **segnatura** (o **firma**) costituita dal nome del metodo e dal numero e tipo dei suoi parametri.
Lo stesso nome di metodo può essere usato per operazioni diverse, con definizioni diverse che richiedono un diverso tipo o numero di parametri, cioè una diversa segnatura.

La definizione di più metodi con lo stesso nome e segnatura diversa è nota come **overloading** (o **sovraccarico**).

Quando si richiama un metodo sovraccarico il metodo effettivo da utilizzare viene individuato in base alla corrispondenza dei parametri nella chiamata e nella definizione.

THIS

Il termine ***this*** (o ***self***) per un oggetto è una costante di istanza che rappresenta un riferimento all'oggetto stesso.

In C++ viene usato il termine *this* che è un puntatore all'oggetto stesso. **Nota**

Nelle chiamate di metodi il termine *this* viene usato per:

- fare riferimento mediante l'operatore -> a variabili istanza dell'oggetto stesso (e in tal caso normalmente può essere omesso, a meno che ci siano variabili locali con lo stesso nome delle variabili di istanza);
- chiamare un metodo dell'oggetto stesso, cioè inviare un messaggio a sé stesso (e in tal caso può essere omesso);
- passare se stesso come parametro in modo che il destinatario sappia chi ha richiamato il metodo, cioè chi ha inviato il messaggio; in questo caso si usa sempre il termine *this* esplicitamente;
- restituire se stesso come valore di ritorno; in questo caso si usa sempre il termine *this* esplicitamente.

La chiamata di un metodo dell'oggetto stesso, cioè l'invio di un messaggio a se stesso, permette di nascondere a un'operazione l'implementazione di un'altra operazione. **Nota**

10.5 INCAPSULAMENTO E INFORMATION HIDING

L'**incapsulamento** indica la proprietà degli oggetti di mantenere al loro interno sia gli attributi che i metodi, cioè **stato** e **comportamento** dell'oggetto.

Attributi e metodi sono quindi incapsulati all'interno dell'oggetto.
Alcuni attributi e metodi possono anche essere nascosti all'interno dell'oggetto, cioè resi invisibili agli altri oggetti; si parla in questo caso di **information hiding** o **occultamento** delle informazioni.

Attributi e metodi nascosti (o invisibili) sono detti **privati**.
Attributi e metodi visibili sono detti **pubblici** e costituiscono l'interfaccia dell'oggetto.
Quando si dice che un metodo è visibile o pubblico si intende che è visibile la sua segnatura, cioè che è possibile richiamare il metodo; l'implementazione dei metodi però è sempre nascosta, sia per i metodi privati che per quelli pubblici.

Si ha sempre **occultamento dell'implementazione**, e si può avere **occultamento delle informazioni**.

Ogni oggetto ha una porzione esterna o interfaccia e una porzione interna occultata (con visibilità limitata). Gli altri oggetti comunicano solo attraverso l'interfaccia. La parte interna è protetta da manomissioni e inoltre può essere modificata senza influire su altre parti del programma.

In particolare l'occultamento permette di nascondere gli attributi in modo che gli oggetti di altre classi non vi possano accedere.
Di solito gli attributi vengono dichiarati appunto privati e vi si può accedere solo tramite opportuni metodi (metodi di accesso).

Se gli attributi sono privati lo stato dell'oggetto diventa accessibile e modificabile solo attraverso i metodi dell'interfaccia.

Nota

Usando attributi privati il programmatore può avere un maggior controllo e stabilire il tipo di operazioni che si possono effettuare (per esempio si può permettere di vedere il contenuto di una variabile ma non di modificarlo).

Occultare metodi, cioè rendere dei metodi privati, permette di definire dei metodi che non devono essere visibili all'esterno ma che possono essere richiamati solo da altri metodi dello stesso oggetto.

Una classe *Persona* potrebbe avere come attributi *nome*, *data di nascita* e *indirizzo*, privati. Potrebbe poi rendere disponibili metodi pubblici per:

- mostrare il nome,
- mostrare la data di nascita,
- mostrare l'indirizzo,
- modificare l'indirizzo,

ma non per modificare nome e data di nascita perché questi dati non possono mai variare dopo la creazione dell'oggetto.

10.6 CLASSI E OGGETTI IN NOTAZIONE UML

Le classi e le relazioni tra le classi si possono descrivere in un diagramma di classi con la notazione **UML** (**Unified Modeling Language** - linguaggio unificato di modellazione).
Si possono usare strumenti automatici di modellazione che partendo da codice esistente disegnano i diagrammi e viceversa.

Rappresentazione di una classe

Il simbolo di una **classe** è un rettangolo diviso in tre sezioni orizzontali:

- la prima contiene il nome in grassetto (il nome per convenzione inizia con una lettera maiuscola);
- quella centrale elenca gli attributi;
- la terza elenca i metodi.

Si possono aggiungere altre sezioni con nomi che ne definiscono la finalità (come l'elenco delle eccezioni).

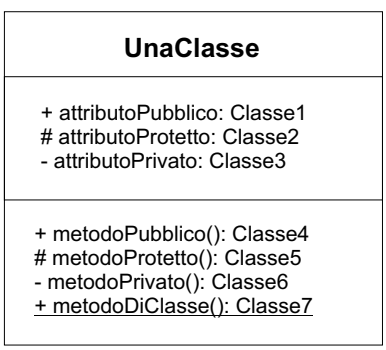


Figura 10.1 *Notazione UML per una classe*

In certi casi, per semplificare, si può usare solo un rettangolo con il nome, oppure, invece di elencare tutti i metodi, si possono elencare solo quelli utili per il diagramma che si sta realizzando.

Ogni nome di attributo è seguito da *:Classe*.

Spesso in pratica per le classi C++ si usa la notazione *Classe attributo* corrispondente alla dichiarazione degli attributi.

Nota

Gli attributi di sola lettura si possono far precedere da / (il significato preciso di questo simbolo però è attributo derivato).

Un esempio di attributo derivato è il volume di un parallelepipedo derivato da larghezza, lunghezza e altezza.

Nella terza sezione per ogni metodo vanno elencati anche i parametri.
Se il metodo restituisce un valore è seguito da *:Classe* per indicare la classe del valore di ritorno.

Nota

Spesso in pratica per le classi C++ si usa la notazione *Classe metodo(parametri)* corrispondente alla dichiarazione dei metodi.

I metodi *get()* e *set()* per gli attributi possono essere lasciati impliciti.
I metodi sovraccarichi appaiono più volte.

Ad ogni attributo e metodo si premette un **prefisso di visibilità**:
+ pubblici (si può omettere perché il diagramma di solito descrive solo attributi e operazioni pubblici)
- privati
protetti

Attributi e metodi di classe (statici) vanno sottolineati (oppure preceduti dal simbolo $).

Se una classe è astratta il nome va indicato in corsivo e seguito dalla proprietà *{abstract}.*
Anche i metodi virtuali vanno in corsivo seguiti da *{abstract}*.

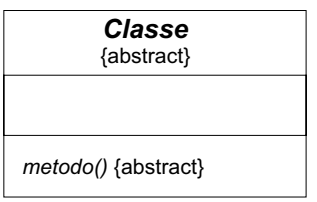


Figura 10.2
Notazione UML per una classe astratta

Rappresentazione di un oggetto

Il simbolo di un **oggetto** è un rettangolo diviso in due sezioni orizzontali:

- la prima contiene il nome dell'oggetto nella forma *nomeOggetto: Classe*, sottolineato (e non in grassetto); il nome dell'oggetto in realtà è il nome di un riferimento e può essere omesso, lasciando soltanto *:Classe*;
- la seconda elenca gli attributi e i valori corrispondenti.

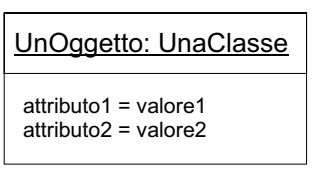


Figura 10.3
Notazione UML per un oggetto

Nota

A volte il rettangolo di rappresentazione di un oggetto viene disegnato con gli angoli arrotondati per distinguerlo ulteriormente dalla rappresentazione di una classe.

10.7 PROGETTAZIONE DELLE CLASSI

Una classe:

- rappresenta il modello per la creazione di oggetti;
- specifica i metodi che si possono usare per gli oggetti che appartengono alla classe;
- definisce i particolari dell'implementazione (variabili istanza e codice dei metodi);
- è un contenitore di metodi statici e di oggetti statici.

Una dichiarazione di classe crea un nome di tipo. Ogni tipo classe ha due tipi di membri: attributi e metodi.
La classe, cioè, è un modello che definisce gli attributi degli oggetti che verranno costruiti a partire dalla classe, e i metodi che specificano il comportamento di tali oggetti.

Individuazione delle classi

Per creare un'applicazione utilizzando la programmazione a oggetti bisogna prima di tutto **individuare le classi** necessarie a descrivere il tipo di oggetti da utilizzare.

Per ogni classe bisogna definire la struttura e il comportamento, cioè individuare gli attributi che descrivono gli oggetti e i metodi che ciascun oggetto deve eseguire per risolvere il problema.

Ogni classe dovrebbe rappresentare un singolo concetto; attributi e metodi messi a disposizione dalla classe devono avere una buona coesione, cioè essere strettamente correlate al concetto rappresentato dalla classe.

Regola pratica per individuare classi e metodi

- le classi corrispondono ai sostantivi che ricorrono nella formulazione del problema;
- i metodi corrispondono ai verbi.

La **progettazione delle classi** è il problema fondamentale della programmazione a oggetti. Una buona progettazione delle classi è indispensabile per il buon funzionamento del programma e per la riusabilità delle classi.

Spazio degli stati di una classe

Lo **spazio degli stati** è l'insieme di tutti gli stati in cui qualunque oggetto della classe si può trovare, cioè l'insieme di tutti i possibili insiemi di valori che si possono assegnare agli attributi.

In certe classi non tutti i punti dello spazio degli stati sono accessibili, cioè non tutte le possibili combinazioni di valori per gli attributi sono utilizzabili.

Lo spazio degli stati legale è definito da una **invariante di classe** cioè da una condizione che tutti gli oggetti devono soddisfare.

> Si pensi ad una classe *Triangolo* che ha come attributi tre oggetti *Punto* che costituiscono i vertici del triangolo; non si possono assegnare tre punti a caso ma la relazione tra i tre punti deve essere tale che la somma di due lati del triangolo sia maggiore del terzo lato.

Ci sono degli **stati illegali** se l'oggetto può raggiungere degli stati che violano l'invariante di classe.

> Nella classe *Triangolo* si può raggiungere uno stato illegale se si può spostare un solo vertice.

Ci sono degli **stati incompleti** se nello spazio ci sono stati legali che gli oggetti non possono raggiungere.

Il **comportamento** di una classe è l'insieme di transizioni tra gli stati che a un oggetto è consentito effettuare.

Comportamento degli oggetti

Per ottenere un comportamento corretto **non** ci devono essere:

- **comportamenti illegali**, cioè che consentono transizioni illegali (anche se gli stati sono legali);

- **comportamenti pericolosi**, cioè che passano per uno stato illegale;
- **comportamenti non pertinenti**, cioè che non appartengono alla classe;
- **comportamenti incompleti**, cioè mancanza di comportamenti necessari;
- **comportamenti scomodi**, cioè che richiedono due o più metodi per un singolo comportamento;
- **comportamenti replicati**, cioè comportamenti che si possono richiedere in più modi (può essere un errore, o per comodità, o per variazioni di comportamenti preesistenti che gli altri usano ancora).

Un altro problema di comportamento può derivare dalla **combinazione di più comportamenti** alternativi in un solo metodo, che sceglie il comportamento in base a un parametro, o anche dalla combinazione di più comportamenti che vengono applicati tutti; è sempre meglio separare ciascun comportamento in un metodo diverso.

Nella classe *Persona*, invece di usare un metodo che cambia indirizzo e numero di telefono, è meglio avere due metodi separati, uno per cambiare indirizzo e l'altro per cambiare numero di telefono.

Effetti collaterali

Se i metodi accessori o modificatori modificano dei parametri si dice che c'è un **effetto collaterale**; gli effetti collaterali dovrebbero essere ridotti al minimo.

Graduatoria dai casi migliori ai peggiori:

- metodi accessori o modificatori che non modificano parametri espliciti (nessun effetto collaterale);
- metodi che modificano parametri espliciti (effetti collaterali);
- metodi che modificano una variabile statica o che stampano messaggi.

Precondizioni e postcondizioni

Tutti i metodi hanno una **precondizione** che deve essere vera quando il metodo inizia e una **postcondizione** che deve essere vera quando il metodo termina.

La **precondizione** indica le condizioni che descrivono quali input sono validi, cioè le condizioni che devono essere soddisfatte dai parametri, quando si richiama un metodo.

Le precondizioni non soddisfatte possono essere gestite lanciando eccezioni o meglio mediante asserzioni: le asserzioni permettono di far terminare automaticamente il programma se le precondizioni non sono rispettate.

La **postcondizione** indica le condizioni che descrivono quali valori di ritorno sono validi, cioè lo stato in cui si trova l'oggetto dopo l'esecuzione del metodo.

Precondizioni e postcondizioni devono essere documentate.

Progettazione per contratto

Un metodo garantisce che le postcondizioni saranno soddisfatte, cioè che svolgerà correttamente il suo compito, se sono soddisfatte le precondizioni quando il metodo viene richiamato.

La programmazione difensiva invece non fa nessuna assunzione sulle precondizioni: il metodo effettua tutti i controlli sull'input per individuare tutti i possibili errori (non ci si fida dell'implementazione dei metodi che si usano).

Nota

Il primo programma a oggetti

C++ 10.1

Esempio 10.1

Creazione di una classe con un metodo per scrivere un saluto.

```
#include <iostream>
using std::cout;

class Ciao {
    public:
        void saluto();
};//Ciao

void Ciao::saluto(){
    cout << "Ciao a tutti\n";
}//Saluto

int main(void){
    Ciao ciao;
    ciao.saluto();
    getchar();
    return 0;
}//main
```

Figura 10.4 *Esecuzione dell'esempio*

Per richiamare il metodo *saluto()* bisogna creare un oggetto della classe *Ciao*.
Non essendo specificato nessun costruttore per creare un oggetto basta dichiarare una variabile di tipo *Ciao* . Attraverso il riferimento all'oggetto creato si può richiamare il metodo *saluto()*.
Queste operazioni naturalmente devono essere inserite in un altro metodo della classe o in una funzione come *main()*.

Nota

Attenzione alla differenza tra *Ciao* e *ciao*: *Ciao* rappresenta la classe (il tipo), mentre *ciao* è una variabile che rappresenta un oggetto.

L'implementazione del metodo *saluto()* può avvenire all'interno della classe. In questo caso prende il nome di metodo **inline**.

Esempio 10.2

Classe *Ciao* con metodo inline.

```
#include <iostream>
using std::cout;

class Ciao {
    public:
        void saluto(){
            cout << "Ciao a tutti\n";
        }//Saluto
};//Ciao

int main(void){
    Ciao ciao;
    ciao.saluto();
    getchar();
    return 0;
}//main
```

Per mantenere l'occultamento delle informazioni è bene dividere in più file la dichiarazione della classe, l'implementazione dei metodi e il collaudo (funzione *main()*) della classe); naturalmente perché ciò sia possibile non vanno usati i metodi inline.
Inoltre dividendo le classi in più file, se più persone lavorano sulle classi di una applicazione, ciascuno può modificare solo il file di una classe.

Nota

Un'applicazione può essere costituita da più classi e avere un numero arbitrario di metodi *main()* in quanto ogni classe può averne uno. Per ogni esecuzione però viene usato solo un *main()*, quello della classe linkata. La possibilità di avere un *main()* per ogni classe può essere utile durante la fase di sviluppo e collaudo del codice.

Classe e codice di collaudo in file separati

Per salvare la classe e il codice di collaudo in file separati si può:

- salvare la dichiarazione della classe nel file *NomeClasse.h*,
- salvare l'implementazione dei metodi nel file *NomeClasse.cpp*,
- salvare il codice di collaudo (funzione *main()* ed eventualmente altro) nel file *ProvaClasse.cpp*.

Tutti i file *.cpp* che necessitano della classe (per l'implementazione o l'uso dei metodi) devono includere con la direttiva *#include* il file *NomeClasse.h*.

Ciascun file può includere altri file.

Esempio 10.3

Modifica dell'esempio 10.1 con la separazione in tre file della dichiarazione della classe, l'implementazione dei metodi e il collaudo.
Ciao.h contiene la definizione della classe; *Ciao.cpp* l'implementazione dei metodi e *ProvaCiao.cpp* la funzione *main()*. Entrambi i file *.cpp* includono *Ciao.h*.

```
//file Ciao.h
class Ciao {
    public:
        void saluto();
};//Ciao
```

```
//file Ciao.cpp
#include <iostream>
#include "Ciao.h"
using namespace std;
void Ciao::saluto(){
    cout << "Ciao a tutti\n";
}//Saluto
```

```
//ProvaCiao.cpp
#include <iostream>
#include "ciao.h"

int main(void){
    Ciao c = Ciao();
    c.saluto();
    getchar();
    return 0;
}//main
```

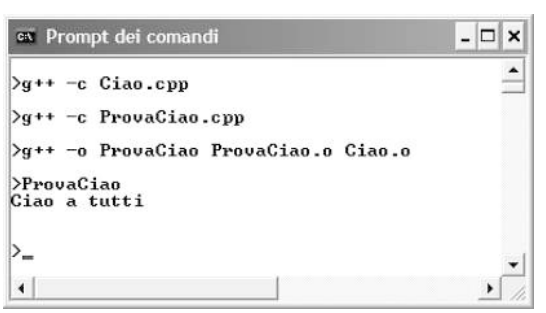


Figura 10.5
Compilazione dei file .cpp e successiva esecuzione

CLASSI

C++

10.2

Struttura di una classe:

```
class Nomeclasse {
    // attributi
    // metodi
};
```

Le dichiarazioni degli attributi possono trovarsi in qualsiasi punto della classe, ma non all'interno di metodi. Conviene porle all'inizio della classe per maggior leggibilità.
I metodi possono trovarsi in qualsiasi punto ma non inseriti uno nell'altro.

DICHIARAZIONE DI ATTRIBUTI

Gli attributi (o variabili di istanza) sono variabili dichiarate in una classe.

Tutti i metodi della stessa classe possono utilizzare gli attributi, come normali variabili.
I metodi di altre classi e le funzioni devono usare un riferimento a un oggetto della classe per accedere a un attributo:

```
oggetto.attributo
```

La **dichiarazione** degli attributi determina il nome, il tipo e la modalità di accesso.
La forma più semplice di dichiarazione è

```
Tipo nome;
```

Il tipo può essere uno qualsiasi dei tipi di C++, predefiniti o classi.

Questa dichiarazione può essere inserita in un blocco specificato da un modificatore che permette di impostare proprietà dell'attributo.

I **modificatori di accesso** permettono di stabilire il livello di protezione degli attributi cioè da chi possono essere usati gli attributi.
Se non si specifica un modificatore di accesso l'attributo viene considerato per default di tipo **private**.
I modificatori di accesso che si possono specificare sono:

public	rende l'attributo visibile a tutti;
private	rende l'attributo visibile solo alla classe in cui è dichiarato;
protected	l'attributo è visibile nella classe stessa e in tutte le classi derivate.

ESEMPIO 10.4

Classe *Articolo* con solo attributi.

```
//Articolo.h
#include <string>
using std::string;

class Articolo {
    public:
        string descrizione;
        double prezzo;
};//Articolo
```

```
//ProvaArticolo.cpp
#include <iostream>
#include "Articolo.h"
```

Rappresenta un articolo in vendita; gli attributi sono pubblici e quindi accessibili da qualsiasi funzione: una funzione può creare un oggetto *Articolo* e leggere o modificare il valore degli attributi.

Figura 10.6 *Esecuzione dell'esempio*

```
using namespace std;
int main(void){
    Articolo art;
    art.descrizione = "quaderno";
    art.prezzo = 1.50;
    cout << "Descrizione articolo: " << art.descrizione << endl;
    cout << "Prezzo: " << art.prezzo << endl;
    getchar();
    return 0;
}//main
```

Nota

In genere gli attributi sono privati e sono disponibili opportuni metodi di accesso.

Oltre a specificare un modificatore di accesso nella dichiarazione degli attributi si può usare anche il qualificatore ***const*** che rende la variabile non modificabile e quindi in pratica permette di definire una costante.

Dichiarazione di metodi

I metodi devono sempre essere dichiarati all'interno di una classe. L'implementazione può avvenire in un file distinto.

I metodi di una classe possono chiamarsi l'un l'altro semplicemente usando il nome del metodo seguito dai parametri.
Le funzioni per chiamare un metodo devono usare un oggetto della classe:

```
oggetto.metodo(parametri)
```

Nota

Nella chiamata di metodi della stessa classe non è necessario usare un oggetto perché viene sempre sottinteso il puntatore *this* che punta all'oggetto corrente.

I metodi possono essere richiamati anche ricorsivamente.

La forma più semplice di **dichiarazione** di un metodo è:

```
TipoDiRitorno nomeMetodo(parametri);
```

L'implementazione di un metodo è:

```
TipoDiRitorno Classe::nomeMetodo(parametri) {
    // codice del metodo
}//nomeMetodo
```

La dichiarazione e l'implementazione possono essere unite e il metodo allora viene detto metodo **inline**:

```
TipoDiRitorno nomeMetodo(parametri) {
    // codice del metodo
}//nomeMetodo
```

L'elenco dei parametri è del tipo:

```
Tipo nomeVariabile, Tipo nomeVariabile  ...
```

e può essere costituito da qualsiasi numero di parametri.

Se non ci sono parametri bisogna comunque indicare le parentesi.
Se l'ultimo parametro (dopo almeno un parametro fisso) è costituito da tre puntini (...) il metodo può ricevere un numero variabile di parametri.

I parametri possono essere passati **per valore**, **per riferimento** o **per indirizzo**.

TipoDiRitorno indica il tipo di dato restituito dal metodo; se un metodo non restituisce alcun valore bisogna indicare *void* come tipo di ritorno.
Un metodo che restituisce un tipo diverso da *void* deve restituire un valore mediante l'istruzione ***return*** *espressione*.
Nell'istruzione *return* si può indicare una variabile ma anche direttamente un'espressione; il valore risultato dell'espressione deve essere di un tipo compatibile con il tipo di ritorno (cioè uguale o che possa essere convertito implicitamente nel tipo di ritorno).
Se il metodo non restituisce alcun valore si può comunque usare l'istruzione *return* per indicare la conclusione del metodo.
L'istruzione *return* provoca la conclusione del metodo e il ritorno al chiamante, dopo aver restituito il valore, se indicato.
In genere *return* è l'ultima istruzione del metodo ma ci possono essere più istruzioni *return*, per esempio una in ogni ramo di una istruzione condizionale.

Nota

Ogni ramo di una istruzione condizionale deve restituire un valore; se un ramo non restituisce un valore il compilatore segnala errore.

Si possono realizzare metodi che restituiscono più di un risultato restituendo un puntatore ad un array che contenga i risultati, restituendo un riferimento (o un puntatore) a un oggetto che contenga i risultati o definendo uno o più parametri che siano riferimenti (o puntatori) a oggetti che al termine conterranno i risultati.

La dichiarazione del metodo può essere inserita in un blocco specificato da un modificatore che permette di impostare proprietà del metodo.

I **modificatori di accesso** permettono di limitare l'accesso a un metodo.
Se non si specifica un modificatore di accesso il metodo viene considerato per default di tipo **private**.
I modificatori di accesso che si possono specificare sono:

public	il metodo può essere richiamato da tutti;
private	il metodo può essere richiamato solo nella classe in cui è definito;
protected	il metodo può essere richiamato nella classe stessa e in tutte le classi derivate.

Se il metodo lancia delle eccezioni la dichiarazione può contenere la clausola *throw*:

```
TipoDiRitorno nomeMetodo(parametri) throw (ListaDiEccezioni)
```

Un metodo che non modifica le variabili di istanza dell'oggetto su cui è invocato è detto **metodo costante** e ha come dichiarazione:

```
TipoDiRitorno nomeMetodo(parametri) const;
```

Esempio 10.5

Classe *Articolo* con attributi privati e metodi *set()* e *get()* per impostare gli attributi e ottenerne il valore.

Nota

I metodi *get()* sono costanti.

```
//Articolo.h
#include <string>
using std::string;
```

```
class Articolo {
    private:
        string descrizione;
        double prezzo;
    public:
        void setDescrizione(string);
        void setPrezzo(double);
        string getDescrizione(void) const;
        double getPrezzo(void) const;
};//Articolo
```

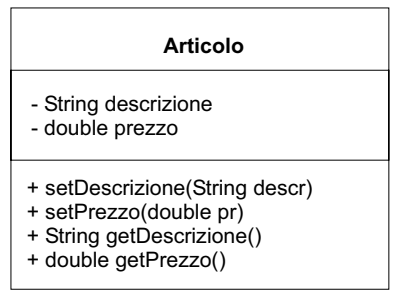


Figura 10.7 *Rappresentazione UML della classe Articolo*

```
//Articolo.cpp
#include "Articolo.h"

void Articolo::setDescrizione(string descr){
    descrizione = descr;
}//setDescrizione

void Articolo::setPrezzo(double pr){
    prezzo = pr;
}//setPrezzo

string Articolo::getDescrizione(void) const{
    return descrizione;
}//getDescrizione

double Articolo::getPrezzo(void) const{
    return prezzo;
}//getPrezzo
```

Passaggio dei parametri

Il passaggio dei parametri può avvenire

- per **valore**: il parametro attuale fornito al momento della chiamata viene copiato nel parametro formale e le eventuali modifiche agiscono sulla copia, non sull'originale, quindi non si può modificare il contenuto di una variabile o oggetto passati come parametro;
- per **riferimento**: il parametro formale è un riferimento che assume il valore del riferimento del parametro attuale fornito al momento della chiamata; le modifiche apportate all'oggetto all'interno del metodo avvengono sempre sullo stesso oggetto, quindi hanno effetto;
- per **indirizzo**: il parametro formale è un puntatore che assume il valore dell'indirizzo del parametro attuale fornito al momento della chiamata; le modifiche apportate all'oggetto all'interno del metodo avvengono sempre sullo stesso oggetto, quindi hanno effetto.

Uso di variabili di istanza e locali

All'interno di un metodo si possono utilizzare variabili di istanza e variabili locali.

Quando in un metodo ci si riferisce a una **variabile di istanza** ci si riferisce automaticamente alla variabile di istanza dell'oggetto per il quale si invoca il metodo.

Le variabili dichiarate all'interno di un metodo sono **locali** al metodo.
Una variabile locale a un metodo è visibile solo all'interno delle parentesi graffe in cui è dichiarata.
Le variabili locali devono sempre essere inizializzate.

Se si usa per una variabile locale lo stesso nome di una variabile di istanza, la variabile locale ha la precedenza (nasconde la variabile di istanza omonima).

Per riferirsi a una variabile di istanza nascosta si può usare *this->variabile*.

Nota

THIS

I parametri passati ad un metodo tra parentesi vengono chiamati parametri espliciti.
Ad ogni metodo viene passato anche un **parametro implicito** costituito dal puntatore all'oggetto su cui viene chiamato il metodo, indicato dalla parola ***this***; il tipo del parametro implicito è quello della classe che definisce il metodo.

Il parametro implicito *this* può essere usato per fare riferimento alle variabili di istanza dell'oggetto; cioè se *variabile* è una variabile di istanza, nel metodo ci si può riferire ad essa indifferentemente con *variabile* o *this->variabile*.
Anche se non si usa in modo esplicito, *this* viene anteposto implicitamente a ogni riferimento ad attributi o metodi.
Per convenzione si usa esplicitamente *this* solo quando necessario, per esempio per fare riferimento a una variabile di istanza nascosta da una variabile locale.

ESEMPIO 10.6

Uso di *this* per fare riferimento alle variabili di istanza.

Il file *Articolo.h* è quello dell'esempio 10.5.

Nota

```
//Articolo.cpp
#include "Articolo.h"

void Articolo::setDescrizione(string descrizione){
    this->descrizione = descrizione;
}//setDescrizione

void Articolo::setPrezzo(double prezzo){
    this->prezzo = prezzo;
}//setPrezzo
```

this può essere usato anche per passare come parametro ad altri metodi l'oggetto corrente.

```
double Articolo::getPrezzo(void) const{
    stampaDescrizione(*this);
    return prezzo;
}//getPrezzo

void Articolo::stampaDescrizione(Articolo a) const{
    cout << a.getDescrizione() << endl;
}//mostraDescrizione

Articolo art;
art.getPrezzo();
```

Il metodo *getPrezzo()* richiama il metodo *stampaDescrizione()* passandogli come parametro l'articolo su cui è stato invocato.

Se il metodo che riceve l'oggetto corrente è statico ed appartiene ad un'altra classe l'invocazione diventa:

```
ClasseDelMetodo::metodo(*this);
```

Overloading o sovraccarico

Si possono scrivere definizioni diverse dello stesso metodo (**overloading**) purché abbiano segnatura diversa.
La **segnatura** di un metodo è costituita dal nome del metodo e dal numero e tipo dei suoi parametri; possono esserci più definizioni con lo stesso numero di parametri purché il tipo sia diverso. Quale metodo utilizzare dipende dai parametri passati al momento della chiamata.

Nota

Il tipo del valore di ritorno non fa parte della segnatura e neppure le eccezioni lanciate dal metodo perché sarebbe troppo ambiguo determinare il metodo richiamato.

In C++ è possibile sovraccaricare anche moltissimi operatori (circa 40). La funzione che gestisce un qualsiasi operatore *opGen* ha nome ***operator****opGen* con numero e tipo di parametri dipendenti dalla natura dell'operatore e dalla classe.

```
bool operator==(const Articolo&) const;
```
prototipo di una funzione per l'overloading dell'operatore == relativo alla classe *Articolo*.

Esempio 10.7

Overloading del metodo *getPrezzo()* nella classe *Articolo*.

```
double Articolo::getPrezzo(void) const{
    return prezzo;
}//getPrezzo

double Articolo::getPrezzo(double pr){
    double vecchioPrezzo = prezzo;
    prezzo = pr;
    return vecchioPrezzo;
}//getPrezzo
```

Il secondo metodo *getPrezzo()* permette di modificare il prezzo e di ottenere il valore del prezzo precedente.
Quale metodo utilizzare dipende dal modo in cui il metodo viene richiamato, con o senza parametro.

Non si potrebbe invece fare l'overloading del metodo:

```
void Articolo::setPrezzo(double pr) {
   prezzo = pr;
}//setPrezzo
```

con il metodo:

```
double Articolo::setPrezzo(double pr) {
   double vecchioPrezzo = prezzo;
   prezzo = pr;
   return vecchioPrezzo;
}//setPrezzo
```

infatti i due metodi sarebbero indistinguibili perché hanno la stessa segnatura (il tipo del valore di ritorno non fa parte della segnatura).

Overloading degli operatori ++ e --

Gli operatori unari di incremento e decremento unitario possono essere sovraccaricati. Le funzioni o i metodi che li sovraccaricano sono rispettivamente *operator++* e *operator--*.

Poiché gli operatori possono essere in forma prefissa e postfissa, diventa necessario specificare a quale forma si riferisce il sovraccarico.

Solitamente si usa un parametro fittizio di tipo *int* per la forma postfissa.

Nel caso dell'operatore ++ i sovraccarichi come metodi della classe seguono gli schemi di dichiarazione:

```
NomeClasse NomeClasse::operator++();          forma prefissa;
NomeClasse NomeClasse::operator++(int);       forma postfissa.
```

Nota: Gli schemi di dichiarazione per l'operatore -- sono del tutto analoghi.

Funzioni friend

In una classe è possibile dichiarare anche funzioni e non solo metodi.

Nella dichiarazione della classe non si è in grado di distinguere una funzione da un metodo. La differenza sta nell'implementazione in cui per il metodo è usato l'operatore :: (operatore di risoluzione della visibilità) per associare il metodo alla classe.

Le variabili di istanza sono solitamente dichiarate nel blocco *private* e le funzioni non possono accedervi direttamente. Il C++ mette a disposizione un particolare tipo di funzioni, le funzioni **friend** (o funzioni amiche), a cui è concesso accedere alle variabili di istanza.

La dichiarazione di una funzione *friend* è

```
friend TipoDiRitorno nomeFunzione(parametri);
```

Nota: Le funzioni *friend* superando il divieto di accesso alle variabili di istanza imposto dal modificare di accesso *private*, sono uno strumento potente; è bene non abusarne.

Solitamente le funzioni che sovraccaricano gli operatori sono *friend*.

Sovraccarico degli operatori come funzioni friend:

```
friend NomeClasse operator++(NomeClasse&);          forma prefissa;
friend NomeClasse operator++(NomeClasse&, int);     forma postfissa.
```

Costruttori

Ogni classe ha almeno un metodo **costruttore** (eventualmente **implicito**) che viene richiamato quando viene creata una istanza della classe.

Il costruttore ha sempre lo stesso nome della classe.
A differenza degli altri metodi non specifica un tipo di dati restituito (nemmeno *void*).

Il costruttore viene richiamato con la sintassi ***Classe(listaParametri)*** e restituisce l'oggetto creato.
Il costruttore riserva la memoria per l'oggetto e ne inizializza gli attributi.

Nota: Gli attributi non inizializzati assumono valori casuali.

Affinché altre classi siano in grado di creare oggetti della classe i costruttori devono essere **pubblici**.

Nota: Si può definire un costruttore privato quando si vuole creare una classe per permettere di usare metodi statici della classe ma non dare la possibilità di creare istanze della classe.

Possono esserci più costruttori con numero e tipo di parametri diversi.
Il nome è sovraccarico e il compilatore decide quale chiamare esaminando i parametri.

Se non si definiscono costruttori viene fornito un **costruttore di default** senza parametri che alloca lo spazio in memoria e inizializza le variabili di istanza con valori casuali.

Nota

Nell'esempio 10.4 l'allocazione dell'oggetto *art* avveniva nel contesto della sua dichiarazione attraverso l'implicita invocazione del costruttore di default perché la classe *Articolo* non possedeva nessun altro costruttore.

Se in una classe è presente un costruttore parametrico allora il costruttore di default implicito non è più disponibile, se si cerca di utilizzarlo si verifica un errore di compilazione.

Esempio 10.8

Definizione dei costruttori della classe *Articolo*.

```
//Articolo.h
#include <string>
using std::string;

class Articolo {
    private:
        string descrizione;
        double prezzo;
    public:
        ...
        Articolo(string descr);
        Articolo(string descr, double pr);
};//Articolo
```

```
//Articolo.cpp
#include "Articolo.h"

Articolo::Articolo(string descr){
    descrizione = descr;
    //non inizializza il campo prezzo che avrà un valore casuale
}//Articolo

Articolo::Articolo(string descr, double pr){
    descrizione = descr;
    prezzo = pr;
}//Articolo

//metodi
```

```
//ProvaArticolo.cpp
#include <iostream>
#include "Articolo.h"
using namespace std;

int main(void){
    Articolo art1 = Articolo("articolo1", 100.67);
    Articolo art2 = Articolo("articolo2");
    cout << "Descrizione articolo: " << art1.getDescrizione() << endl;
    cout << "Prezzo: " << art1.getPrezzo() << endl;
```

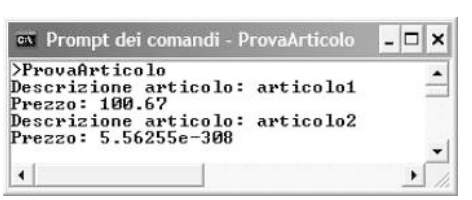


Figura 10.8 *Esecuzione dell'esempio*

```
    cout << "Descrizione articolo: " << art2.getDescrizione() << endl;
    cout << "Prezzo: " << art2.getPrezzo() << endl;
    getchar();
    return 0;
}//main
```

L'attributo *prezzo* dell'oggetto *art2* non è stato inizializzato dal costruttore, pertanto assume valori casuali privi di significato.

Un costruttore può essere invocato anche nella forma:

```
Classe nomeOggetto(listaParametri)
```

```
Articolo art("articolo1", 20.5);
```
equivale a
```
Articolo art = Articolo("articolo1", 20,5)
```

Costruttori con argomenti di default e convertitori di tipo

È possibile specificare nella dichiarazione di un costruttore il valore di default dei suoi argomenti.

```
Classe (Tipo1 =valore1, ..., TipoN =valoreN);
```

Per un negozio che vende molti articoli a 4,99 euro è comodo avere il costruttore:
```
Articolo(string ="Articolo stardard", double = 4.99);
```

La definizione di un costruttore con *n* argomenti di default definisce implicitamente tutti i costruttori aventi da *0* a *n-1* argomenti. Una chiamata del costruttore in cui vengono specificati solo alcuni parametri crea un oggetto con i rimanenti campi inizializzati con i valori di default.

Assegnando a una variabile del tipo della classe un valore del tipo del primo argomento del costruttore viene richiamato il costruttore con un parametro; il costruttore funge così da **convertitore di tipo** (viene effettuata implicitamente la conversione dal tipo del parametro al tipo dell'oggetto).

```
Articolo art1("articolo1", 100.67);
Articolo art2 = Articolo();
Articolo art3;
Articolo art4("articolo4");
Articolo art5 = (string) "articolo5";
```

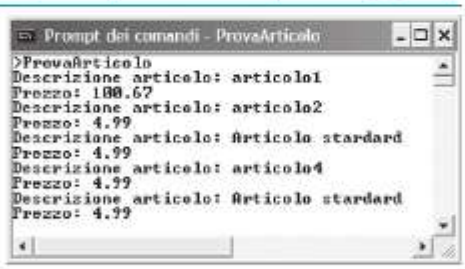


Figura 10.9
Esecuzione dell'esempio

Costruttori copia

Se una classe ha almeno un costruttore automaticamente possiede anche il **costruttore copia** così chiamato perché permette la creazione di un nuovo oggetto partendo da uno esistente. L'oggetto creato avrà gli attributi con gli stessi valori dell'oggetto di partenza.
Il costruttore copia può essere invocato con due sintassi diverse:

```
Classe nuovoOggetto = oggettoEsistente;
Classe nuovoOggetto = Classe(oggettoEsistente);
```

Nota

La prima delle due notazioni è dovuta al sovraccarico dell'operatore di assegnazione = che avviene in modo implicito definendo la classe.

La segnatura del costruttore copia è *Classe(const Classe&)*, cioè il costruttore riceve un riferimento a un oggetto della classe. Il qualificatore *const* ha lo scopo di impedire la modifica dell'oggetto ricevuto.

```
Articolo art3 = art1;
Articolo art4 = Articolo(art2);
```
vengono creati i nuovi oggetti *art3* e *art4* con gli stessi attributi di *art1* e *art2*.

Costruttore mediante lista di inizializzazione

Non è possibile inizializzare una costante all'interno del costruttore con l'operatore di assegnazione =. L'implementazione del costruttore in presenza di attributi costanti diventa:

```
Classe::Classe(parametri) : attributo1(valore), attributo2(valore), ... {
    //eventuali altre istruzioni
}//Classe
```

indipendentemente dalla natura degli attributi (variabili o costanti).

La notazione prende il nome di costruttore mediante **lista di inizializzazione**.

```
Articolo::Articolo(string descr, double pr, int scaf, long cod):
    descrizione(descr), prezzo(pr), scaffale(scaf), codice(cod){
}//Articolo
```

Il distruttore

Gli oggetti vengono cancellati automaticamente alla fine del loro periodo di vita da un particolare metodo implicito, il **distruttore**.

Il distruttore può essere visto come l'antitesi del costruttore; infatti ha come nome ***~Classe***.

Nota

Il carattere ~ (tilde) è disponibile con Alt+126 del tastierino numerico (in Linux con AltGr+').

Quando un oggetto contiene come campi di istanza altri oggetti, il distruttore implicito non libera la memoria allocata per gli oggetti interni. Diventa necessario definire un distruttore della classe con un suo corpo di istruzioni. Non è possibile modificare la segnatura del distruttore.

```
class Articolo {
    ...
    public:
        ...
        ~Articolo();
};//Articolo

Articolo::~Articolo(){
    cout << "Oggetto Rimosso\n";
}//~Articolo()
```

Tipi di variabili, ciclo di vita, ambito di visibilità e inizializzazione

Le dichiarazioni di variabili possono apparire in qualunque punto del codice.

Tipi di variabili:

- variabili di istanza,
- variabili locali,
- variabili parametro.

Il **ciclo di vita** determina quando la variabile viene creata e per quanto ha senso.
Le variabili di istanza vengono create quando si costruisce l'oggetto e continuano a esistere finché si può accedere all'oggetto (poi l'oggetto viene eliminato dal distruttore di default o specifico).
Una variabile locale viene creata quando il programma elabora l'istruzione che la definisce e rimane in vita finché il programma esce dal blocco che racchiude la definizione della variabile.
Le variabili parametro vengono create quando si chiama un metodo e rimangono in vita fino al ritorno al chiamante.

L'**ambito di visibilità** di una variabile indica la parte del programma da cui si può accedere alla variabile.
In generale una variabile è visibile nel blocco in cui è stata dichiarata, dopo la sua dichiarazione.
L'ambito delle variabili di istanza è costituito da tutti i metodi della classe.
L'ambito di una variabile locale va dal punto della sua dichiarazione alla fine del blocco che la contiene. Una variabile dichiarata in un ciclo *for* è disponibile solo all'interno del *for*.
Ci possono essere variabili con lo stesso nome in metodi o blocchi diversi, dato che il loro ambito è distinto.
L'ambito di un parametro è tutto il corpo del metodo.
Se variabili con lo stesso nome hanno ambiti sovrapposti alcune variabili possono venire nascoste; se ci sono variabili con lo stesso nome delle variabili di istanza prevale la variabile locale e si può fare riferimento alla variabile di istanza tramite il puntatore *this* (quando si usa una variabile vengono cercate nell'ordine: variabili locali del blocco di codice, parametri del metodo o costruttore al cui interno è il codice, variabili di istanza).

La visibilità di una variabile determina quando la variabile viene **inizializzata**.
Le variabili di istanza vengono inizializzate al momento della creazione dell'oggetto da parte del costruttore.
Le variabili locali vengono inizializzate ogni volta che viene eseguita la loro dichiarazione.
I parametri vengono inizializzati ogni volta che viene richiamato il metodo.

Le **variabili di istanza**, se non vengono inizializzate con valori espliciti, assumono **valori casuali**.

Le variabili locali se non vengono inizializzate esplicitamente contengono valori casuali privi di significato.
Le variabili parametro vengono inizializzate in base alla modalità di passaggio.

Scelta del tipo di accesso

Le **variabili di istanza** in genere vengono dichiarate **private**; le **costanti** di solito sono **pubbliche**.
I **metodi** normalmente sono **pubblici**.
Sono privati i metodi che vengono usati nell'implementazione ma non devono essere visibili all'esterno.

Tabella 10.1 *Modificatori di accesso per attributi e metodi*

Accesso	Visibilità di attributi e metodi
public	tutti
private	solo all'interno della classe
protected	nella classe e in tutte le classi derivate

Eccezioni ed asserzioni

Ogni metodo dovrebbe essere documentato descrivendo quali input sono validi, cioè quali precondizioni devono essere soddisfatte.
Il metodo deve verificare se le precondizioni sono soddisfatte e in caso negativo sollevare una eccezione.

Esempio 10.9

In un metodo o funzione per il calcolo della radice quadrata la precondizione è che il valore sia positivo.

Per semplicità non è stata definita una classe ma si è preferito lanciare l'eccezione da una funzione. Nulla cambia se l'eccezione è lanciata da un metodo di una classe.

Nota

```
#include <iostream>
#include <cmath>
using namespace std;

double radiceA(double num) throw(int){
    if(num < 0)
        throw 0;
    return sqrt(num);
}//radiceA

int main(void){
    double numero;
    cout << "Inserire il numero:\n";
    cin >> numero;
    fflush(stdin);
    try{
        cout << "La radice vale: " << radiceA(numero) << endl;
    }catch(int){
        cout << "Radice impossibile: numero negativo\n";
        getchar();
        return 1;
    }//catch
    getchar();
    return 0;
}//main
```

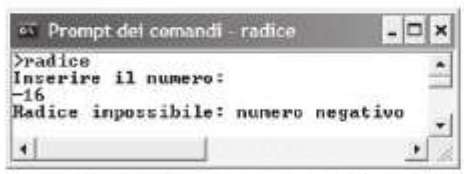


Figura 10.10 *Esecuzione dell'esempio*

Invece di usare le eccezioni, per garantire che le precondizioni siano rispettate si possono usare le asserzioni.

Un'**asserzione** ha il formato

```
assert(condizione)
```

condizione è la precondizione che deve essere soddisfatta per l'esecuzione della funzione.

Se la condizione è falsa l'esecuzione termina.

La funzione ***assert()*** è definita nel file di libreria *<cassert>*.

Esempio 10.10

Precondizione che il valore per il calcolo della radice quadrata sia positivo garantita con un'asserzione.

```
#include <iostream>
#include <cassert>
#include <cmath>
using namespace std;
```

```
double radiceA(double num){
    assert(num >= 0);
    return sqrt(num);
}//radiceA

int main(void){
    double numero;
    cout << "Inserire il numero:\n";
    cin >> numero;
    cout << "La radice vale: " << radiceA(numero) << endl;
    return 0;
}//main
```

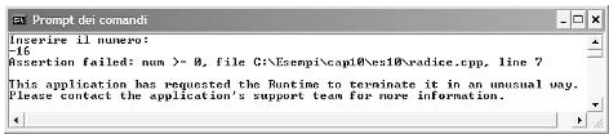


Figura 10.11 *Esecuzione dell'esempio*

Confronto tra eccezioni e asserzioni

Le **asserzioni** sono preferibili quando servono come controllo al programmatore durante lo sviluppo.
Con le **eccezioni** il programma è meno leggibile, meno efficiente, meno semplice (bisogna usare istruzioni *try catch*) e meno affidabile (si potrebbero scrivere istruzioni *catch* che non fanno niente, solo per compilare il programma).

Le asserzioni sono utilizzate nella progettazione per contratto, le eccezioni nella programmazione difensiva.

Oggetti

C++ 10.3

Gli oggetti vengono **creati** richiamando il costruttore della classe; bisogna specificare gli eventuali parametri necessari alla creazione:

```
Classe(parametri);
```

Nota Nel paragrafo C++ 10.2 si sono viste altre notazioni possibili.

Quando si crea un oggetto gli si associa uno stato iniziale, cioè un valore per ogni attributo. L'inizializzazione degli attributi in genere viene fatta dal costruttore.

I nuovi oggetti vengono allocati in un'area di memoria del sistema detta heap.
Se non c'è spazio viene lanciata l'eccezione *bad_alloc*.

È possibile creare nuovi oggetti partendo da oggetti esistenti grazie all'overloading dell'operatore di assegnazione = e al costruttore copia.

L'uso degli operatori di confronto (e anche di altri operatori, come quelli di incremento) con oggetti, generalmente dà errori di compilazione.
È necessario sovraccaricare lo specifico operatore per adattare il suo comportamento alla natura della classe.

Overloading dell'operatore <<

È possibile rappresentare lo stato di un oggetto con una stringa (in modo che abbia una forma stampabile).

La conversione dello stato dell'oggetto in stringa si può ottenere sovraccaricando l'operatore <<.
Di solito il sovraccarico dell'operatore è realizzato con una funzione friend.

Nota

Se non si sovraccarica l'operatore la scrittura *cout << nomeOggetto* provoca un errore di compilazione perché l'operatore << non sa come è fatto *nomeOggetto*.

L'operatore << agisce su un oggetto della classe ***ostream***.

Nota

La classe *ostream*, che sarà trattata nel capitolo 16, è definita nel file di libreria <*iostream*>.

La dichiarazione di una funzione friend per sovraccaricare l'operatore <<, in generale è

```
friend ostream& operator<<(ostream&, const NomeClasse&);
```

L'implementazione segue lo schema:

```
ostream& operator<<(ostream& os, const NomeClasse& oggetto){
    os << ... << //attributi dell'oggetto;
    return os;
}//operator<<
```

Esempio 10.11

Modifica della classe *Articolo*; overloading dell'operatore <<.

Nella classe ***Articolo*** la dichiarazione è

```
friend ostream& operator<<(ostream&, const Articolo&);
```

e l'implementazione è

```
ostream& operator<<(ostream& os, const Articolo& a){
    os << "Descrizione articolo: " << a.descrizione << endl;
    os << "Prezzo: " << a.prezzo << endl;
    return os;
}//operator<<
```

Per stampare i dati di un oggetto della classe *Articolo* basta fare:

```
cout << art1 << endl;
```

Puntatori ad oggetti

Si può definire un puntatore di tipo oggetto di una classe con l'istruzione:

```
NomeClasse* nomePuntatore;
```

Se è già stato creato un oggetto della classe è possibile far puntare ad esso il puntatore con:

```
nomePuntatore = &nomeOggetto;
```

Attraverso l'operatore *new* si può creare direttamente un oggetto identificato dal puntatore chiamando un costruttore della classe:

```
nomePuntatore = new NomeClasse(parametri);
```

In entrambi i casi i campi e i metodi (pubblici) dell'oggetto sono accessibili al puntatore attraverso l'operatore ->.

Esempio 10.12

Modifica della classe *Articolo*; uso di puntatori ad oggetti.

```
//ProvaArticolo.cpp
#include <iostream>
```

```
#include "Articolo.h"
using namespace std;

int main(void){
    double p = 100.67;
    Articolo *pA1, *pA2;
    Articolo art1("Testo descrizione", p);
    pA1 = &art1;
    pA2 = new Articolo("Altra descrizione", 2*p);
    art1.mostraOggetto();
    pA1->mostraOggetto();
    pA2->mostraOggetto();
    delete pA1;
    delete pA2;
    getchar();
    return 0;
}//main
```

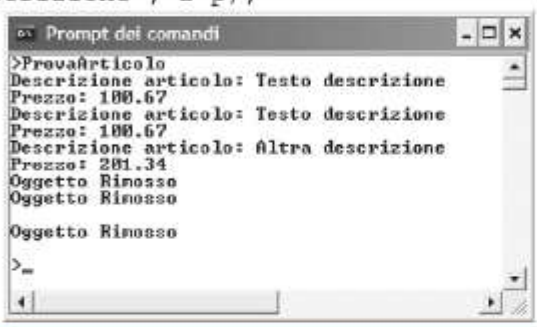


Figura 10.12 *Esecuzione dell'esempio*

OGGETTI FUNZIONE

Un **oggetto-funzione** è una istanza di una classe contenente un metodo per sovraccaricare l'operatore *()* chiamato **operatore di chiamata a funzione**.
Il sovraccarico ha la forma

```
TipoValoreRitorno operator() (listaParametri)
```

e consente di usare un oggetto *oggF* della classe con il sovraccarico di *()* come nome di funzione:

```
oggF(listaParametriAttuali)
```

VARIABILI E METODI STATICI

C++ 10.4

VARIABILI STATICHE

Le **variabili statiche** o **di classe** permettono di definire proprietà comuni a tutte le istanze della classe. Per indicare che una variabile è statica nella dichiarazione si deve usare il modificatore ***static***.

```
static NomeTipo attributo;
```

L'accesso a una variabile statica di una classe diversa avviene usando il **nome della classe** e non un riferimento a un oggetto.

```
Classe::attributo
```

Oltre al modificatore *static* si può specificare il qualificatore *const*. In questo caso si deve inizializzare la variabile con l'operatore =.
Di solito le variabili statiche sono private come gli altri attributi.
Le costanti statiche invece di solito sono pubbliche.

Una variabile statica è unica per tutte le istanze della classe; se un oggetto la modifica, la modifica vale per tutti gli oggetti.

Le variabili statiche vengono create la prima volta che si usa la classe (prima di costruire il primo oggetto della classe); devono essere sempre inizializzate nella sezione di implementazione con la sintassi:

```
Tipo Classe::nomeVar = valore;
```

Se le variabili statiche non vengono inizializzate si verifica un errore di compilazione.

Alle variabili statiche si può accedere da qualsiasi metodo, statico o no.

Si può usare una variabile statica per esempio per assegnare un numero consecutivo a ogni oggetto costruito.
Il metodo costruttore può incrementare la variabile.

ESEMPIO 10.13

Si può usare una variabile statica nella classe *Articolo* per contare il numero di articoli creati.

```
//Articolo.h
#include <string>
#include <iostream>
using namespace std;

class Articolo {
    private:
        string descrizione;
        double prezzo;
    public:
        static int numArt;
        void setDescrizione(string);
        void setPrezzo(double);
        string getDescrizione(void) const;
        double getPrezzo(void) const;
        friend ostream& operator<<(ostream&, const Articolo&);
        Articolo(string ="Articolo stardard", double = 4.99);
};//Articolo
```

```
//Articolo.cpp
#include <iostream>
#include "Articolo.h"
using namespace std;

int Articolo::numArt = 0;

Articolo::Articolo(string descr, double pr){
    numArt++;
    descrizione = descr;
    prezzo = pr;
}//Articolo

//altri metodi
```

```
//ProvaArticolo.cpp
#include <iostream>
#include "Articolo.h"
using namespace std;
```

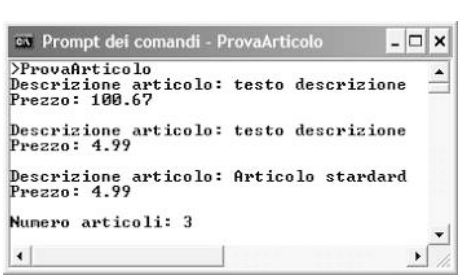


Figura 10.13 *Esecuzione dell'esempio*

```
int main(void){
   ...
    cout << "Numero articoli: " << Articolo::numArt << endl;
    getchar();
    return 0;
}//main
```

Nella creazione di oggetti con il costruttore copia, la variabile *numArt* non viene incrementata; questo perché il costruttore copia ha segnatura diversa rispetto al costruttore a tre parametri che incrementa *numArt*.

Inserendo nell'esempio precedente le righe:

```
Articolo art4 = Articolo(art1); //costruttore copia
Articolo art5 = art2;           //costruttore copia
```

si ottiene ancora 3 come valore di *numArt*.

Per fare in modo che la creazione di oggetti con il costruttore copia incrementi la variabile *numArt*, bisogna ridefinire tale costruttore.

Nella dichiarazione della classe va aggiunta la riga:

```
Articolo(const Articolo&);
```

e nella sezione di implementazione vanno aggiunte le righe:

```
Articolo::Articolo(const Articolo& a){
    numArt++;
    *this = a;
}//Articolo
```

Metodi statici

I **metodi statici** o **di classe** sono metodi che non operano su un oggetto ma per conto di una classe. I metodi statici vengono richiamati usando il **nome della classe** e non un oggetto.

```
NomeClasse::metodo()
```

All'interno della classe i metodi statici possono essere invocati direttamente con il loro nome senza bisogno della classe e dell'operatore ::.

I metodi statici non possono accedere a variabili di istanza di oggetti della classe ma solo a variabili statiche; inoltre possono richiamare solo altri metodi statici della classe (ma possono creare oggetti di una classe, o riceverli come parametri e usarne attributi e metodi).

I metodi statici non hanno il parametro implicito *this* perché non si riferiscono a nessun oggetto.

Un metodo statico non può essere un metodo costante.

Il motivo principale per scrivere metodi statici è fornire elaborazioni esclusivamente numeriche; dal momento che i numeri non sono oggetti non si possono passare come parametri impliciti.

Esempio 10.14

Modifica dell'esempio 10.13. La variabile statica *numArt* è definita privata e per accedere ad essa è stato definito il metodo statico *getNumArt()*.
Vengono riportate solo le parti modificate.

```
class Articolo {
    private:
        string descrizione;
        double prezzo;
        static int numArt;
    public:
        static int getNumArt(void);
       // metodi e costruttori
};//Articolo

int Articolo::getNumArt(void){
    return numArt;
}//getNumArt

int main(void){
    ...
    cout << "Numero articoli: " << Articolo::getNumArt() << endl;
    getchar();
    return 0;
}//main
```

C++ 10.5

Classi interne e classi friend

Le **classi interne** o **classi annidate** sono classi definite all'interno di un'altra classe.

Una classe interna privata, dichiarata nel blocco *private*, non è assolutamente visibile al di fuori della classe che la contiene.
Una classe interna pubblica, dichiarata nel blocco *public*, può essere usata come tipo di una variabile in una classe esterna alla classe che la contiene, specificandone il nome per intero:

```
Classe::ClasseInterna nomeOggetto;
```

I metodi della classe interna possono accedere solo agli attributi pubblici e ai metodi statici pubblici della classe che la contiene; non possono accedere invece agli attributi privati.
È vero anche il contrario: i metodi della classe esterna non possono accedere direttamente agli attributi (e ai metodi) privati della classe interna.
Una classe interna o esterna può accedere alla parte privata dell'altra se è dichiarata ***friend***.

```
class Esterna{
    friend class Interna;
    public:
        class Interna{
            ...
        }//Interna
    ...
}//Esterna
```

Interna accede ad *Esterna*.

```
class Esterna{
    public:
        class Interna{
            friend class Esterna;
            ...
        }//Interna
    ...
}//Esterna
```

Esterna accede ad *Interna*.

Nota

Si possono dichiarare entrambe le classi *friend*.

VERIFICA

QUESTIONARIO

1. Che cosa si intende per stato di un oggetto?
2. Che cosa si intende per comportamento di un oggetto?
3. Quali sono i vantaggi della programmazione a oggetti?
4. Che cosa sono le variabili di istanza?
5. Qual è la differenza tra i metodi e le funzioni classiche?
6. Che cos'è un metodo costruttore?
7. Che cos'è e a che cosa serve un oggetto?
8. Qual è la differenza tra attributi e metodi pubblici e attributi e metodi privati?
9. Che cosa si intende per metodo accessore e metodo modificatore?
10. In che cosa consiste l'esecuzione di un programma a oggetti?
11. Come possono essere interpretate le varie parti della chiamata a un metodo nell'analogia con l'invio di messaggi?
12. Che cosa fanno metodi informativi, interrogativi e imperativi?
13. Qual è il significato del puntatore *this* e quali sono i suoi utilizzi?
14. Che cos'è l'incapsulamento?
15. In che cosa consiste l'information hiding o occultamento delle informazioni?
16. Come vengono rappresentati classi e oggetti in un diagramma UML?
17. Che cosa si intende per spazio degli stati di una classe?
18. Che cos'è l'invariante di classe?
19. Come si può definire il comportamento di una classe?
20. Quali comportamenti si devono evitare?
21. Quando si dice che un metodo accessore o modificatore produce un effetto collaterale?
22. Che cosa sono la precondizione e la postcondizione di un metodo?
23. Che cos'è un metodo costante?
24. Qual è la struttura di una dichiarazione di classe?
25. Qual è la struttura di una implementazione di un metodo?
26. Quali sono i modificatori di accesso per gli attributi e i metodi e quale significato hanno?
27. Che cosa sono i metodi inline?
28. Da quali parti è costituita la dichiarazione di un metodo?
29. Con quale modalità avviene il passaggio dei parametri?
30. Che cosa sono le variabili locali e dove sono visibili?
31. Che cos'è il parametro implicito *this*?
32. Come si può realizzare l'overloading di un metodo?
33. A che cosa serve il sovraccarico dell'operatore <<?
34. Come si definisce un costruttore per la classe?
35. Come si usa il costruttore della classe per creare un oggetto?
36. Che cos'è il distruttore per la classe?
37. Che cosa sono le funzioni amiche di una classe?
38. Qual è la differenza tra metodi e funzioni friend di una classe?
39. Che cosa sono le asserzioni?
40. Nel contesto delle classi, cosa permette l'operatore *new*?
41. Come funzionano la copia e il confronto tra oggetti?
42. Qual è il significato del modificatore *static*?

43. Come si accede ad attributi e metodi statici?
44. Che cosa sono le classi interne?
45. Che cosa sono classi friend?

TEST

1 ..... *Indicare se le seguenti affermazioni sulla programmazione a oggetti sono vere o false.*

V	F	
☐	☐	1) Lo stato di un oggetto è rappresentato dall'insieme dei valori assegnati agli attributi dell'oggetto.
☐	☐	2) Il comportamento di un oggetto è la variazione dei valori degli attributi dell'oggetto.
☐	☐	3) Una classe è costituita da un insieme di oggetti.
☐	☐	4) L'istanziazione di una classe produce un oggetto.
☐	☐	5) Ogni classe ha sempre almeno un metodo costruttore, eventualmente implicito.
☐	☐	6) La vita degli oggetti creati termina alla fine del programma.
☐	☐	7) Si può cambiare il contenuto di un oggetto durante l'esecuzione del programma.
☐	☐	8) Un oggetto deve essere utilizzato per richiamare un metodo dell'oggetto.
☐	☐	9) Una variabile privata è una variabile che può essere usata solo all'interno di un metodo e non dagli altri metodi.
☐	☐	10) È possibile rendere un attributo di sola lettura dichiarando che la variabile corrispondente è in realtà una costante.
☐	☐	11) Un attributo statico o di classe è un attributo che non può essere modificato.
☐	☐	12) A un attributo o a un metodo statico non si accede attraverso un oggetto ma attraverso il nome della classe.
☐	☐	13) L'esecuzione di un programma a oggetti consiste nell'interazione degli oggetti attraverso chiamate di metodi.
☐	☐	14) I parametri di un metodo non possono essere oggetti.
☐	☐	15) Il mittente di un messaggio è l'oggetto su cui viene richiamato il metodo.
☐	☐	16) Il destinatario di un messaggio è l'oggetto su cui viene richiamato il metodo.
☐	☐	17) Un metodo interrogativo restituisce informazioni sullo stato dell'oggetto.
☐	☐	18) La segnatura di un metodo comprende il nome del metodo, tipo e numero di parametri e tipo di valore restituito dal metodo.
☐	☐	19) L'overloading consiste nella definizione di più metodi con lo stesso nome e segnatura diversa.
☐	☐	20) Il riferimento *this* rappresenta l'oggetto stesso.
☐	☐	21) I termini incapsulamento e information hiding sono sinonimi.
☐	☐	22) L'implementazione di un metodo è sempre nascosta.
☐	☐	23) Possono essere occultati sia attributi che metodi.

2 ..... *Indicare se le seguenti affermazioni sulle classi C++ sono vere o false.*

V	F	
☐	☐	1) Si possono dichiarare variabili di istanza all'interno di un metodo.
☐	☐	2) Tutti i metodi devono essere definiti in una classe.
☐	☐	3) Più classi pubbliche possono essere contenute nello stesso file.
☐	☐	4) I metodi della classe in cui è dichiarato un attributo devono utilizzare un oggetto per accedere all'attributo.
☐	☐	5) I metodi della classe in cui è dichiarato un metodo devono utilizzare un oggetto per richiamare il metodo.
☐	☐	6) Il tipo di valore di ritorno *void* indica che il metodo non restituisce alcun valore.
☐	☐	7) Se un metodo non restituisce un valore non può usare l'istruzione *return*.
☐	☐	8) Un metodo può restituire un oggetto.
☐	☐	9) Se il parametro di un metodo è un oggetto il passaggio avviene sempre per riferimento.
☐	☐	10) Se si dà a una variabile locale lo stesso nome di una variabile di istanza si ottiene un errore di compilazione.
☐	☐	11) *this* può essere usato all'interno di una classe per invocare un metodo.
☐	☐	12) Si possono definire metodi ricorsivi.
☐	☐	13) Il valore di ritorno di un metodo fa parte della segnatura del metodo.
☐	☐	14) Il costruttore ha lo stesso nome della classe.
☐	☐	15) Il costruttore può avere dei parametri.
☐	☐	16) Il costruttore restituisce l'oggetto creato.
☐	☐	17) I costruttori di solito hanno accesso privato.
☐	☐	18) Non ci possono essere più costruttori con lo stesso nome, cioè in overloading.
☐	☐	19) Il costruttore copia permette la creazione di un nuovo oggetto partendo da uno esistente.
☐	☐	20) Le variabili di istanza possono essere inizializzate implicitamente.
☐	☐	21) Le variabili locali possono essere inizializzate implicitamente.
☐	☐	22) Un metodo costante restituisce sempre lo stesso valore.
☐	☐	23) È sempre possibile utilizzare l'operatore == per confrontare se due oggetti hanno lo stesso valore.
☐	☐	24) Per creare e distruggere oggetti bisogna utilizzare gli operatori *new* e *delete*.
☐	☐	25) Un attributo *static* non può essere definito *const*.
☐	☐	26) Alle variabili statiche si può accedere solo dai metodi statici.
☐	☐	27) I metodi statici possono accedere solo a variabili statiche e non a variabili di istanza.
☐	☐	28) I metodi statici non hanno il parametro implicito *this*.
☐	☐	29) Le classi interne non possono avere modificatori di accesso.

3 ..... *Segnare tutte le affermazioni esatte riguardo ai modificatori di accesso.*

- ☐ si possono applicare sia ad attributi che a metodi;
- ☐ le classi interne devono sempre essere *public*;
- ☐ se non si specifica un modificatore di accesso il valore di default è *private*;
- ☐ gli attributi di solito vanno dichiarati *private*;
- ☐ il modificatore *protected* indica che l'attributo o il metodo può essere utilizzato da tutte le classi derivate.

Esercizi

1 ..... Elencare alcuni attributi e metodi per le classi e descriverle con un diagramma UML:
- *Libro*,
- *Cerchio*,
- *Automobile*.

2 ..... Sia data le classe *NumeroNaturale* con *n* da 0 a 10.000 (0 <= n < 10.000) con metodi per:
- sommare un altro numero e ottenere ancora un oggetto della classe,
- sottrarre un altro numero e ottenere ancora un oggetto della classe.

Qual è lo spazio degli stati? Quale è l'invariante di classe?
Quali sono le precondizioni e postcondizioni dei due metodi?
Realizzare la classe e garantire le precondizioni dei metodi prima mediante eccezioni e poi mediante asserzioni.

3 ..... Sia data la classe *NumeroIntero* con *n* strettamente compreso tra -10.000 e 10.000 (-10.000 < n < 10.000) con metodi per:
- sommare un altro numero e ottenere ancora un oggetto della classe,
- sottrarre un altro numero e ottenere ancora un oggetto della classe.

Qual è lo spazio degli stati? Quale è l'invariante di classe?
Quali sono le precondizioni e postcondizioni dei due metodi?
Realizzare la classe e garantire le precondizioni dei metodi prima mediante eccezioni e poi mediante asserzioni.

4 ..... Creare una classe *Mesi* che abbia come attributi un array con i nomi dei mesi e un array con il numero di giorni dei mesi e metodi per accedere ai dati.

5 ..... Classe con metodi statici per calcolare:
- la media di tre valori,
- il minimo di tre valori,
- il massimo di tre valori.

6 ..... Classe con metodi statici per calcolare il numero *n* della successione di Fibonacci (il valore di un termine viene calcolato sommando i valori dei due termini precedenti):
- in modo iterativo,
- in modo ricorsivo.

7 ..... Progettare classi per risolvere esempi ed esercizi dei capitoli dal 3 al 7 in modo orientato agli oggetti.

8 ..... Riscrivere tutti gli esempi del capitolo separando in file distinti la dichiarazione della classe, l'implementazione dei metodi e il collaudo.

9 ..... Aggiungere alla classe *Articolo* dell'esempio 10.4 il sovraccarico dell'operatore == attraverso un metodo della classe; due articoli siano uguali se sono uguali descrizione e prezzo.

PROPOSTE DI LAVORO

Realizzare le classi

1 ..... ***Libro***
attributi:
- autore,
- titolo,
- editore.

2 ..... ***Carta*** da gioco
attributi:
- seme,
- valore.

3 ..... ***Proprietà***
attributi:
- nome,
- valore.

4 ..... ***Misura***
attributi:
- unità di misura,
- valore.

5 ..... ***Record***
attributi:
- sport,
- campione,
- valore record.

6 ..... ***Messaggio***
attributi:
- mittente,
- destinatario,
- testo messaggio,
- data.

7 ..... ***Oggetto***
attributi:
- forma,
- colore,
- dimensione: *Misura*.

8 ..... ***Punto*** nel piano
attributi:
- coordinate x e y;

metodi per:
- calcolare la distanza da un punto.

9 ..... ***Quadrato***
attributo: lato;
costruttori:
- senza parametri per quadrato di lato 1,
- con parametro la misura del lato;

metodi per:
- il calcolo del perimetro,
- il calcolo dell'area.

10 ..... ***Triangolo*** rettangolo
attributi:
- base,
- altezza;

metodi per:
- il calcolo del terzo lato,
- il calcolo del perimetro,
- il calcolo dell'area.

11 ..... ***Bit***
attributo: bit (valore 0 o 1);
metodi: Set, Reset e Complementa.

12 ..... ***Rettangolo***
attributi:
- base,
- altezza;

metodi per:
- il calcolo del perimetro,
- il calcolo dell'area,
- verificare se è un quadrato,
- verificare se ha l'area maggiore di un altro rettangolo,
- verificare se è più largo di un altro rettangolo,
- verificare se è più alto di un altro rettangolo,
- verificare se può contenere un altro rettangolo, cioè se è sia più largo che più alto.

13 ..... ***Cerchio***
attributi:
- centro: *Punto*,
- raggio;

costruttori:
- con parametro il raggio,
- con parametri il raggio e il centro;

metodi per:
- il calcolo della circonferenza,
- il calcolo dell'area.

14 ..... ***Data***
attributi:
- giorno,
- mese,
- anno;

metodi per:
- sommare un certo numero di giorni a una data,
- calcolare il numero di giorni di differenza con un'altra data,
- vari metodi in overloading per calcolare la differenza solo in anni (se il parametro è un anno), anni e mesi (se i parametri sono anno e mese) o anni, mesi e giorni (se i parametri sono anno, mese e giorno).
  (*Suggerimento*: restituire un oggetto *DifferenzaDate* con numero di anni, mesi e giorni).

15 ..... ***Persona***
attributi:
- nome,
- data di nascita: *Data*;

metodi:
vari metodi in overloading per calcolare l'età in anni (se il parametro è un anno), in mesi (se i parametri sono anno e mese), in giorni (se i parametri sono anno, mese e giorno).

16 ..... ***Distanza***
attributi:
- kilometri,
- miglia (un miglio = 1,609 Km);

costruttore e metodo per impostare un valore e calcolare l'altro;
metodi che restituiscono i due valori.
Cambiare la classe in modo che abbia una sola variabile di istanza ma si comporti allo stesso modo.
È possibile avere un costruttore per miglia e uno per Km?

17 ..... ***Array***
attributo: un array di numeri;
costruttori:
- per creare un array vuoto,
- per inserire nell'array attributo i valori di un array passato come parametro;

metodi per:
- chiedere dei numeri come input da tastiera e inserirli nell'array,
- stampare l'array,
- sommare all'array un array di dimensione qualsiasi (somma solo gli elementi comuni e restituisce un array di questa dimensione);
- aggiungere un numero in fondo all'array,
- cancellare un elemento dell'array, dato l'indice.
- moltiplicare l'array per un valore;
- effettuare il prodotto scalare con un altro array;
- elevare al quadrato ogni elemento dell'array;
- invertire gli elementi dell'array;
- effettuare una rotazione di ordine n degli elementi dell'array;
- verificare se l'array è uguale a un altro array passato come parametro (cioè se gli array hanno esattamente gli stessi elementi nello stesso ordine); il metodo deve restituire un valore booleano;
- concatenare in fondo all'array un array passato come parametro creando un nuovo array;
- fondere un array (ordinato) con un altro (ordinato) creando un nuovo array.

18 ..... ***Matrice***
attributo: un array di array di numeri;
metodi per:
- leggere la matrice,
- stampare la matrice,
- impostare e restituire un elemento,
- eseguire la somma con un'altra matrice,
- eseguire il prodotto con un'altra matrice,
- restituire una riga della matrice,
- restituire una colonna della matrice.

19 ..... Realizzare il gioco del 15 (non grafico) con 4 array di 4 elementi contenenti i numeri da 1 a 15 e un elemento vuoto con i metodi per spostare un numero tenendo conto delle regole del gioco. Sovraccaricare l'operatore << per la stampa del gioco.

20 ..... Gioco del tris (non grafico); il programma deve disegnare la griglia ad ogni passo con <<, permettere di inserire crocette e circoletti e dichiarare il vincitore.

11 Ereditarietà e polimorfismo

Prerequisiti

- Concetti fondamentali della programmazione a oggetti
- Saper definire classi

Obiettivi

Conoscenze

- Concetti fondamentali relativi all'ereditarietà: ereditarietà, overriding, classi astratte, ereditarietà multipla, polimorfismo
- Diagrammi UML per la rappresentazione della relazione di ereditarietà
- Principi di progettazione di una sottoclasse (conformità di tipo)
- Dichiarazione di sottoclassi: significato dei modificatori in relazione all'ereditarietà, overriding e overloading dei metodi e degli operatori nelle sottoclassi, significato di polimorfismo, cast
- Dichiarazione di classi astratte
- Ereditarietà multipla e derivazione virtuale

Abilità/capacità

- Saper descrivere concettualmente una gerarchia di classi
- Rappresentare le relazioni di ereditarietà tra le classi con un diagramma UML
- Realizzare classi come estensione di altre classi
- Realizzare gerarchie di classi usando classi astratte
- Realizzare gerarchie di classi usando l'ereditarietà multipla

11.1 EREDITARIETÀ

L'**ereditarietà** è la creazione di una nuova classe con le caratteristiche di una classe esistente e con altre caratteristiche specifiche.

La classe più generale, che costituisce la base per l'ereditarietà, viene detta **superclasse** mentre la classe più specializzata, che eredita, viene detta **sottoclasse** (la terminologia deriva dagli insiemi: i sottoinsiemi sono più specializzati); si dice anche che la sottoclasse **estende** la superclasse o **deriva** dalla superclasse (detta anche **classe base**).

Una sottoclasse eredita automaticamente tutti gli attributi e i metodi della superclasse come se fossero definiti al suo interno; inoltre nella sottoclasse si possono aggiungere nuovi attributi e metodi, ridefinire i metodi ereditati per cambiarne il comportamento e aggiungere altre definizioni di metodi esistenti, con segnature diverse.

Per esempio dalla classe *Persona* si potrebbe derivare la classe *Studente* che oltre agli attributi di *Persona* ha *scuola* e *classe* frequentata e oltre ai metodi di *Persona* ha metodi per mostrare e modificare la scuola e mostrare e modificare la classe.

L'ereditarietà consiste quindi nella creazione di sottoclassi a partire da classi più generali, cioè nell'estensione di classi esistenti per crearne altre più specializzate.
Una ragione importante per usare l'ereditarietà è la possibilità di riutilizzare il codice.

Una classe ha due interfacce: una pubblica destinata ai programmatori che usano la classe e una protetta destinata ai programmatori che estendono la classe; entrambe definiscono dei veri contratti e vanno progettate appositamente; se una classe non è stata progettata per essere estesa può essere utilizzata in modo inappropriato.

Nota

Un gruppo di classi correlate per ereditarietà è chiamato **gerarchia di classi**. Una gerarchia di classi ha una struttura ad albero con le classi più generali alla radice, e le più specializzate (le sottoclassi) verso il basso.
Nella programmazione a oggetti è comune realizzare gerarchie di classi.

Si procede costruendo prima le classi più generali e poi quelle più specializzate.

L'**ereditarietà multipla** permette di avere gerarchie di classi con struttura a grafo.

Definizione di sottoclassi

Rispetto agli **attributi** una sottoclasse:

- **eredita gli attributi dalla superclasse** (automaticamente);
- **può definire nuovi attributi** che valgono solo per gli oggetti della sottoclasse; se si definisce un attributo con lo stesso nome di un attributo della superclasse si hanno due variabili con lo stesso nome, una messa in ombra: è visibile solo quella nuova, mentre all'altra non si può accedere.

Rispetto ai **metodi** una sottoclasse:

- **eredita i metodi della superclasse**; i metodi vengono ereditati automaticamente e si possono applicare agli oggetti della sottoclasse;
- **può sovrascrivere metodi della superclasse** (**overriding**), cioè definire metodi con lo stesso nome e stesso numero e tipo di parametri dei metodi della superclasse, cioè con la stessa segnatura; se un metodo ha la stessa segnatura di un metodo della superclasse si

dice che è sovrascritto, ridefinito o superato; il nuovo metodo prevale: ogni volta che si applica il metodo a un oggetto della sottoclasse viene eseguito quello della sottoclasse, non l'originale; l'overriding quindi permette di cambiare il comportamento di un metodo nella sottoclasse (e quindi anche eventualmente di neutralizzare un metodo);

- **può aggiungere metodi con lo stesso nome ma segnatura diversa** (**overloading**);
- **può definire nuovi metodi**.

Tutti i metodi definiti nella sottoclasse (nuovi, aggiunti per overloading o sovrascritti per overriding) si possono applicare solo a oggetti della sottoclasse.

I metodi ridefiniti di solito chiamano al loro interno il metodo corrispondente della superclasse, per eseguire le operazioni della superclasse e aggiungerne altre.

RIASSUMENDO

Se S (sottoclasse) eredita da C (superclasse):

- un metodo di C non può chiamare un metodo di S,
- un metodo di C non può fare riferimento a variabili istanza di S,
- un metodo di S può chiamare un metodo di C,
- un metodo di S può fare riferimento a variabili istanza di C (e naturalmente anche di S).

Se *Studente* deriva da *Persona*:
i metodi di *Persona* non possono chiamare metodi di *Studente*,
i metodi di *Persona* non possono utilizzare le variabili istanza di *Studente*,
i metodi di *Studente* possono chiamare metodi di *Persona*,
i metodi di *Studente* possono utilizzare le variabili istanza di *Persona* (e di *Studente*).

CLASSI ASTRATTE

Le **classi astratte** sono classi che costituiscono solo dei modelli per la creazione di sottoclassi; contengono dei metodi lasciati appositamente senza implementazione, cioè per cui viene scritta soltanto la dichiarazione (**metodi virtuali puri** o **astratti**).

Nota

Se una classe ha un metodo virtuale puro è astratta.

Le classi astratte non possono essere utilizzate per creare oggetti, cioè non possono essere istanziate.
Servono solo per poter creare delle sottoclassi partendo da una superclasse comune; la superclasse astratta permette di stabilire quali metodi devono essere definiti nelle sottoclassi.
Le sottoclassi ereditano i metodi virtuali puri e li devono definire tutti, cioè devono fornire un'implementazione per tutti i metodi.
Le sottoclassi che non definiscono tutti i metodi virtuali puri ereditati sono a loro volta astratte.

Come esempio si può considerare una gerarchia di classi per descrivere il comportamento degli animali:

```
Animale
    Mammifero
        Cane
        Delfino
    Uccello
        Aquila
        Pinguino
```

La classe *Animale* per sua natura è astratta, infatti non esistono animali generici (lo stesso vale per le classi *Mammifero* e *Uccello*).
La classe *Animale* potrebbe contenere i metodi virtuali puri *siMuove()* che indica in che modo si muove l'animale e *verso()* che indica qual è il verso dell'animale, per obbligare ogni classe relativa a uno specifico animale ad implementarli.
Una classe deve implementare questi metodi perché sia possibile creare un oggetto della classe. Quindi per esempio si può creare uno specifico oggetto *cane* della classe *Cane* solo se la classe *Cane* implementa i metodi *siMuove()* e *verso()*.

Ereditarietà multipla

L'ereditarietà multipla permette ad una sottoclasse di **estendere più classi**, cioè di ereditare attributi e metodi da più di una classe.

L'ereditarietà multipla presenta dei problemi perché si potrebbero ereditare da classi diverse attributi e metodi con lo stesso nome, ma in contrasto tra di loro.

Il **C++** per risolvere i problemi dell'ereditarietà multipla introduce il concetto di **derivazione virtuale**.

11.2 POLIMORFISMO

Il **polimorfismo** è la possibilità di richiamare su vari oggetti uno stesso metodo che agisce in modo diverso in base al tipo di oggetto su cui è richiamato.
Il tipo effettivo di un oggetto determina il metodo da chiamare: la stessa elaborazione funziona per oggetti di forme diverse e si adatta alla natura degli oggetti.

Il polimorfismo si può vedere in due modi:

- una variabile può contenere riferimenti a oggetti di classi diverse in momenti diversi;
- un singolo nome di metodo (o di attributo) può essere definito in più di una classe e assumere implementazioni diverse in ogni classe; in ogni momento verrà scelto il metodo adatto in base al tipo dell'oggetto su cui viene richiamato.

Quindi è una forma di polimorfismo anche la scelta del metodo nel caso di overloading; alcuni testi parlano di polimorfismo debole nel caso dell'overloading e di polimorfismo forte nel caso dell'overriding.

Nota

Il polimorfismo è strettamente legato all'ereditarietà. Ogni oggetto del tipo di una sottoclasse può anche essere visto come del tipo di una qualsiasi delle superclassi; si può quindi assegnare un riferimento a un oggetto di una sottoclasse a una variabile dichiarata del tipo di una delle superclassi; se si usa questa variabile per richiamare un metodo che è ridefinito nella sottoclasse viene richiamato il metodo della sottoclasse (il tipo reale dell'oggetto) e non quello della superclasse (il tipo della variabile).

Per esempio se il metodo *stampaDati()* è definito nella classe *Persona* in modo da stampare nome, data di nascita e indirizzo e ridefinito in *Studente* in modo da stampare anche scuola e classe frequentata, se un puntatore *persona* di tipo *Persona* punta a un oggetto della classe *Studente* la chiamata *persona->stampaDati()* richiamerà il metodo di *Studente*.

Nell'esempio della gerarchia degli animali un puntatore *animale* di tipo *Animale* può puntare a un oggetto di una qualsiasi delle sottoclassi, per esempio a un oggetto della classe *Cane* o *Aquila*; la chiamata *animale->verso()* richiama il metodo adeguato.

In pratica al momento della scrittura del codice si può non conoscere esattamente la classe di cui verrà chiamato il metodo, cioè l'oggetto che invia il messaggio non conosce l'esatta classe del destinatario a cui sta inviando il messaggio.
Quale metodo eseguire viene stabilito in base al tipo effettivo del riferimento contenuto nella variabile.

Il polimorfismo in genere è implementato tramite il legame dinamico (**dynamic binding**): il codice da eseguire viene determinato al momento dell'esecuzione e non in fase di compilazione.

11.3 DIAGRAMMI UML DELL'EREDITARIETÀ

In un diagramma di classi si rappresenta l'**ereditarietà** con una freccia con la punta a triangolo vuoto che va dalla sottoclasse alla superclasse.

Ogni classe può essere rappresentata da un rettangolo con soltanto il nome.

Se da una classe derivano più classi si può rappresentare l'ereditarietà con una freccia separata per ogni classe o con un'unica freccia che si suddivide in più rami.

Nota La gerarchia delle classi si può rappresentare anche come solo testo, con i livelli indicati dall'indentazione.

Nota Nel caso di ereditarietà multipla da una classe possono partire più frecce dirette alle superclassi.

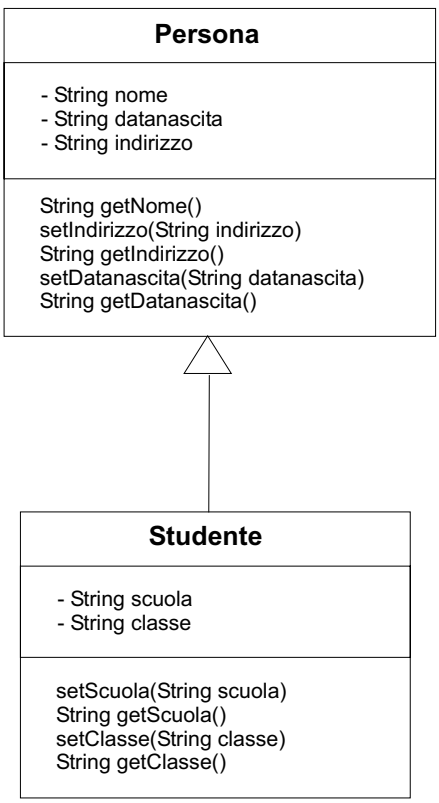


Figura 11.1 *Notazione UML per l'ereditarietà*

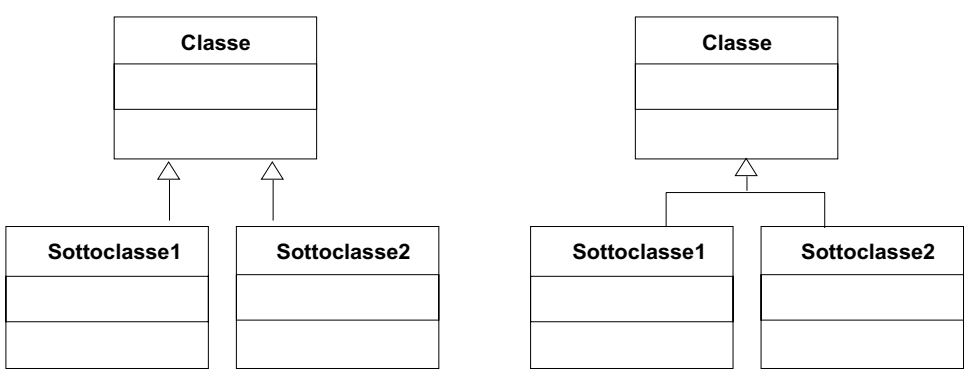


Figura 11.2 *Alternative per la rappresentazione dell'ereditarietà*

Quando da una classe derivano più sottoclassi la superclasse risulta partizionata nelle sottoclassi.
La freccia dell'ereditarietà può riportare le **proprietà di partizionamento** (tra parentesi graffe):

- **disjoint** (disgiunto) o **overlapping** (sovrapposto) indica se le sottoclassi sono disgiunte (cioè se nessun elemento può appartenere contemporaneamente a più di una sottoclasse) o no; il caso più comune è che siano disgiunte;

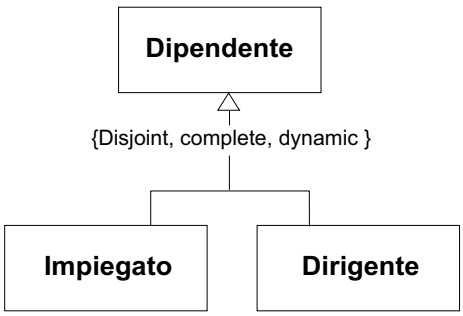


Figura 11.3 *Rappresentazione dell'ereditarietà con le proprietà di partizionamento*

> Nella gerarchia degli animali tutte le sottoclassi degli animali sono disgiunte.

- **complete** (completo) o **incomplete** (incompleto) indica se ci sono tutte le possibili sottoclassi o no, cioè se le sottoclassi comprendono tutti i possibili elementi o se ci possono essere elementi che non appartengono a nessuna delle sottoclassi (a volte si può avere un partizionamento incompleto perché si definiscono solo le sottoclassi necessarie all'applicazione);

> Nella gerarchia degli animali il partizionamento è completo solo se si rappresentano tutti gli animali.

- **dynamic** (dinamico) o **static** (statico) indica se un elemento può passare da una sottoclasse all'altra.

> Nella gerarchia
> ```
> Dipendente
>     Impiegato
>     Dirigente
> ```
> un impiegato può diventare dirigente o anche viceversa.

Si può specificare anche il discriminatore di partizionamento cioè l'attributo in base al quale un elemento appartiene all'una o all'altra classe.

> Nella gerarchia degli animali l'attributo *classe* indica se un animale appartiene alla classe *Mammifero* o *Uccello*.

Una superclasse completamente partizionata in sottoclassi può diventare una classe astratta perché tutti gli oggetti appartengono a una delle sottoclassi.

11.4 PRINCIPI DI PROGETTAZIONE

Sottoclassi

Nella progettazione delle sottoclassi, per quanto riguarda invariante di classe, spazio degli stati e comportamento, bisogna tenere presente le indicazioni seguenti.

L'**invariante** di ogni sottoclasse deve essere restrittiva almeno come quella della superclasse (cioè più restrittiva o equivalente).

Se S è una sottoclasse di C lo **spazio degli stati** di S:

- deve comprendere tutti gli attributi di C, ma può anche includerne altri; la sottoclasse cioè può estendere lo spazio degli stati;
- deve essere delimitato da (o confinato in) quello di C, cioè gli attributi di S che sono anche attributi di C devono usare solo valori che sono ammessi dagli attributi di C; per esempio, se un attributo di C ha un intervallo di validità, i valori dello stesso attributo di S devono rientrare in questo intervallo o in un sottointervallo.

Ogni metodo della superclasse ha un metodo corrispondente nella sottoclasse con lo stesso nome e segnatura (eventualmente ereditato).
La **precondizione** di ogni metodo ridefinito nella sottoclasse deve essere più debole di quella della superclasse (o equivalente) e la **postcondizione** di ogni metodo della sottoclasse deve essere più restrittiva di quella della superclasse (o equivalente).

Per i metodi di modifica è necessario rispettare anche il principio del **comportamento chiuso**, cioè il comportamento ereditato dalla superclasse deve rispettare l'invariante della sottoclasse.

> Se la classe *Poligono* ha un metodo per aggiungere un vertice e la sottoclasse *Triangolo* lo usa, non ha un comportamento chiuso perché un triangolo a cui viene aggiunto un vertice non è più un triangolo. Quindi qualcosa non va nella progettazione della gerarchia.

In pratica una gerarchia di classi ben costruita deve rispettare il principio di **conformità di tipo**. Una classe è l'implementazione di un tipo. Ogni sottoclasse deve essere anche un sottotipo. Il principio di conformità di tipo dice che se S è una sottoclasse di C, S deve essere conforme a C, cioè si deve poter fornire un oggetto di tipo S ogni volta che è richiesto un oggetto di tipo C, mantenendo la correttezza per qualsiasi metodo eseguito.

Polimorfismo

Una variabile polimorfa può puntare in momenti diversi a oggetti di classi diverse.
L'**ambito di polimorfismo** di una **variabile** è l'insieme delle classi degli oggetti a cui può puntare.

Un metodo polimorfo è definito per molte classi diverse.
L'**ambito di polimorfismo** di un **metodo** è l'insieme delle classi per cui il metodo è definito.

Perché le cose funzionino correttamente bisogna che l'ambito della variabile usata come riferimento per la chiamata di un metodo rientri nell'ambito del metodo richiamato, altrimenti il metodo potrebbe non essere definito per l'oggetto corrente.

Possibili errori

Alcuni comuni errori di progettazione sono:

- invertire la gerarchia delle classi in base a quella che sembra la gerarchia nella realtà;

  > Dato che un dirigente gerarchicamente nella realtà è più importante di un dipendente ci si potrebbe confondere e derivare *Dipendente* da *Dirigente* invece che viceversa.

- confondere l'ereditarietà con qualche tipo di associazione;

  > Un cerchio ha un punto come centro ma non è un punto.

- confondere classi e istanze.

  > Questo problema si verifica quando si devono considerare contemporaneamente gruppi come specie o società e individui come animali o dipendenti.

Ereditarietà: dichiarazione di una sottoclasse

C++ 11.1

La dichiarazione di una sottoclasse ha la forma:

```
class NomeClasse : public NomeSuperclasse {
    // attributi
    // metodi
};
```

Il modificatore ***public*** indica che la sottoclasse è in grado di accedere alle risorse della sezione *public* della superclasse. In questo caso si parla di **derivazione pubblica**.
Esistono, anche se meno diffuse, le **derivazioni private** e le **derivazioni protette** rispettivamente indicate dai modificatori di accesso ***private*** e ***protected***.

Attributi e metodi

Attributi e metodi si dichiarano come di consueto.

Ogni attributo e metodo può avere il modificatore *static* e il qualificatore *const* e può essere scritto in una delle sezioni definite dai modificatori di accesso *private*, *public* e *protected*.

Il modificatore di accesso ***protected*** permette ad una risorsa della superclasse di essere visibile alle sottoclassi ma non alle altre classi.

Gli attributi di solito sono dichiarati privati; se la sottoclasse deve poter modificare gli attributi conviene rendere disponibile con accesso *protected* un metodo di modifica.

Tabella 11.1 *Significato dei modificatori di accesso*

Accesso	Visibilità
public	a tutte le classi
private	solo all'interno della classe
protected	a tutte le sottoclassi

Attributi e metodi in una sottoclasse

In una derivazione pubblica una sottoclasse eredita tutti gli attributi e i metodi della superclasse dichiarati *public* e *protected* che rimangono tali.
In una derivazione privata una sottoclasse eredita tutti gli attributi e i metodi della superclasse dichiarati *public* o *protected* ma questi diventano tutti *private*.
In una derivazione protetta una sottoclasse eredita tutti gli attributi e i metodi della superclasse dichiarati *public* e *protected* ma questi diventano rispettivamente *protected* e *private*.

Il modificatore ***static*** rende l'attributo o il metodo statico o di classe; attributi e metodi statici protetti sono accessibili in ogni classe derivata.

Overriding e overloading

Overriding

In una sottoclasse si può definire un metodo con lo stesso nome e stesso numero e tipo di parametri della superclasse (**overriding**); questo serve per sovrascrivere (o superare) il metodo, cambiandone il comportamento all'interno della sottoclasse; si può sovrascrivere un metodo anche in modo che non faccia nulla per togliere una funzionalità alla classe.

Quando si sovrascrive un metodo i parametri e il tipo di ritorno devono coincidere; si possono cambiare i nomi dei parametri ma non il numero, l'ordine e il tipo.

Overloading

Nella sottoclasse si possono aggiungere definizioni diverse (con diverso tipo o numero di parametri) a un metodo della superclasse per ampliarne le funzionalità.
Quando il metodo viene richiamato viene scelta la definizione opportuna tra tutte quelle disponibili nella classe e nelle superclassi in base al numero e tipo dei parametri passati.

Richiamo di un metodo della superclasse

In generale per richiamare all'interno di un metodo di una sottoclasse un metodo della superclasse, viene usato l'operatore ::

```
NomeSuperclasse::metodo(parametri);
```

Di solito nell'overriding il metodo ridefinito aggiunge istruzioni al metodo della superclasse: richiama attraverso l'operatore :: il metodo della superclasse e aggiunge (prima o dopo) altre istruzioni.

```
TipoDiRitorno nomeMetodo(parametri) {
    ...
    NomeSuperclasse::nomeMetodo(parametri);
    ...
}//nomeMetodo
```

Analogamente il costruttore della sottoclasse può richiamare il costruttore della superclasse; solitamente viene usata la notazione mediante lista di inizializzazione:

```
Sottoclasse::Sottoclasse(parametri) : Superclasse(parametri) {
    //altre inizializzazioni
}
```

Se il costruttore della sottoclasse non richiama esplicitamente un costruttore della superclasse, per prima cosa viene chiamato automaticamente il costruttore predefinito della superclasse; se la superclasse non ha un costruttore senza parametri il compilatore genera un errore.

Esempio 11.1

Creazione della classe *Persona* (che gestisce nome, data di nascita e indirizzo) e della sottoclasse *Studente* (che aggiunge scuola e classe frequentata).

Nota

Le dichiarazioni delle classi, le implementazioni e il collaudo sono in file distinti.
È possibile riunire tutto il codice in un unico file.

```
//Persona.h
#include <string>
using namespace std;
class Persona {
    private:
        string nome;
        string dataNascita;
        string indirizzo;
    public:
        void setIndirizzo(string);
        void setDataNascita(string);
        string getIndirizzo(void) const;
        string getNome(void) const;
```

```
        string getDataNascita(void) const;
        void stampaDati(void) const;
        Persona(string ="***", string ="***", string ="***");
};//Persona
- - - - - - - - - -
//Persona.cpp
#include <iostream>
#include "Persona.h"
using namespace std;

Persona::Persona(string nome, string dataNascita, string
indirizzo){
    this->nome = nome;
    this->dataNascita = dataNascita;
    this->indirizzo = indirizzo;
}//Persona

string Persona::getNome(void) const{
    return nome;
}//getNome

void Persona::setIndirizzo(string indirizzo){
    this->indirizzo = indirizzo;
}//setIndirizzo

string Persona::getIndirizzo(void) const{
    return indirizzo;
}//getIndirizzo

void Persona::setDataNascita(string dataNascita){
    this->dataNascita = dataNascita;
}//setDataNascita

string Persona::getDataNascita(void) const{
    return dataNascita;
}//getDataNascita

void Persona::stampaDati(void) const{
    cout << nome << " ";
    cout << dataNascita << " ";
    cout << indirizzo << endl;
}//stampaDati
- - - - - - - - - -
//Studente.h
#include "Persona.h"
class Studente : public Persona {
    private:
        string scuola;
        string classe;
    public:
        void setScuola(string);
        void setClasse(string);
        string getScuola(void) const;
        string getClasse(void) const;
        void stampaDati(void) const;
```

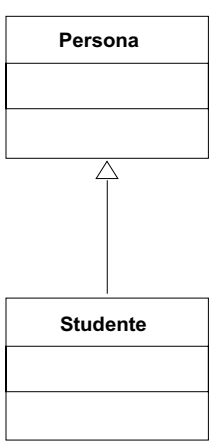


Figura 11.4 *Diagramma UML delle classi dell'esempio*

```
        Studente(string ="###", string ="###", string ="###");
};//Studente
- - - - - - - - - - -
//Studente.cpp
#include <iostream>
#include <string>
#include "Studente.h"
using namespace std;

Studente::Studente(string nome, string scuola, string classe) :
Persona(nome){
    this->scuola = scuola;
    this->classe = classe;
}//Studente

string Studente::getScuola(void) const{
    return scuola;
}//getScuola

void Studente::setScuola(string scuola){
    this->scuola = scuola;
}//setScuola

string Studente::getClasse(void) const{
    return classe;
}//getClasse

void Studente::setClasse(string classe){
    this->classe = classe;
}//setClasse

void Studente::stampaDati(void) const{
    Persona::stampaDati();
    cout << scuola << " ";
    cout << classe << endl;
}//stampaDati
- - - - - - - - - - - - - - -
//ProvaStudente.cpp
#include "Studente.h"

int main(void){
    Studente studente = Studente("Mario Rossi", "Liceo Manzoni");
    studente.setDataNascita("11/05/1993");
    studente.setIndirizzo("via Roma 5, Milano");
    studente.setClasse("III C");
    studente.stampaDati();
    getchar();
    return 0;
}//main
```

Figura 11.5 *Esecuzione dell'esempio*

La classe *Studente* estende sia i dati che il comportamento della superclasse *Persona*: aggiunge gli attributi *classe* e *scuola* e i metodi *get()* e *set()* per questi attributi.

Nota

La sottoclasse potrebbe anche restringere il comportamento esistente, per esempio limitando la data di nascita ad un certo intervallo.

Il costruttore di *Studente* richiama il costruttore di *Persona* e poi aggiunge i nuovi dati.

La classe *Studente* ridefinisce il metodo *stampaDati()* aggiungendo funzionalità, per ottenere un comportamento adeguato a *Studente*: richiama prima il metodo corrispondente di *Persona* mediante l'operatore :: e poi stampa i nuovi dati (il metodo *stampaDati()* ereditato tratta solo persone perché non conosce le variabili *scuola* e *classe*).

Assegnazione di oggetti di classi derivate

Un oggetto di una sottoclasse è visto come un oggetto della superclasse (chiamato anche sottooggetto) a cui sono aggiunti nuovi attributi.

L'assegnazione di un oggetto della superclasse a una variabile del tipo della sottoclasse dà un errore di compilazione.

```
Persona persona1, persona2;
Studente studente1, studente2;
persona1 = studente1; //assegnazione corretta
studente2 = persona2; //errore di compilazione
```

È possibile l'assegnazione diretta tra oggetti di classi derivate solo dalla sottoclasse alla superclasse. In tal caso alla variabile del tipo della superclasse viene assegnato il sottooggetto della sottoclasse.

Nel caso dell'overriding il metodo scelto è sempre quello della superclasse.

```
#include "Studente.h"

int main(void){
    Persona persona = Studente("Mario Rossi", "Liceo Manzoni");
    persona.setDataNascita("11/05/1993");
    persona.setIndirizzo("via Roma 5, Milano");
    persona.stampaDati();
    getchar();
    return 0;
}//main
```

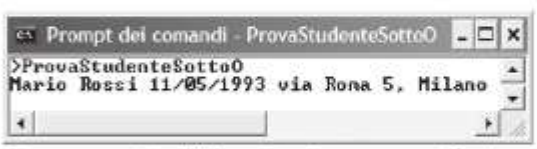


Figura 11.6 *Esecuzione dell'esempio*

A una variabile di tipo *Persona* viene assegnato un oggetto di tipo *Studente*; quando si richiama il metodo *stampaDati()* viene richiamato il metodo della classe *Persona*.

Usando una variabile del tipo della superclasse, anche se contiene un oggetto della sottoclasse, si possono richiamare solo i metodi definiti nella superclasse e non quelli definiti a partire dalla sottoclasse.

Per esempio non si può fare

```
persona.setClasse("IV A");
```

Polimorfismo

Quando si lavora con una gerarchia di classi si può utilizzare il polimorfismo.
Il **polimorfismo** permette di avere lo stesso metodo definito in classi diverse e di usare una stessa istruzione per richiamare il metodo opportuno, in base all'oggetto che si sta utilizzando.

In C++ il polimorfismo è gestito tramite puntatori o tramite riferimenti.

In base al polimorfismo, in C++ un puntatore del tipo della superclasse può puntare a un oggetto di una sottoclasse.

I puntatori del tipo della sottoclasse possono essere usati dal codice scritto per la superclasse. Ogni volta che è richiesto un puntatore del tipo della superclasse si può fornire un puntatore del tipo della sottoclasse, cioè un puntatore di un tipo derivato può essere usato dovunque sia richiesto un puntatore di un supertipo.

Per esempio se il parametro di un metodo è del tipo della superclasse si può fornirgli un puntatore del tipo della sottoclasse.

> Un puntatore di tipo *Studente* può essere utilizzato per puntare sia a un oggetto *Studente* che a un oggetto *Persona*.
> In pratica si può usare un puntatore di tipo *Studente* ogni volta che è richiesto un puntatore di tipo *Persona*.

Se si richiama un metodo usando un puntatore del tipo della superclasse che punta a un oggetto del tipo della sottoclasse, la scelta del metodo avviene in base al tipo del puntatore.

Affinché la scelta del metodo avvenga in base al tipo puntato e non al tipo del puntatore il metodo della superclasse e tutti i metodi sovrascritti delle sottoclassi devono essere definiti mediante la clausola ***virtual***.

```
virtual TipoDiRitorno nomeMetodo(parametri);
```

Questo permette di agire su puntatori del tipo delle sottoclassi, senza bisogno di sapere al momento della stesura del codice quale sarà la sottoclasse utilizzata.

Nota

Per accedere a una variabile di istanza viene usato il tipo del puntatore e non il tipo puntato come per i metodi.

Nel caso dell'overloading la scelta del metodo è fatta dal compilatore prima dell'esecuzione in base ai parametri (selezione anticipata); nel caso dell'overriding invece la scelta del metodo si basa sul tipo del parametro implicito noto solo durante l'esecuzione (**selezione posticipata**).

Esempio 11.2

Modifica dell'esempio 11.1. Invocazione di metodi delle sottoclassi attraverso un puntatore del tipo della superclasse. Le classi *Persona* e *Studente* hanno come unica variazione il metodo *mostraDati()* dichiarato *virtual*.

```
//ProvaStudentePoli.cpp
#include "Studente.h"

int main(void){
    Studente studente = Studente("Mario Rossi", "Liceo Manzoni");
    studente.setDataNascita("11/05/1993");
    studente.setIndirizzo("via Roma 5, Milano");
    studente.setClasse("III C");
    Persona* persona = &studente;
    persona->stampaDati();
    getchar();
    return 0;
}//main
```

Figura 11.7 *Esecuzione dell'esempio*

Un puntatore di tipo *Persona* punta a un oggetto di tipo *Studente*; quando si richiama il metodo *stampaDati()* viene richiamato il metodo della classe *Studente* anche se il puntatore è di tipo *Persona*. Se i metodi *stampaDati()* non sono dichiarati *virtual*, viene richiamato il metodo della classe *Persona*.

Per assegnazioni che coinvolgono le superclassi è necessario usare i puntatori o i riferimenti. Non tutte le assegnazioni sono possibili; alcune sono particolarmente pericolose perché non danno errori di compilazione ma di esecuzione.

```
Persona persona("Mario Rossi", "11/05/1991");
Studente studente("Antonio Bianchi","ITIS Severi");
Persona* pPers = &persona;
Studente* pStud = &studente;
pPers->stampaDati();
pStud->stampaDati();
pPers = pStud;
pPers->stampaDati();
```

L'assegnazione tra i due puntatori è possibile perché *pPers* è un puntatore alla superclasse; i metodi *stampaDati()* sono dichiarati *virtual* quindi il metodo *stampaDati()* richiamato con il puntatore *pPers* è quello della classe *Studente*.

```
Persona* pPerSt = &studente;
Studente* pStud2 = static_cast<Studente*>(pPerSt);
pPerSt->stampaDati();
```

In questo caso il cast è necessario ma non sufficiente: il puntatore *pPerSt* punta ad un oggetto della classe *Studente*.

```
Persona perStud = Studente("Mario Rossi", "Liceo Manzoni");
Studente *pPerStud = static_cast<Studente*>(&perStud);
pPerStud->setClasse("IV A");
pPerStud->stampaDati();
```

Questo codice non dà errori di compilazione ma in esecuzione dà risultati imprevedibili.

Nota

Le stesse conclusioni si possono trarre usando i riferimenti anziché i puntatori.

Cast tra classi

Per quanto riguarda il **cast** tra classi si possono avere tre casi:

- **dal tipo di una sottoclasse al tipo di una superclasse**: il cast da un tipo più esteso a uno meno esteso viene detto ampliamento (widening) o forzatura verso l'alto (casting up) o forzatura sicura (safe casting); è sempre possibile; può essere eseguito sia in modo esplicito che in modo implicito, infatti si può usare un tipo dove è richiesto un supertipo;
- **dal tipo di una superclasse al tipo di una sottoclasse**: il cast da un tipo meno esteso a uno più esteso viene detto restringimento (narrowing) o forzatura verso il basso (casting down) o forzatura insicura (unsafe casting); si può fare solo se si è sicuri che l'oggetto sia effettivamente del tipo della sottoclasse e solo in modo esplicito;
- tra classi che non fanno parte della stessa gerarchia: non è possibile; il codice non viene nemmeno compilato.

Overloading dell'operatore << nelle classi derivate

Una classe derivata eredita anche le funzioni friend della superclasse.

È utile sovraccaricare l'operatore << in una superclasse con una funzione friend in modo che la funzione possa essere ereditata e utilizzata nella sottoclasse.

ESEMPIO 11.3

Nelle classi *Persona* e *Studente* invece del metodo *stampaDati()* si può sovraccaricare l'operatore <<.

Nella classe ***Persona***

```
friend ostream& operator<<(ostream&, const Persona&);

ostream& operator<<(ostream& os, const Persona& pers){
    os << pers.nome << " " << pers.dataNascita << " ";
    os << pers.indirizzo << "\n";
    return os;
}//operator<<
```

Nella classe ***Studente***

```
friend ostream& operator<<(ostream&, const Studente&);

ostream& operator<<(ostream& os, const Studente& st){
    Persona pers = st;
    os << pers << st.scuola << " " << st.classe << "\n";
    return os;
}//operator<<
```

C++ 11.2

CLASSI ASTRATTE

Quando si estende una classe si possono ridefinire o meno i metodi della superclasse.
Per rendere obbligatoria la ridefinizione di un metodo in tutte le sottoclassi si può dichiarare il metodo virtuale con il modificatore ***virtual***.

```
virtual TipoDiRitorno nomeMetodo (parametri);
```

I metodi virtuali non possono essere privati.
Costruttori, metodi statici e funzioni friend non possono essere virtuali.

I metodi virtuali non sono solo definiti ma anche implementati.
Una classe contenente un metodo virtuale può essere usata per istanziare oggetti.

Se il metodo viene solo dichiarato ma non implementato è detto **metodo virtuale puro** o **metodo astratto**.

```
virtual TipoDiRitorno nomeMetodo (parametri) =0;
```

In genere si dichiara virtuale puro un metodo se non esiste una buona implementazione predefinita per la superclasse e quindi se solo le sottoclassi possono sapere come definire il metodo in modo appropriato.

Una classe che ridefinisce un metodo può trasformare il metodo da concreto a virtuale puro; ciò è utile quando l'implementazione di default di una classe non è valida per una parte della gerarchia delle classi.

Una **classe** che ha almeno un metodo virtuale puro (perché lo dichiara o lo eredita) è chiamata classe **astratta**.

Le classi astratte servono soltanto come basi da cui derivare sottoclassi che definiscono i metodi astratti.

Se una classe è astratta non si possono creare oggetti di quella classe (viene segnalato un errore di compilazione; manca la definizione di alcuni metodi che potrebbero essere richiamati).

Variabili del tipo della classe astratta

Non si possono costruire oggetti di una classe astratta ma si possono usare puntatori (o riferimenti) del tipo della classe astratta per puntare (o riferirsi) a oggetti delle sottoclassi.

Esempio 11.4

Definizione della classe *Animale*, della sottoclasse *Mammifero* e delle sottoclassi di *Mammifero* *Cane* e *Delfino*.

Nota

Le dichiarazioni (e le implementazioni) delle classi *Cane* e *Delfino* sono nello stesso file perché entrambe derivate della classe *Mammifero*. La separazione in file distinti con la conseguente doppia inclusione del file *Mammifero.h* non è accettata da molti compilatori.

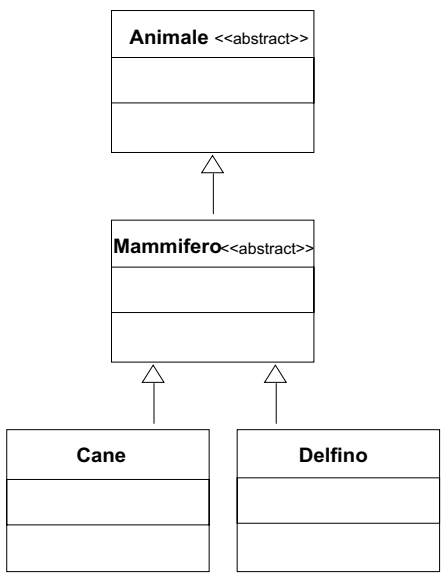


Figura 11.8 *Diagramma UML delle classi dell'esempio*

```
//Animale.h
#include <string>
using namespace std;
class Animale {
    private:
        string regno;
        string classe;
        string specie;
    public:
        virtual string siMuove(void) const =0;
        virtual string verso(void) const =0;
        void stampaDati(void) const;
        Animale(string ="***", string ="***");
};//Animale
```

```
//Animale.cpp
#include <iostream>
#include <string>
#include "Animale.h"
using namespace std;

Animale::Animale(string c, string s){
    regno = "animale";
    classe = c;
    specie = s;
}//Animale
```

```
void Animale::stampaDati(void) const{
    cout << "regno: " << regno;
    cout << ", classe: " << classe;
    cout << ", specie: " << specie << endl;
}//stampaDati
- - - - - - - - - - -
//Mammifero.h
#include "Animale.h"
class Mammifero : public Animale {
    public:
        virtual string siMuove(void) const;
        Mammifero(string ="###");
};//Mammifero
- - - - - - - - - - - -
//Mammifero.cpp
#include "Mammifero.h"

Mammifero::Mammifero(string s) : Animale("mammifero", s){
}//Mammifero

string Mammifero::siMuove(void) const{
    return "corre";
}//siMuove
- - - - - - - - - - - -
//CaneDelfino.h
#include "Mammifero.h"

class Cane : public Mammifero {
    private:
        string razza;
    public:
        virtual string verso(void) const;
        Cane(string ="---");
};//Cane

class Delfino : public Mammifero {
    public:
        virtual string verso(void) const;
        virtual string siMuove(void) const;
        Delfino(void);
};//Delfino
- - - - - - - - - - - - - -
//CaneDelfino.cpp
#include "CaneDelfino.h"

Cane::Cane(string r) : Mammifero("cane"){
    razza = r;
}//Cane

string Cane::verso(void) const{
    return "abbaia";
}//verso

Delfino::Delfino(void) : Mammifero("delfino"){
}//Delfino
```

```
string Delfino::verso(void) const{
    return "nessuno";
}//verso
string Delfino::siMuove(void) const{
    return "nuota";
}//siMuove
- - - - - - - - - - - - -
//ProvaAnimali.cpp
#include <iostream>
#include <string>
#include "CaneDelfino.h"
using namespace std;
int main(void){
    Cane cane;
    cane.stampaDati();
    cout << cane.siMuove() << endl;
    cout << cane.verso() << endl;
    Delfino delfino;
    delfino.stampaDati();
    cout << delfino.siMuove() << endl;
    cout << delfino.verso() << endl;
   ...
}//main
```

Figura 11.9 *Esecuzione dell'esempio*

La classe *Animale* prevede due metodi astratti *siMuove()* e *verso()*; ogni specifico animale deve ridefinire questi metodi nel modo più opportuno.
La classe *Mammifero* ridefinisce il metodo *siMuove()* dandone una implementazione perché la maggior parte dei mammiferi corrono ma non ridefinisce *verso()* e quindi lo eredita astratto, perciò anche la classe *Mammifero* è astratta.
La classe *Cane* dà una implementazione per il metodo *verso()* ed eredita *siMuove()* da *Mammifero* e quindi è una classe concreta da cui si possono costruire oggetti.
La classe *Delfino* ridefinisce tutti e due i metodi in modo da sostituire il metodo *siMuove()*.

Per accedere a oggetti delle sottoclassi si possono usare anche puntatori o riferimenti del tipo della classe astratta *Animale*.

```
//ProvaAnimaliPR.cpp
#include <iostream>
#include <string>
#include "CaneDelfino.h"
using namespace std;
int main(void){
    Delfino delfino;
    delfino.stampaDati();
    cout << delfino.siMuove() << endl;
    cout << delfino.verso() << endl;
    Animale* pAnim = new Cane("Pastore tedesco");
    pAnim->stampaDati();
    Animale& rAnim = delfino;
    rAnim.stampaDati();
   ...
}//main
```

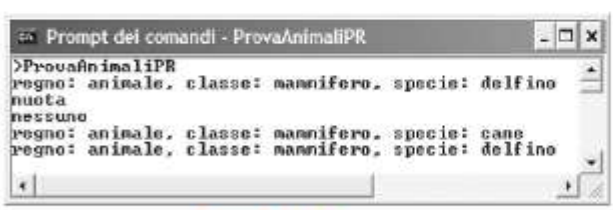


Figura 11.10 *Esecuzione dell'esempio*

Esempio 11.5

Anche i metodi *stampaDati()* della gerarchia degli animali possono essere più correttamente sostituiti con i sovraccarichi dell'operatore <<.

La classe ***Animale*** è una classe astratta pertanto non è possibile creare oggetti ma solo puntatori per accedere al sottooggetto contenuto negli oggetti delle classi derivate. La dichiarazione diventa:

```
friend ostream& operator<<(ostream&, const Animale*);
```

e l'implementazione diventa

```
ostream& operator<<(ostream& os, const Animale* an){
    os << "regno: " << an->getRegno();
    os << ", classe: " + an->getClasse();
    os << ", specie: " + an->getSpecie() + "\n";
    return os;
}//operator<<
```

Diventa anche necessario dichiarare e implementare dei metodi *get()* per accedere agli attributi definiti privati.

Nella classe ***Cane*** la dichiarazione è

```
friend ostream& operator<<(ostream&, Cane&);
```

e l'implementazione è

```
ostream& operator<<(ostream& os, Cane& cane){
    Animale* pAni = &cane;
    os << pAni << "razza: " << cane.razza << "\n";
    return os;
}//operator<<
```

Si può quindi fare:

```
Cane cane = Cane("cocker");
cout << cane;
```

Figura 11.11 *Esecuzione dell'esempio*

C++ 11.3 Ereditarietà multipla

In C++ è consentita l'**ereditarietà multipla** ovvero la possibilità di derivare una classe da due o più classi.

```
class NomeClasse : public Superclasse1, public Superclasse2, ...,
public SuperclasseN {
    // attributi
    // metodi
};
```

Nota Invece del modificatore *public* è possibile usare anche *private* e *protected*.

La classe derivata eredita le risorse pubbliche e protette di tutte le superclassi; un oggetto è visto come l'insieme dei propri attributi e dei sottooggetti provenienti dalle superclassi.

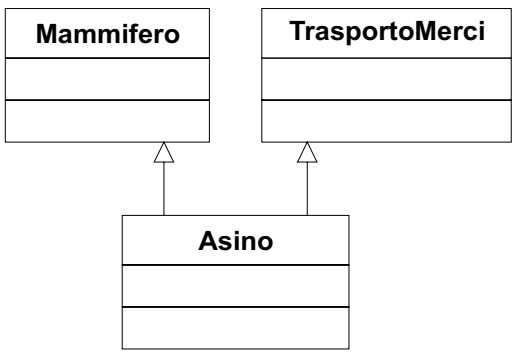
Se un metodo è presente in più superclassi, per risolvere l'ambiguità, nella chiamata è specificata mediante l'operatore :: anche la classe in un cui è contenuto:

```
Classe::metodo(parametri);
```

ESEMPIO 11.6

Definizione della classe *Asino* derivata dalle classi *Mammifero* (dell'esempio 11.4) e *TrasportoMerci*.

Figura 11.12 *Diagramma UML delle classi dell'esempio*




```
//TrasportoMerci.h
#include <string>
using namespace std;

class TrasportoMerci {
    private:
        unsigned int portata;
        string alimentazione;
    public:
        string getAlimentazione(void) const;
        unsigned int getPortata(void) const;
        TrasportoMerci(unsigned int =0, string ="");
};//TrasportoMerci
```

```
//TrasportoMerci.cpp
#include "TrasportoMerci.h"

TrasportoMerci::TrasportoMerci(unsigned int port, string alim){
    portata = port;
    alimentazione = alim;
}//TrasportoMerci

unsigned int TrasportoMerci::getPortata(void) const{
    return portata;
}//getPortata

string TrasportoMerci::getAlimentazione(void) const{
    return alimentazione;
}//getAlimentazione
```

```
//Asino.h
#include "Mammifero.h"
#include "TrasportoMerci.h"

class Asino : public Mammifero, public TrasportoMerci {
    private:
        string razza;
    public:
        virtual string verso(void) const;
        void stampaDati(void) const;
        Asino(string ="---", unsigned int =0);
};//Asino
```

```
//Asino.cpp
#include <iostream>
#include "Asino.h"
using namespace std;

Asino::Asino(string r, unsigned int port) : Mammifero("asino"),
TrasportoMerci(port,"vegetale"){
    razza = r;
}//Asino

string Asino::verso(void) const{
    return "raglia";
}//verso

void Asino::stampaDati(void) const{
    Animale::stampaDati();
    cout << "razza: " << razza << endl;
}//stampaDati
```

```
//ProvaAsino.cpp
#include <iostream>
#include "Asino.h"
using namespace std;

int main(void){
    Asino as = Asino("di Pantelleria", 100);
    as.stampaDati();
    cout << as.siMuove() << endl;
    cout << as.verso() << endl;
    cout << "mangia " << as.getAlimentazione() << endl;
    cout << "trasporta " << as.getPortata() << " kg" << endl;
    getchar();
    return 0;
}//main
```

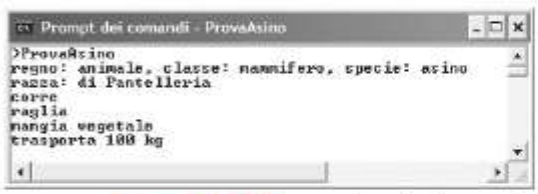


Figura 11.13 *Esecuzione dell'esempio*

Derivazione virtuale

La **derivazione a diamante** è un caso particolare di derivazione: una classe è derivata da due classi che a loro volta sono derivate dalla stessa classe.

Tutte le sottoclassi ereditano le risorse pubbliche e protette della loro superclasse creando ambiguità e spreco di memoria.

> La classe *Quattro* della figura 11.14 eredita dalle classi *Due* e *Tre*, ritrovandosi con due sottooggetti della classe *Uno*; il richiamo dei metodi della superclasse *Uno* risulta ambiguo.

Nei casi di derivazione a diamante si usa la **derivazione virtuale**.

Nella derivazione virtuale non viene creato il sottoggetto in ogni singola sottoclasse, ma viene creato un solo sottooggetto e in ogni sottoclasse viene creato un puntatore ad esso.

> Nella derivazione virtuale l'estensione della classe *Uno* diventa virtuale.

Per ottenere la derivazione virtuale di una classe bisogna specificare la parola riservata ***virtual*** nella dichiarazione della sottoclasse. La classe prende il nome di **classe base virtuale**.

Nel caso della derivazione a diamante della figura 11.14 la derivazione virtuale della classe *Uno* avviene secondo lo schema

```
class Uno {
    // attributi
    // metodi
};
class Due : virtual public Uno{
    // attributi
    // metodi
};
class Tre: virtual public Uno{
    // attributi
    // metodi
};
class Quattro: public Due, public Tre{
    // attributi
    // metodi
};
```

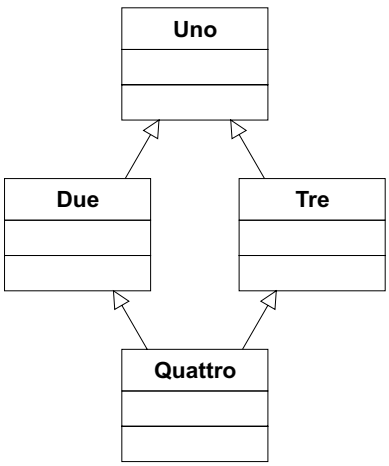


Figura 11.14 *Diagramma UML di una derivazione a diamante*

Nota

Nel diagramma UML a volte la derivazione virtuale si può trovare rappresentata con una freccia punteggiata.

ESEMPIO 11.7

Derivazione virtuale di una derivazione a diamante. Uso dei puntatori per accedere agli oggetti delle superclassi.
La classe *Forma2D* è la classe base virtuale astratta.

```
//Forma2D.h
class Forma2D {
    private:
        double lato;
    public:
        virtual double capienza(void) const =0;
        double fondo() const;
        Forma2D(double =0);
};//Forma2D
```

```
//Forma2D.cpp
#include "Forma2D.h"

Forma2D::Forma2D(double lato){
    this->lato = lato;
}//Forma2D

double Forma2D::fondo(void) const{
    return lato*lato;
}//fondo
```

```
//Forma3DFormaPunta.h
#include "Forma2D.h"
```

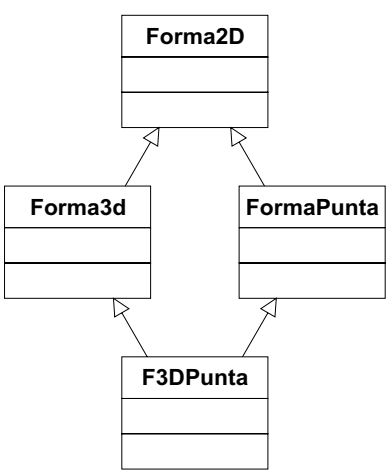


Figura 11.15 *Diagramma UML delle classi dell'esempio*

```
class Forma3D : virtual public Forma2D {
    private:
        double altezza;
    public:
        virtual double capienza(void) const;
        Forma3D(double =0, double =0);
};//Forma3D

class FormaPunta : virtual public Forma2D {
    private:
        double altezza;
    public:
        virtual double capienza(void) const;
        FormaPunta(double =0, double =0);
};//FormaPunta
- - - - - - - - - - - - - - - - - -
//Forma3DFormaPunta.cpp
#include "Forma3DFormaPunta.h"

Forma3D::Forma3D(double a, double b) : Forma2D(a){
    altezza = b;
}//Forma3D

double Forma3D::capienza(void) const{
    return Forma2D::fondo()*altezza;
}//capienza

FormaPunta::FormaPunta(double a, double b) : Forma2D(a){
    altezza = b;
}//FormaPunta

double FormaPunta::capienza(void) const{
    return Forma2D::fondo()*altezza/3;
}//capienza
- - - - - - - - - -
//F3DPunta.h
#include "Forma3DFormaPunta.h"

class F3DPunta : public Forma3D, public FormaPunta {
    public:
        virtual double capienza(void) const;
        F3DPunta(double, double, double);
};//F3DPunta
- - - - - - - - - - -
//F3DPunta.cpp
#include "F3DPunta.h"

F3DPunta::F3DPunta(double a, double b, double c) : Forma2D(a),
Forma3D(a, b), FormaPunta(a, c){}//F3DPunta

double F3DPunta::capienza(void) const{
    return Forma3D::capienza() + FormaPunta::capienza();
}//capienza
```

```
//ProvaF3DPunta.cpp
#include <iostream>
#include "F3DPunta.h"
using namespace std;

int main(void){
    Forma3D f3d(20, 15);
    FormaPunta fp(20, 15);
    F3DPunta f3dpu(20, 15, 15);
    Forma2D* pF2d = &f3d;
    cout << "capienza Forma3D: " << pF2d->capienza() << endl;
    pF2d = &fp;
    cout << "capienza FormaPunta: " << pF2d->capienza() << endl;
    pF2d = &f3dpu;
    cout << "capienza F3DPunta: " << pF2d->capienza() << endl;
    getchar();
    return 0;
}//main
```

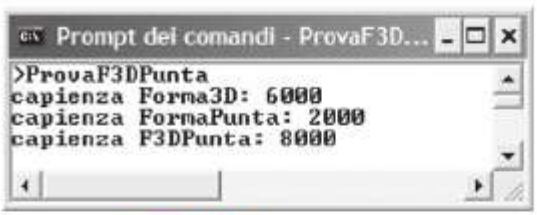


Figura 11.16 *Esecuzione dell'esempio*

VERIFICA

QUESTIONARIO

1. Che cos'è l'ereditarietà?
2. Qual è lo stato ereditato da una sottoclasse e come si può modificare?
3. Qual è il comportamento ereditato da una sottoclasse e come si può modificare?
4. Qual è lo scopo dell'uso dell'operatore :: nel caso di classi derivate?
5. Che cos'è una gerarchia di classi?
6. Che cos'è una classe astratta?
7. A cosa servono le classi astratte?
8. In che cosa consiste l'ereditarietà multipla?
9. Che cos'è la derivazione virtuale?
10. Che cos'è una derivazione a diamante?
11. Che cos'è il polimorfismo?
12. Quali sono i due aspetti del polimorfismo?
13. Che cosa si intende per legame dinamico o dynamic binding?
14. Come viene rappresentata l'ereditarietà in un diagramma UML?
15. Come deve essere l'invariante di classe di una sottoclasse rispetto a quella della superclasse?
16. Come deve essere lo spazio degli stati di una sottoclasse rispetto a quella della superclasse?
17. Che cos'è il principio del comportamento chiuso?
18. Che cos'è il principio di conformità di tipo?
19. Qual è la struttura della dichiarazione di una sottoclasse?
20. Qual è la differenza tra i modificatori di accesso *protected* e *private* per attributi e metodi?
21. Qual è la differenza tra metodi virtuali e metodi virtuali puri?
22. In che cosa consiste l'overriding di un metodo della superclasse nella sottoclasse?
23. In che cosa consiste l'overloading di un metodo della superclasse nella sottoclasse?

24. Che cosa devono fare i costruttori di una sottoclasse?
25. Qual è il legame tra il polimorfismo e i puntatori?
26. A che cosa servono metodi astratti e classi astratte?
27. Qual è la differenza tra classi astratte e classi virtuali astratte?
28. Quali problemi risolve la derivazione virtuale?

TEST

1 ..... *Indicare se le seguenti affermazioni sull'ereditarietà sono vere o false.*

V	F	
☐	☐	1) L'ereditarietà permette di definire una classe a partire da una classe già esistente.
☐	☐	2) Nell'ereditarietà le superclassi sono più specializzate delle sottoclassi.
☐	☐	3) In una sottoclasse si può ridefinire un metodo della superclasse con la stessa segnatura.
☐	☐	4) In una sottoclasse si può definire un metodo della superclasse con segnatura diversa.
☐	☐	5) L'ereditarietà singola da luogo a diagrammi UML ad albero.
☐	☐	6) Una classe può richiamare un metodo di una sua sottoclasse.
☐	☐	7) Una classe può richiamare un metodo di una sua superclasse.
☐	☐	8) Non è più possibile richiamare un metodo della superclasse che è stato ridefinito nella sottoclasse.
☐	☐	9) Una classe astratta contiene solo metodi virtuali puri.
☐	☐	10) Non è possibile creare un oggetto di una classe astratta.
☐	☐	11) L'ereditarietà multipla permette di ereditare da più classi.
☐	☐	12) In C++ l'ereditarietà multipla è consentita.
☐	☐	13) La derivazione virtuale risolve problemi di ambiguità.
☐	☐	14) Il polimorfismo permette di scegliere il metodo da richiamare in base al tipo dell'oggetto su cui viene richiamato.
☐	☐	15) L'ereditarietà singola può dar luogo a diagrammi UML ad albero.
☐	☐	16) Una variabile del tipo di una classe può contenere oggetti di sottoclassi.
☐	☐	17) Si può assegnare a una variabile del tipo di una sottoclasse un oggetto di una superclasse.
☐	☐	18) Il polimorfismo è permesso dal legame dinamico, cioè dalla scelta del metodo da eseguire durante l'esecuzione invece che durante la compilazione.

2 ..... *Indicare se le seguenti affermazioni sull'ereditarietà in C++ sono vere o false.*

V	F	
☐	☐	1) Se una sottoclasse ha il modificatore di accesso *private* non accede a nessuna risorsa della superclasse.
☐	☐	2) Se una sottoclasse ha il modificatore di accesso *public* accede a tutte le risorse della superclasse.

V	F		
☐	☐	3)	I metodi privati di una classe non possono essere utilizzati nelle sottoclassi.
☐	☐	4)	Il modificatore *protected* non può essere applicato a un metodo ridefinito.
☐	☐	5)	Quando si ridefinisce un metodo il nome dei parametri non ha importanza.
☐	☐	6)	In una sottoclasse non si può definire un metodo con lo stesso nome di uno della superclasse e segnatura diversa.
☐	☐	7)	Il costruttore della superclasse è richiamato con la notazione lista di inizializzazione.
☐	☐	8)	La chiamata a un costruttore della superclasse è sempre esplicitamente o implicitamente la prima istruzione del costruttore della sottoclasse.
☐	☐	9)	Le classi con metodi statici non si possono derivare.
☐	☐	10)	Quando si richiama un metodo usando un puntatore che punta a un oggetto la scelta del metodo dipende dal tipo del puntatore.
☐	☐	11)	Il cast da una superclasse a una sottoclasse è sempre possibile.
☐	☐	12)	Se una classe è astratta non si può istanziare.
☐	☐	13)	I metodi astratti possono essere privati.
☐	☐	14)	Una classe può essere astratta anche se non ha metodi astratti.
☐	☐	15)	Una classe che eredita un metodo virtuale puro e non lo implementa è astratta.
☐	☐	16)	Il sovraccarico di un operatore non può avvenire definendo un metodo della classe.
☐	☐	17)	L'operatore :: opportunamente usato in una sottoclasse permette di accedere alle variabili e ai metodi della superclasse.
☐	☐	18)	Un puntatore può puntare in momenti diversi a oggetti di classi diverse.

3 ..... Data la gerarchia

```
Animale
    Mammifero
        Cane
        Delfino
```

Quali assegnazioni sono permesse?

	da	a
☐	Animale	Mammifero
☐	Mammifero	Animale
☐	Cane	Animale

	da	a
☐	Animale	Cane
☐	Cane	Delfino
☐	Delfino	Cane

Esercizi

1 ..... Descrivere una possibile gerarchia di classi per:

- figure geometriche,
- persone della scuola (studenti, insegnanti ecc.),
- mezzi di trasporto.

2 Date le classi *NumeroNaturale* con 0 <= n < 10.000 e *NumeroIntero* con -10.000 < n < 10.000 una delle due classi può estendere l'altra?
Se sì quale? Se no perché?

3 Aggiungere alla classe *Studente* dell'esempio 11.1 un array con i voti della pagella di fine anno.

4 Se la classe *Animale* dell'esempio 11.4 avesse un metodo

```
void Animale::azioni(void) {
    cout << this->siMuove();
    cout << this->verso();
}
```

con le istruzioni

```
Cane cane  = Cane();
Animale animale = cane;
animale.azioni();
```

cosa succederebbe? Quali metodi *siMuove()* e *verso()* verrebbero richiamati?

PROPOSTE DI LAVORO

1 Estendere la classe *Persona* dell'esempio 11.1 in modo da ottenere la classe *PersonaStatoCivile* aggiungendo una variabile *sesso* che può contenere i valori *F* e *M* e una variabile *stato* che può contenere uno dei valori *celibe*, *nubile*, *coniugato*, *vedovo*, *divorziato* (gestiti tramite costanti della classe) e aggiungere i metodi per gestire questi dati.

2 Derivare dalla classe *Persona* dell'esempio 11.1 una classe *Insegnante* con attributi:
- scuola,
- materia,
- array di nomi delle classi in cui insegna.

Sovraccaricare l'operatore <<.

3 Creare la classe *QuadratoColorato* che estende la classe *Quadrato* (con attributo lato).

4 Creare la classe *Rettangolo* (con attributi base e altezza) e definire la classe *Quadrato* come sottoclasse di *Rettangolo*.
Definire anche la classe *Parallelepipedo* come sottoclasse di *Rettangolo*.

5 Creare la gerarchia di classi:
- *Punto*,
- *Pixel* (in più il colore),
- *PuntoIn3D* (in più la coordinata *z*).

6 Definire una classe astratta *Operazione* e le classi derivate *Addizione*, *Sottrazione* ecc. in modo che in un metodo si possa eseguire una operazione generica passandogli come parametri un oggetto di tipo *Operazione* e due operandi.

7 Realizzare una gerarchia di classi di triangoli di vario tipo (per esempio scaleni, isosceli, equilateri).

8 Realizzare una gerarchia di classi relative alle figure geometriche.

9 Definire una classe *Data* (anno, mese, giorno) con i sovraccarichi degli operatori ==, <<, ++ (per incrementare la data di un giorno) e -- (per decrementare la data) e opportuni metodi.
Estendere la classe *Data* aggiungendo l'orario in ore e minuti.

ESEMPI DI PROGETTAZIONE E IMPLEMENTAZIONE

12

PREREQUISITI

- Saper progettare classi e applicazioni a oggetti con relazioni di ereditarietà
- Saper rappresentare classi e relazioni di ereditarietà tra le classi con diagrammi UML

OBIETTIVI

CONOSCENZE

- Relazioni di associazione e dipendenza
- Diagrammi UML per la rappresentazione delle relazioni tra le classi
- Diagrammi UML di interazione tra oggetti
- Concetto di accoppiamento tra le classi
- Array di oggetti
- Design pattern (Builder, Composite e Iterator)

ABILITÀ/CAPACITÀ

- Individuare le relazioni tra le classi di una applicazione
- Progettare un'applicazione a oggetti mettendo in relazione tra di loro le classi
- Rappresentare le relazioni tra le classi con un diagramma UML
- Individuare e valutare diversi schemi di progettazione per realizzare un'applicazione a oggetti e implementarli in C++
- Utilizzare alcuni design pattern

12.1 RELAZIONI TRA LE CLASSI

Le **relazioni** possibili tra le classi sono:

- l'**ereditarietà:**

è una relazione tra una classe più generale e una più specializzata; viene indicata come relazione **IsA** (è un); è la relazione più comune e caratteristica; quando si deve progettare una classe, se è possibile, bisogna partire da classi già esistenti; ogni classe deve essere progettata in modo da poter essere estesa, se necessario.

> La classe *Studente* può estendere la classe *Persona*: uno studente è una persona.

- l'**associazione:**

rappresenta un legame tra istanze di classi; si può pensare come una classe i cui attributi sono oggetti di altre classi; nel caso più semplice una classe ha come attributo un oggetto di un'altra classe (cioè ha una variabile istanza del tipo di un'altra classe); viene indicata come relazione **HasA** (ha un).

> Date le classi *Persona* e *Provincia* si potrebbe definire la classe associazione *Residenza* con un attributo di tipo *Persona* e uno di tipo *Provincia*.
>
> Date le classi *Persona* e *Cane* si potrebbe definire la classe associazione *ProprietarioCane* con un attributo di tipo *Persona* e uno di tipo array di *Cane*.

Sono casi particolari di associazioni le associazioni di composizione e aggregazione.

La **composizione** indica le parti da cui è composto un oggetto; i componenti possono essere eterogenei e l'oggetto non esiste senza i suoi componenti.

> Data la classe *Punto* (con attributi le coordinate *x*, *y* di un punto):
> - la classe *Cerchio* ha un punto come centro,
> - la classe *Triangolo* ha tre punti come vertici del triangolo.

> La composizione spesso è legata alla propagazione dei messaggi (per esempio un metodo che sposta un oggetto composto richiama i metodi per spostare le sue parti). **Nota**

> Un oggetto tagliato in pezzi può essere considerato una composizione di porzioni di oggetto. **Nota**

L'**aggregazione** è un insieme di parti omogenee (e potrebbe esistere anche senza le sue parti).

> Una *Classe* (nel senso scolastico) è una aggregazione di oggetti *Studente*.

- la **dipendenza:**

una classe dipende da un'altra se uno dei suoi metodi la usa in qualche modo:

- un metodo riceve un oggetto della classe dipendente o come parametro o in conseguenza della chiamata di un altro metodo;
- un metodo crea un nuovo oggetto della classe dipendente;
- un metodo usa un metodo statico o una variabile statica della classe dipendente.

12.2 DIAGRAMMI UML DELLE ASSOCIAZIONI

Un'**associazione** è rappresentata da una linea di congiunzione tra due classi sulla quale appare il nome dell'associazione.

Figura 12.1 *Rappresentazione di una associazione*

Il ruolo di una classe può apparire accanto ad essa.

All'estremità di ogni linea appare la **molteplicità** dell'associazione:

0..*	qualunque numero, può essere abbreviata in *;
1..1	esattamente 1, può essere abbreviata in 1;
0..1	0 o 1;
1..*	1 o più di 1;
n	esattamente n.

L'associazione, invece che semplicemente dal nome, può essere rappresentata da una classe collegata da una linea tratteggiata alla linea di associazione. In questo modo si possono descrivere anche gli attributi e le operazioni della classe associazione.

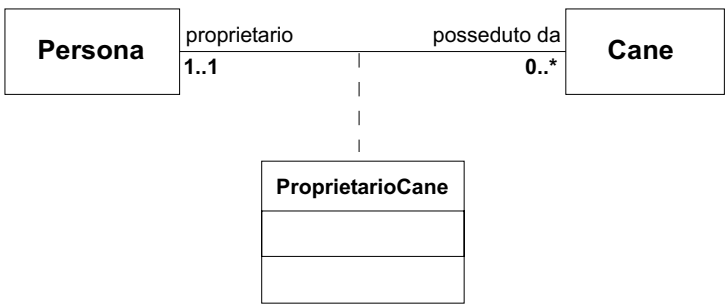


Figura 12.2 *Rappresentazione di una associazione come classe*

Nota

L'associazione è sempre una classe, anche se viene indicato soltanto il nome sulla linea di congiunzione tra le classi.

Se l'associazione coinvolge tre o più classi si usa un rombo per collegare le linee di associazione. La molteplicità sarà sempre 0..* altrimenti si potrebbe scomporre in due o tre associazioni binarie.

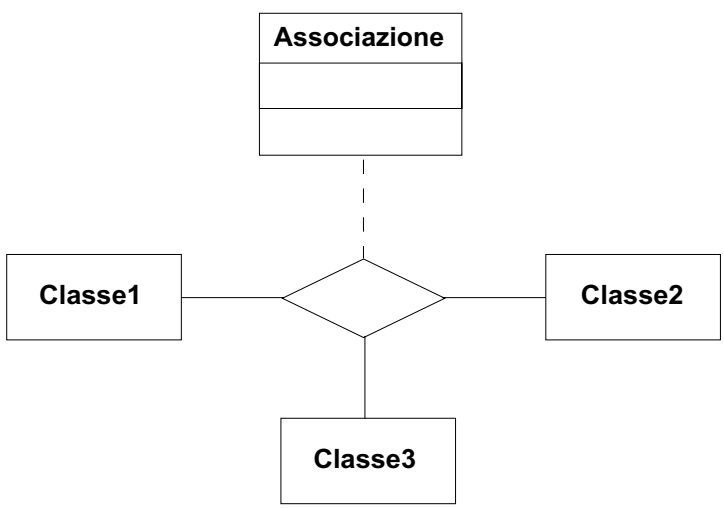


Figura 12.3 *Notazione UML per una associazione che coinvolge tre classi*

Nelle associazioni si possono aggiungere i simboli di **navigabilità** che indicano che l'implementazione permette di passare rapidamente da un oggetto all'altro (si dice anche di attraversare). I simboli di navigabilità sono rappresentati da punte di frecce nel verso della navigabilità sulla linea dell'associazione.

Per ottenere la navigabilità in un senso il primo oggetto deve contenere un riferimento al secondo.
Per ottenere la navigabilità nelle due direzioni gli oggetti devono avere riferimenti reciproci.

Le associazioni di composizione e aggregazione hanno una notazione particolare.
Sulle linee di composizione e aggregazione di solito non appare alcun nome.

La **composizione** è rappresentata da una linea di associazione con un rombo nero dal lato dell'oggetto composto; vicino al componente si può indicare il ruolo e la molteplicità. L'oggetto composto di solito ha i riferimenti ai componenti (nel diagramma si possono aggiungere le frecce di navigabilità); se necessario si può prevedere anche un riferimento dai componenti all'oggetto composto.

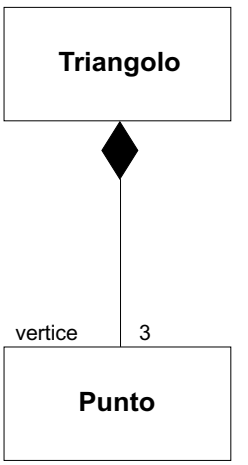


Figura 12.4 *Composizione*

L'**aggregazione** è rappresentata da una linea di associazione con un rombo vuoto dal lato dell'oggetto aggregato; vicino al componente si può indicare il ruolo; la molteplicità deve essere indicata da entrambi i lati.

L'aggregazione può avere delle **proprietà** tra parentesi graffe:

- **ordered** indica che gli elementi sono ordinati;

> Per esempio paragrafi ordinati in una sequenza di testo.

- **bag** (multiinsieme) consente a un elemento di apparire più volte;
- **set** vieta i duplicati; è il valore di default.

L'aggregazione si può implementare con un riferimento all'insieme dei costituenti.

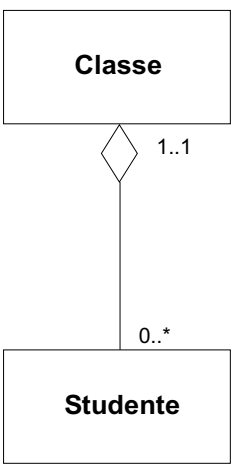


Figura 12.5 *Aggregazione*

Dipendenze

La dipendenza è rappresentata da una linea tratteggiata con una freccia con la punta aperta che punta alla classe dipendente.

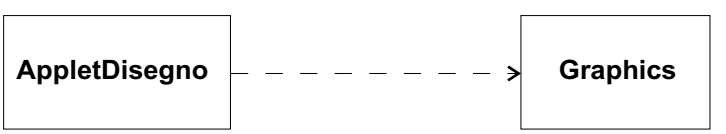


Figura 12.6 *Dipendenza*

12.3 DIAGRAMMI DI INTERAZIONE TRA OGGETTI

Durante l'esecuzione è fondamentale la chiamata di metodi (cioè l'invio di messaggi agli oggetti).
I **diagrammi di interazione** tra oggetti descrivono le chiamate di metodi (cioè lo scambio di messaggi) in un caso specifico.
Ci sono due forme di diagrammi di interazione tra oggetti: il **diagramma di collaborazione** e il **diagramma di sequenza**; i due diagrammi possono essere convertiti uno nell'altro.

Nel **diagramma di collaborazione** gli oggetti sono rappresentati da rettangoli con nomeOggetto:Classe sottolineato in modo da indicare che sono istanze; si può specificare

anche il nome di un metodo all'interno del quale viene chiamato il metodo da descrivere (cioè all'interno del quale viene inviato il messaggio). Gli oggetti sono identificati con i nomi che gli altri usano per farvi riferimento.

Nota

Un oggetto non ha un nome proprio: si deve usare un riferimento.

Le chiamate di metodi sono indicate da una freccia dal mittente al destinatario; nella chiamata del metodo vanno indicati i parametri attuali (non quelli formali).

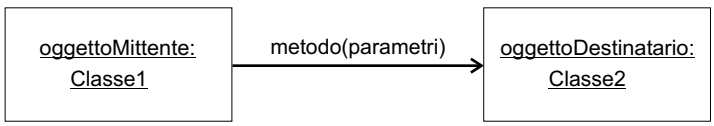


Figura 12.7 *Diagramma di collaborazione*

Se la chiamata è polimorfica la classe associata al destinatario deve essere la superclasse di tutte le classi, più in basso possibile nella gerarchia di ereditarietà; si può esplicitare il polimorfismo indicando la classe tra parentesi.

Si possono considerare chiamate iterate, per esempio perché i messaggi sono mandati a tutti gli elementi di una aggregazione; in questo caso il metodo ha come prefisso un asterisco che indica molteplicità; nel destinatario si precisa :Classe ma non il nome dell'oggetto; il rettangolo dell'oggetto viene duplicato per rappresentare la molteplicità.

Se si richiama un metodo dello stesso oggetto (invio di un messaggio a sé stessi) si può chiamare *self* il destinatario o anche avere una linea chiusa che torna al mittente.

I messaggi asincroni sono rappresentati da una mezza punta nella freccia. Per i messaggi asincroni in concorrenza in broadcast la notazione è come per i messaggi iterati con lo stereotipo <<broadcast>> e come destinatario :(Object).

Il **diagramma di sequenza** mostra la sequenza temporale delle operazioni. Si può vedere non solo a chi appartengono i metodi richiamati ma anche quando vengono richiamati (cioè non solo a chi sono diretti i messaggi ma anche quando vengono inviati).

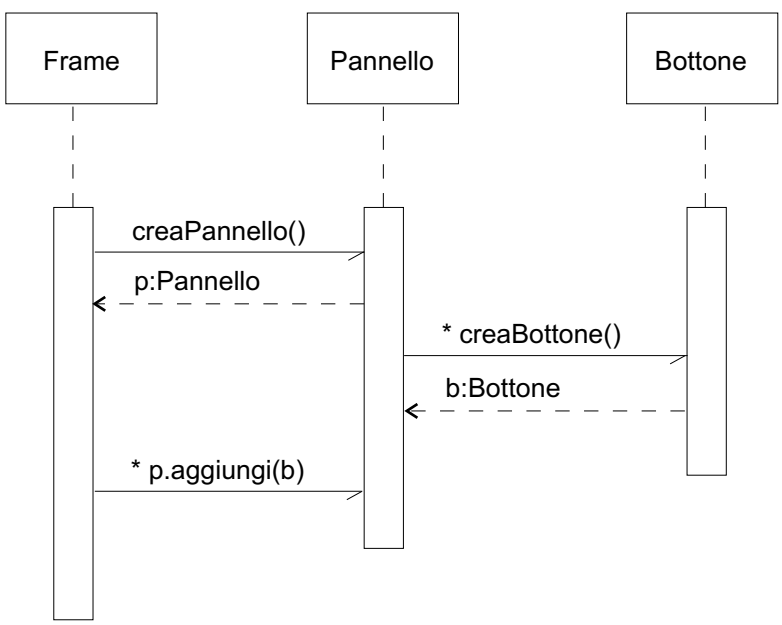


Figura 12.8 *Diagramma di sequenza*

Il tempo cresce verso il basso lungo un asse verticale mentre in orizzontale c'è un elenco di oggetti di cui verranno richiamati i metodi (cioè che riceveranno messaggi); eventualmente può bastare il nome delle classi.

Il diagramma mostra i metodi richiamati; ogni metodo richiamato è rappresentato da una freccia orizzontale; le frecce sono spaziate in modo da indicare la durata approssimativa del metodo. Un limite di questo diagramma è che non si vede se i metodi che dipendono da una condizione vengono richiamati o no; si può aggiungere a sinistra la pseudocodifica della condizione.

12.4 ACCOPPIAMENTO

Un problema fondamentale nella progettazione delle classi è l'accoppiamento (chiamato da alcuni autori anche conascenza).

> Si dice che c'è **accoppiamento** tra due classi se una classe è influenzata da modifiche che possono essere apportate all'altra.

L'accoppiamento può risultare per esempio da relazioni di dipendenza.
Bisogna analizzare le dipendenze tra le classi in modo da ridurre al minimo l'accoppiamento.

12.5 ESEMPI DI ASSOCIAZIONI

Uso di array di oggetti

Le relazioni di associazione e in particolare di composizione e aggregazione in genere sono realizzate da classi in cui uno o più **attributi** sono **array di oggetti**.

Esempio 12.1

Classe *Poligono* che ha come attributo un array di oggetti *Punto*.

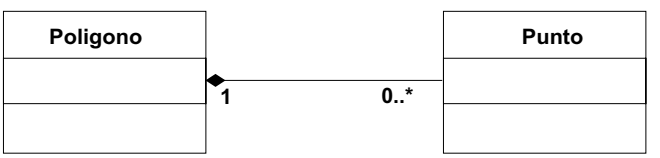


Figura12.9 *Diagramma UML delle classi dell'esempio*

```
//Punto.h
#include <iostream>
using namespace std;

class Punto {
    private:
        int x, y;
    public:
        int getX(void) const;
        int getY(void) const;
        double distanza(const Punto&) const;
        friend ostream& operator<<(ostream&, const Punto&);
        Punto(int =0, int =0);
};//Punto
```

```
//Punto.cpp
#include "Punto.h"
#include <cmath>

Punto::Punto(int x, int y){
    this->x = x;
    this->y = y;
}//Punto

int Punto::getX(void) const{
```

```
    return x;
}//getX

int Punto::getY(void) const{
    return y;
}//getY

double Punto::distanza(const Punto& p) const{
    return sqrt((x - p.x)*(x - p.x) + (y - p.y)*(y - p.y));
}//distanza

ostream& operator<<(ostream& os, const Punto& p){
    os << "(" << p.x << "," << p.y << ")";
    return os;
}//operator<<
```

- - - - - - - - -

```
//Poligono.h
#include "Punto.h"

class Poligono {
    private:
        Punto vertici[9];
        int numVertici;
    public:
        void aggiungiVertice(const Punto&);
        void stampaVertici(void) const;
        double getPerimetro(void) const;
        Poligono(void);
};//Poligono
```

- - - - - - - - - -

```
//Poligono.cpp
#include <iostream>
#include "Poligono.h"
using namespace std;

Poligono::Poligono(void){
    numVertici = 0;
}//Poligono

void Poligono::aggiungiVertice(const Punto& p){
    vertici[numVertici++] = p;
}//aggiungiVertice

void Poligono::stampaVertici(void) const{
    for(int i=0; i<numVertici; i++)
        cout << vertici[i] << endl;
}//stampaVertici

double Poligono::getPerimetro(void) const{
    double perimetro = 0;
    for(int i=0; i<numVertici-1; i++)
        perimetro += vertici[i].distanza(vertici[i+1]);
    perimetro += vertici[numVertici-1].distanza(vertici[0]);
    return perimetro;
}//getPerimetro
```

```
//ProvaPoligono.cpp
#include <iostream>
#include "Poligono.h"
using namespace std;

int main(void){
    Poligono poligono;
    for(int i=0; i<3; i++){
        Punto p = Punto(2*i, i);
        poligono.aggiungiVertice(p);
    }//for
    poligono.stampaVertici();
    cout << poligono.getPerimetro() << endl;
   ...
}//main
```

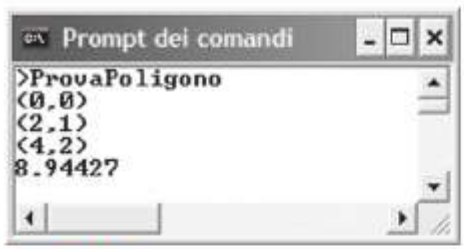


Figura 12.10 *Esecuzione dell'esempio*

ESEMPIO 12.2

Classe *Mesi* che ha come attributo un array di oggetti *Mese* con nome del mese e numero dei giorni del mese.

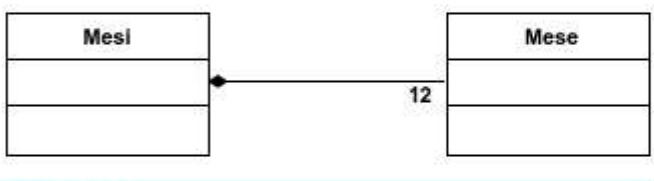


Figura 12.11 *Diagramma UML delle classi dell'esempio*

```
//Mese.h
#include <iostream>
#include <string>
using namespace std;

class Mese {
    private:
        string nome;
        int giorni;
    public:
        string getMese(void) const;
        int getGiorni(void) const;
        friend ostream& operator<<(ostream&, const Mese&);
        Mese(string ="", int =0);
};//Mese
```

```
//Mese.cpp
#include "Mese.h"

Mese::Mese(string n, int g){
    nome = n;
    giorni = g;
}//Mese

string Mese::getMese(void) const{
    return nome;
}//getMese

int Mese::getGiorni(void) const{
    return giorni;
}//getGiorni

ostream& operator<<(ostream& os, const Mese& m){
```

```
    os << m.nome << " " << m.giorni;
    return os;
}//operator<<
```

```
//Mesi.h
#include "Mese.h"
class Mesi {
    private:
        Mese mesi[12];
    public:
        string getMese(int) const;
        int getGiorni(int) const;
        Mese getDati(int) const;
        Mesi(void);
};//Mesi
```

```
//Mesi.cpp
#include "Mesi.h"
Mesi::Mesi(void){
    mesi[0] = Mese("gennaio", 31);
    mesi[1] = Mese("febbraio", 28);
    mesi[2] = Mese("marzo", 31);
    mesi[3] = Mese("aprile", 30);
    mesi[4] = Mese("maggio", 31);
    mesi[5] = Mese("giugno", 30);
    mesi[6] = Mese("luglio", 31);
    mesi[7] = Mese("agosto", 31);
    mesi[8] = Mese("settembre", 30);
    mesi[9] = Mese("ottobre", 31);
    mesi[10] = Mese("novembre", 30);
    mesi[11] = Mese("dicembre", 31);
}//Poligono
Mese Mesi::getDati(int i) const{
    return mesi[i-1];
}//getDati
string Mesi::getMese(int i) const{
    return mesi[i-1].getMese();
}//getMese
int Mesi::getGiorni(int i) const{
    return mesi[i-1].getGiorni();
}//getGiorni
```

```
//ProvaMesi.cpp
#include <iostream>
#include "Mesi.h"
using namespace std;
int main(void){
    unsigned short n;
    Mesi mesi;
```

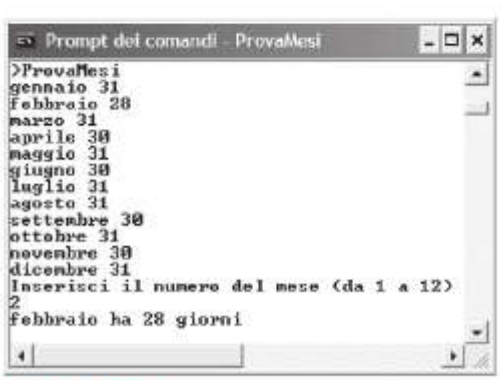


Figura 12.12 *Esecuzione dell'esempio*

```
    for(int i=1; i<=12; i++)
        cout << mesi.getDati(i) << endl;
    cout << "Inserisci il numero del mese (da 1 a 12)\n";
    cin >> n;
    cout << mesi.getMese(n) << " ha "
         << mesi.getGiorni(n) << " giorni\n";
    fflush(stdin);
    ...
}//main
```

ESEMPIO 12.3

Classe per la gestione della tavola pitagorica: il costruttore crea la tabella, il metodo *stampa()* stampa la tabella, il metodo *prodotto()* restituisce il prodotto di due numeri ricavandolo dalla tabella.

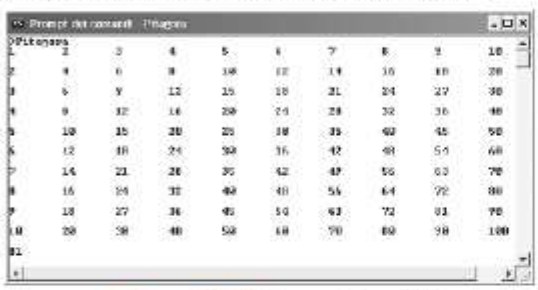


Figura 12.13 *Esecuzione dell'esempio*

```
#include <iostream>
#include <string>
using namespace std;

class Pitagora {
    private:
        static const int Max = 20;
        int tabella[Max][Max];
        int n;
    public:
        int prodotto(int, int) const;
        void stampa(void) const;
        Pitagora(int =0);
};//Pitagora

Pitagora::Pitagora(int n){
    this->n = n;
    for(int i=0; i<n; i++)
        for(int j=0; j<n; j++)
            tabella[i][j] = (i+1) * (j+1);
}//Pitagora

void Pitagora::stampa(void) const{
    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++)
            cout << tabella[i][j] << "\t";
        cout << endl;
    }//for
}//stampa

int Pitagora::prodotto(int i, int j) const{
    return tabella[i-1][j-1];
}//prodotto

int main(void){
    Pitagora p(10);
    p.stampa();
    cout << p.prodotto(9, 9) << endl;
   ...
}//main
```

Uso di vettori

Invece di un array spesso è più comodo utilizzare un oggetto della classe ***vector***.

A volte ci si riferisce a *vector* con il termine vettore. **Nota**

Gli oggetti della classe *vector*, rispetto agli array hanno notevoli **vantaggi**:

- le loro dimensioni possono variare in modo dinamico, in base alle necessità;
- permettono di eseguire facilmente le operazioni di inserimento e cancellazione di elementi (gli altri elementi vengono shiftati automaticamente);
- usando i template (modello di classe) permettono di specificare il tipo di oggetti che devono contenere.

I template e la classe *vector* sono spiegati nel capitolo 13. **Nota**

Si consiglia di usare:

- array se si devono memorizzare valori di tipo primitivo, se l'array è bidimensionale o se si deve ottenere la massima efficienza;
- vettori in tutti gli altri casi.

12.6 SCHEMI DI PROGETTAZIONE

Uno stesso problema può essere risolto utilizzando schemi di progettazione diversi.

Schemi di base

Per esempio considerando come problema la gestione dei prestiti dei libri di una biblioteca, ponendosi come obiettivo sapere quali libri ha in prestito una persona, si possono realizzare almeno quattro schemi di progettazione diversi.

Schema 1

classe *Libro*	con i dati di un libro;
classe *UtenteBiblioteca*	con i dati di una persona e con un insieme dei libri presi in prestito.

Questo schema è semplice ma la classe *UtenteBiblioteca*, che contiene i dati di una persona, non è facilmente riutilizzabile dove serve conoscere i dati di una persona ma non quali libri ha in prestito.

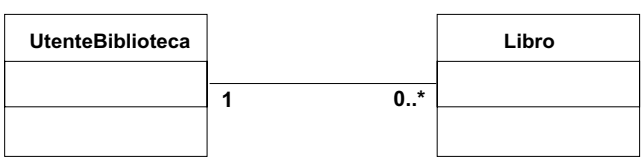


Figura 12.14 *Diagramma UML delle classi dello schema 1*

Schema 2

classe *Libro*	con i dati di un libro;
classe *Persona*	con i dati di una persona;
classe *Prestiti*	con una *Persona* e un insieme dei libri presi in prestito (associa una persona ai libri presi in prestito).

Figura 12.15 *Diagramma UML delle classi dello schema 2*

Questo schema è il più complicato dal punto di vista della realizzazione del codice.
Per trovare i libri che una persona ha in prestito infatti bisogna utilizzare un metodo statico della classe *Prestiti* che riceva come parametro un oggetto *Persona* e restituisca l'insieme dei libri presi a prestito.

Schema 3

classe *Libro*	con i dati di un libro;
classe *Persona*	con i dati di una persona;
classe *Prestiti*	con un insieme dei libri presi in prestito;
classe *UtenteBiblioteca*	sottoclasse, per ereditarietà multipla, di *Persona* e di *PrestitiBiblioteca*.

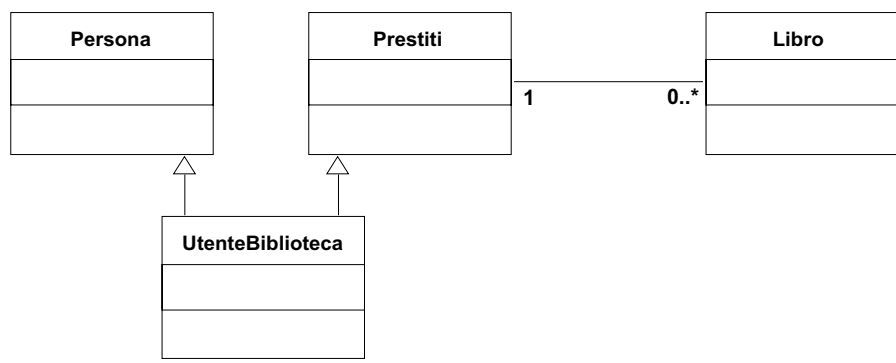


Figura 12.16 *Diagramma UML delle classi dello schema 3*

Questa soluzione è flessibile ma richiede il concetto di ereditarietà multipla.
Inoltre quando si crea una istanza di *UtenteBiblioteca* si inseriscono anche i dati della classe *Persona*; eliminando un *UtenteBiblioteca* dovrebbero restare i suoi dati come *Persona*; in realtà invece bisogna ricreare un'istanza di *Persona*; questo problema è noto come problema del partizionamento dinamico, cioè del passaggio di un oggetto da una classe all'altra.

Schema 4

classe *Libro*	con i dati di un libro;
classe *Persona*	con i dati di una persona;
classe *Prestiti*	con un insieme dei libri presi in prestito;
classe *UtenteBiblioteca*	con una *Persona* e un oggetto della classe *Prestiti*.

La classe *UtenteBiblioteca* in questo caso acquisisce le sue proprietà non attraverso l'ereditarietà ma attraverso il riferimento a due oggetti.
Per ogni *UtenteBiblioteca* vengono istanziati tre oggetti: uno della classe *Persona*, uno della classe *Prestiti* (con l'insieme dei libri in prestito) e uno della classe *UtenteBiblioteca* che ha i primi due oggetti come componenti.

Le operazioni *get()* sulla classe *UtenteBiblioteca* ottengono i dati richiamando i metodi *get()* corrispondenti sugli oggetti componenti (cioè inoltrando dei messaggi agli oggetti componenti).

Nota

Si può dire in generale che l'ereditarietà multipla può essere trasformata in una associazione di composizione con l'inoltro dei messaggi.

Questa soluzione risolve il problema del partizionamento dinamico: eliminando un *UtenteBiblioteca* restano i suoi dati come *Persona*.
Però la semplicità dell'ereditarietà è sostituita dalla complicazione dell'inoltro dei messaggi.

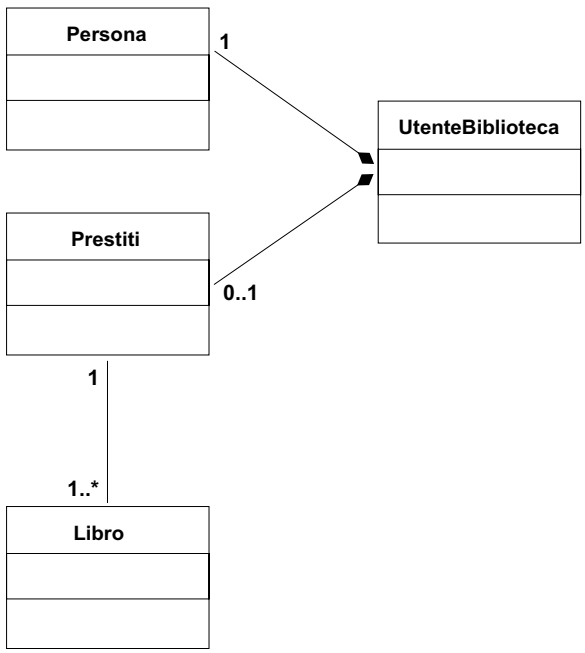


Figura 12.17 *Diagramma UML delle classi dello schema 4*

Generalizzazione

Uno stesso schema di progettazione può essere applicabile a molti casi diversi, ma analoghi dal punto di vista della soluzione.
Gli schemi visti in precedenza sono generalizzabili ad altri tipi di problemi completamente diversi.

Un problema analogo al prestito dei libri è quello relativo al possesso.

Si ottengono schemi analoghi considerando come obiettivo sapere quali cani possiede una persona.

Un esempio completo di implementazione in C++

C++ 12.1

Un altro problema analogo ai precedenti è quello di associare un insieme di attributi a un oggetto.

Definizione degli schemi

Considerando una classe *Attributo* con nome e valore di un attributo e un quadrato a cui si vogliono associare degli attributi come colore, spessore, materiale e così via si possono realizzare gli schemi:

- **Schema 1**

classe *Attributo*	con nome e valore di un attributo;
classe *QuadratoConAttributi*	con il lato del quadrato e con l'insieme degli attributi associati.

- **Schema 2**

classe *Attributo*	con nome e valore di un attributo;
classe *Quadrato*	con il lato del quadrato;
classe *QuadratoConAttributi*	con un *Quadrato* e un insieme degli attributi associati.

Schema 3

classe *Attributo*	con nome e valore di un attributo;
classe *Quadrato*	con il lato del quadrato;
classe *ListaAttributi*	con un insieme di attributi;
classe *QuadratoConAttributi*	sottoclasse, per ereditarietà multipla, di *Quadrato* e di *ListaAttributi*.

Schema 4

classe *Attributo*	con nome e valore di un attributo;
classe *Quadrato*	con il lato del quadrato;
classe *ListaAttributi*	con un insieme di attributi;
classe *QuadratoConAttributi*	con un *Quadrato* e un oggetto della classe *ListaAttributi*.

Implementazione

La classe *Attributo* con nome e valore di un attributo è comune a tutti gli schemi:

```
//Attributo.h
#include <iostream>
#include <string>
using namespace std;

class Attributo {
    private:
        string nome;
        string valore;
    public:
        string getNome(void) const;
        string getValore(void) const;
        void setNome(string);
        void setValore(string);
        friend ostream& operator<<(ostream&, const Attributo&);
        Attributo(string ="**", string ="**");
};//Attributo
- - - - - - - - - - - -
//Attributo.cpp
#include "Attributo.h"

Attributo::Attributo(string nome, string valore){
    this->nome = nome;
    this->valore = valore;
}//Attributo

string Attributo::getNome(void) const{
    return nome;
}//getNome

string Attributo::getValore(void) const{
    return valore;
}//getValore

void Attributo::setNome(string nome){
    this->nome = nome;
}//setNome
```

```
void Attributo::setValore(string valore){
    this->valore = valore;
}//setValore

ostream& operator<<(ostream& os, const Attributo& a){
    os << "Nome: " << a.nome << " Valore: " << a.valore;
    return os;
}//operator<<
```

Per provare gli schemi 1, 3 e 4 si può usare la stessa classe:

```
//Prova.cpp
#include <iostream>
#include "QuadratoConAttributi.h"
using namespace std;

int main(void){
    QuadratoConAttributi q(10);
    q.aggiungiAttributo("colore", "rosso");
    q.aggiungiAttributo("spessore", "15");
    q.elencaAttributi();
    getchar();
    return 0;
}//main
```

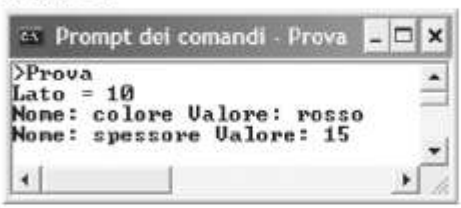


Figura 12.18 *Esecuzione dell'esempio*

SCHEMA 1

classe *Attributo* con nome e valore di un attributo;
classe *QuadratoConAttributi* con il lato del quadrato e con l'insieme degli attributi associati.

Una variabile di istanza di *QuadratoConAttributi* è un array di oggetti *Attributo*.

Figura 12.19 *Diagramma UML delle classi dello schema 1*

ESEMPIO 12.4

```
//QuadratoConAttributi.h
#include "Attributo.h"
class QuadratoConAttributi {
    private:
        int lato;
        int numAttributi;
        Attributo attributi[9];
    public:
        void aggiungiAttributo(string nome, string valore);
        void elencaAttributi(void) const;
        int getLato(void) const;
        int getNumAttributi(void) const;
        void setLato(int);
```

```
        QuadratoConAttributi(int);
        QuadratoConAttributi(void);
};//QuadratoConAttributi
- - - - - - - - - - - - - - - - - - -
//QuadratoConAttributi.cpp
#include "QuadratoConAttributi.h"

QuadratoConAttributi::QuadratoConAttributi(int lato){
    this->lato = lato;
    numAttributi = 0;
}//QuadratoConAttributi

QuadratoConAttributi::QuadratoConAttributi(void){
    lato = 0;
    numAttributi = 0;
}//QuadratoConAttributi

void QuadratoConAttributi::aggiungiAttributo(string nome, string
valore){
    attributi[numAttributi++] = Attributo(nome, valore);
}//aggiungiAttributo

void QuadratoConAttributi::setLato(int lato){
    this->lato = lato;
}//setLato

void QuadratoConAttributi::elencaAttributi(void) const{
    cout << "Lato = " << lato << endl;
    for(int i=0; i<numAttributi; i++)
        cout << attributi[i] << endl;
}//elencaAttributi

int QuadratoConAttributi::getLato(void) const{
    return lato;
}//getLato

int QuadratoConAttributi::getNumAttributi(void) const{
    return numAttributi;
}//getNumAttributi
```

SCHEMA 2

classe *Attributo*	con nome e valore di un attributo;
classe *Quadrato*	con il lato del quadrato;
classe *QuadratoConAttributi*	con un *Quadrato* e un insieme degli attributi associati.

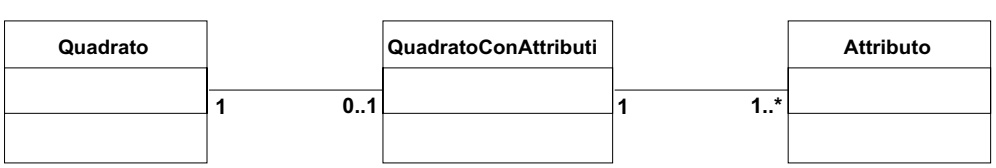


Figura 12.20 *Diagramma UML delle classi dello schema 2*

Per impostare o trovare gli attributi di un quadrato bisogna utilizzare un metodo statico di *QuadratoConAttributi* che riceva come parametro un puntatore a un oggetto *QuadratoConAttributi* e lavori sui suoi attributi.

ESEMPIO 12.5

```
//Quadrato.h
class Quadrato {
    private:
        int lato;
    public:
        int getLato(void) const;
        bool operator==(Quadrato) const;
        void setLato(int);
        Quadrato(int =0);
};//Quadrato
```

```
//Quadrato.cpp
#include "Quadrato.h"

Quadrato::Quadrato(int lato){
    this->lato = lato;
}//Quadrato

bool Quadrato::operator==(Quadrato q) const{
    return lato == q.lato;
}//operator==

void Quadrato::setLato(int lato){
    this->lato = lato;
}//setLato

int Quadrato::getLato(void) const{
    return lato;
}//getLato
```

```
//QuadratoConAttributi.h
#include "Attributo.h"
#include "Quadrato.h"
class QuadratoConAttributi {
    private:
        static QuadratoConAttributi* quadrati;
        static int numQuadrati;
        Quadrato quadrato;
        int numAttributi;
        Attributo attributi[9];
        static int cercaQuadrato(Quadrato);
    public:
        static void aggiungiAttributo(QuadratoConAttributi*,
string nome, string valore);
        static void elencaAttributi(QuadratoConAttributi*);
        static int getNumAttributi(QuadratoConAttributi*);
        QuadratoConAttributi(int =0);
};//QuadratoConAttributi
```

```
//QuadratoConAttributi.cpp
#include "QuadratoConAttributi.h"

int QuadratoConAttributi::numQuadrati = 0;
QuadratoConAttributi* QuadratoConAttributi::quadrati = new
QuadratoConAttributi[99];

QuadratoConAttributi::QuadratoConAttributi(int lato){
    QuadratoConAttributi* quad = quadrati+numQuadrati;
    quadrato = Quadrato(lato);
    numAttributi = 0;
    quad = this;
    quadrati = quad-numQuadrati++;
}//QuadratoConAttributi

void QuadratoConAttributi::aggiungiAttributo(QuadratoConAttributi* q,
string nome, string valore){
    int i = cercaQuadrato(q->quadrato);
    quadrati[i].attributi[quadrati[i].numAttributi] =
Attributo(nome, valore);
    quadrati[i].numAttributi++;
}//aggiungiAttributo

int QuadratoConAttributi::cercaQuadrato(Quadrato q){
    for(int i=0; i<numQuadrati; i++)
        if(quadrati[i].quadrato == q)
            return i;
    return -1;
}//cercaQuadrato

void QuadratoConAttributi::elencaAttributi(QuadratoConAttributi* q){
    int i = cercaQuadrato(q->quadrato);
    cout << "Lato = " << q->quadrato.getLato() << endl;
    for(int j=0; j<quadrati[i].numAttributi; j++)
        cout << quadrati[i].attributi[j] << endl;
}//elencaAttributi

int QuadratoConAttributi::getNumAttributi(QuadratoConAttributi* q){
    int i = cercaQuadrato(q->quadrato);
    return quadrati[i].numAttributi;
}//getNumAttributi
```

```
//Prova.cpp
#include <iostream>
#include "QuadratoConAttributi.h"
using namespace std;

int main(void){
    QuadratoConAttributi q(10);
    QuadratoConAttributi::aggiungiAttributo(&q, "colore", "rosso");
    QuadratoConAttributi::aggiungiAttributo(&q, "spessore", "15");
    QuadratoConAttributi::elencaAttributi(&q);
  ...
}//main
```

SCHEMA 3

classe *Attributo*	con nome e valore di un attributo;
classe *Quadrato*	con il lato del quadrato;
classe *ListaAttributi*	con un insieme di attributi;
classe *QuadratoConAttributi*	sottoclasse, per ereditarietà multipla, di *Quadrato* e di *ListaAttributi*.

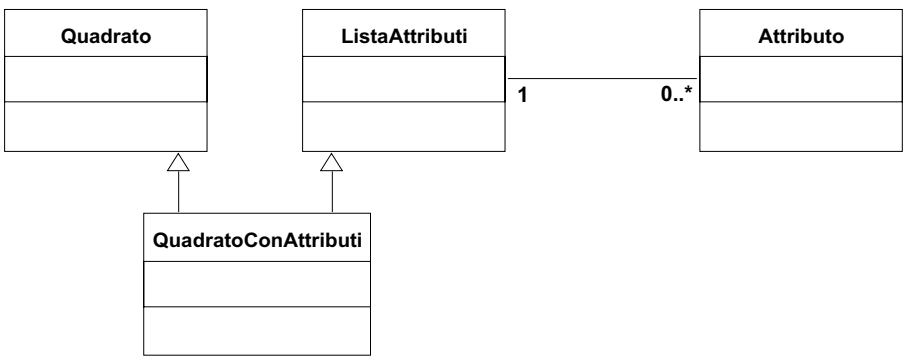


Figura 12.21 *Diagramma UML delle classi dello schema 3*

La classe *ListaAttributi* ha una variabile di istanza che è un array di oggetti *Attributo*. *QuadratoConAttributi* eredita da *ListaAttributi* la capacità di memorizzare attributi e da *Quadrato* la capacità di memorizzare quadrati.

ESEMPIO 12.6

```
//ListaAttributi.h
#include "Attributo.h"

class ListaAttributi {
    private:
        int numAttributi;
        Attributo attributi[9];
    public:
        void aggiungiAttributo(string nome, string valore);
        void elencaAttributi(void) const;
        int getNumAttributi(void) const;
        ListaAttributi(void);
};//ListaAttributi
- - - - - - - - - - - - - - - -
//ListaAttributi.cpp
#include "ListaAttributi.h"

ListaAttributi::ListaAttributi(void){
    numAttributi = 0;
}//ListaAttributi

void ListaAttributi::aggiungiAttributo(string nome, string valore){
    attributi[numAttributi++] = Attributo(nome, valore);
}//aggiungiAttributo

void ListaAttributi::elencaAttributi(void) const{
    for(int i=0; i<numAttributi; i++)
        cout << attributi[i] << endl;
}//elencaAttributi
```

```
int ListaAttributi::getNumAttributi(void) const{
    return numAttributi;
}//getNumAttributi
- - - - - - - - - - - - - - - - - -
//QuadratoConAttributi.h
#include "Quadrato.h"
#include "ListaAttributi.h"

class QuadratoConAttributi : public Quadrato, public ListaAttributi{
    public:
        void elencaAttributi(void) const;
        QuadratoConAttributi(int);
};//QuadratoConAttributi
- - - - - - - - - - - - - - - - - - - -
//QuadratoConAttributi.cpp
#include "QuadratoConAttributi.h"

QuadratoConAttributi::QuadratoConAttributi(int lato) :
Quadrato(lato), ListaAttributi(){
}//QuadratoConAttributi

void QuadratoConAttributi::elencaAttributi(void) const{
    cout << "Lato = " << Quadrato::getLato() << endl;
    ListaAttributi::elencaAttributi();
}//elencaAttributi
```

Schema 4

classe *Attributo*	con nome e valore di un attributo;
classe *Quadrato*	con il lato del quadrato;
classe *ListaAttributi*	con un insieme di attributi;
classe *QuadratoConAttributi*	con un *Quadrato* e un oggetto della classe *ListaAttributi*.

La classe *QuadratoConAttributi* ha una variabile di istanza che è un oggetto della classe *Quadrato* e una che è un oggetto della classe *ListaAttributi*; può usare questi oggetti per richiamare i metodi delle classi *Quadrato* e *ListaAttributi*; le chiamate ai metodi *aggiungiAttributo()* e *elencaAttributi()* vengono rinviate all'oggetto di tipo *ListaAttributi*.

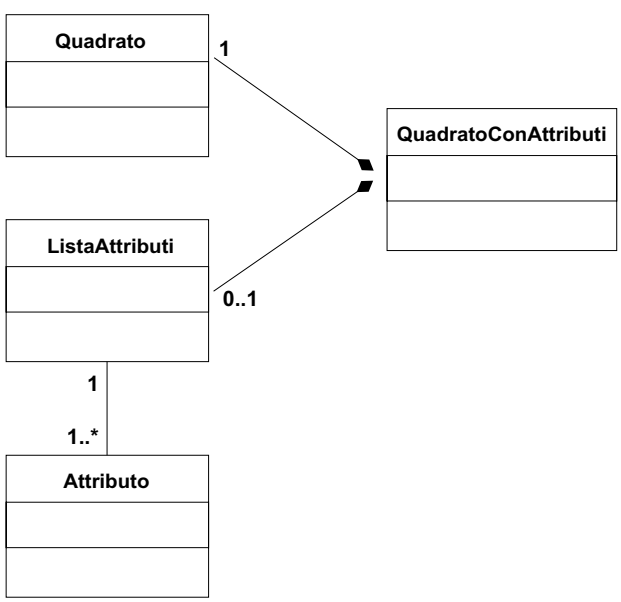


Figura 12.22
Diagramma UML delle classi dello schema 4

ESEMPIO 12.7

```
//QuadratoConAttributi.h
#include "Quadrato.h"
#include "ListaAttributi.h"

class QuadratoConAttributi {
    private:
        Quadrato q;
        ListaAttributi attributi;
    public:
        void elencaAttributi(void) const;
        void aggiungiAttributo(string nome, string valore);
        QuadratoConAttributi(int =0);
};//QuadratoConAttributi
```

```
//QuadratoConAttributi.cpp
#include "QuadratoConAttributi.h"

QuadratoConAttributi::QuadratoConAttributi(int lato){
    q = Quadrato(lato);
}//QuadratoConAttributi

void QuadratoConAttributi::elencaAttributi(void) const{
    cout << "Lato = " << q.getLato() << endl;
    attributi.elencaAttributi();
}//elencaAttributi

void QuadratoConAttributi::aggiungiAttributo(string nome, string
valore){
    attributi.aggiungiAttributo(nome, valore);
}//aggiungiAttributo
```

12.7 I DESIGN PATTERN

La progettazione di applicazioni orientate agli oggetti può essere molto complessa.
Non sempre è facile individuare le classi e le relazioni più adatte.

I **design pattern** sono degli schemi che descrivono soluzioni riutilizzabili di particolari problematiche di progettazione.

I design pattern usano molte astrazioni in modo da limitare le dipendenze.
In pratica ogni pattern descrive un problema che si ripete più volte, ma la soluzione è espressa in modo così generale da lasciare molta libertà nelle scelte di implementazione: ogni schema può essere applicato ogni volta in modo particolare e diverso.

Un uso eccessivo dei design pattern però può portare a realizzare soluzioni troppo complicate. L'obiettivo dovrebbe essere quello di trovare soluzioni semplici riutilizzando quando possibile soluzioni già studiate.

Ogni design pattern è identificato da un **nome** e descrive un particolare **problema di progettazione** (la situazione a cui si può applicare il pattern), la sua **soluzione** (in modo astratto, illustrando il diagramma delle classi da utilizzare, con ruoli, dipendenze e collaborazioni) e le **conseguenze** (risultati dell'applicazione del pattern e vincoli di utilizzo).

I pattern GoF

Esistono molti gruppi (chiamati anche cluster) di design pattern, suddivisi in categorie.
I design pattern più noti sono quelli descritti da Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (conosciuti anche come la **Gang of Four**, GoF) nel libro "Design Patterns: Elements of Reusable Object-Oriented Software" pubblicato nel 1995.

I **pattern GoF** (Gang of Four) sono 23 pattern e sono organizzati in tre categorie:

- cinque **pattern creazionali**, che riguardano la creazione di istanze;

Abstract Factory	fornisce un'interfaccia per creare gruppi di oggetti correlati o dipendenti senza specificare le classi concrete;
Builder	separa il procedimento di costruzione di un oggetto complesso dalla sua rappresentazione, permettendo di usare lo stesso processo di costruzione per creare rappresentazioni differenti;
Factory Method	definisce un'interfaccia per creare un oggetto, lasciando alle classi derivate il compito di creare effettivamente un'istanza;
Prototype	specifica un oggetto da usare come prototipo per la creazione di nuovi oggetti;
Singleton	assicura che si possa creare una sola istanza di una classe;

- sette **pattern strutturali**, che si riferiscono alla composizione di classi e oggetti;

Adapter	permette a classi con interfacce diverse di collaborare tra loro, trasformando l'interfaccia di una classe in un'altra interfaccia compatibile con quella dell'altra classe;
Bridge	disaccoppia un'astrazione dalla sua implementazione in modo che ciascuna possa variare in modo indipendente;
Composite	gestisce una gerarchia di oggetti in cui ogni oggetto può essere un oggetto singolo o a sua volta una composizione di altri oggetti;
Decorator	permette di aggiungere dinamicamente funzionalità ad un oggetto;
Facade	definisce un'interfaccia unificata per un sistema, mascherandone la complessità interna;
Flyweight	condivide parte dello stato di un oggetto, consentendo di gestire molti oggetti simili ma con parti che variano (oggetti a granularità fine);
Proxy	permette di controllare l'accesso ad un oggetto, in modo trasparente, tramite un oggetto sostitutivo più semplice;

- undici **pattern comportamentali**, che si occupano delle modalità con cui classi e oggetti interagiscono tra loro in relazione alle loro diverse responsabilità.

Chain of Responsability	gestisce il passaggio di una richiesta dal mittente al destinatario;
Command	incapsula una richiesta in un oggetto, rendendo possibile la gestione di diverse tipologie di richieste (bufferizzate (*queue*), registrate (*log*) e annullabili (*undo*));
Interpreter	permette di interpretare frasi di un linguaggio, perché definisce una rappresentazione della sua grammatica e del relativo interprete;
Iterator	fornisce un modo per accedere sequenzialmente agli elementi di una collezione senza doverne conoscere la rappresentazione;
Mediator	permette di implementare una funzionalità componendo il comportamento di diversi oggetti, incapsulandone le modalità di interazione;
Memento	permette di ricordare lo stato interno di un oggetto in modo tale da poter riportare successivamente l'oggetto allo stato originale;

Observer	permette di notificare a più oggetti il cambiamento di stato di un oggetto;
State	permette ad un oggetto di modificare il suo comportamento al variare del suo stato interno;
Strategy	permette di definire una famiglia di algoritmi che possono essere usati in modo intercambiabile fra loro;
Template Method	permette di definire la struttura di base di un algoritmo, lasciando alle classi derivate la possibilità di specificarne alcuni passi;
Visitor	rappresenta un'operazione che deve essere eseguita sugli elementi di una gerarchia di oggetti.

Nota

L'argomento dei design pattern è molto complesso. Viene descritto un pattern di ciascuna categoria.

Il design pattern Builder

Builder è un pattern creazionale; separa il procedimento di costruzione di un oggetto complesso dalla sua rappresentazione, permettendo di usare lo stesso processo di costruzione per creare rappresentazioni differenti.

È utile quando si desiderano rappresentazioni diverse dello stesso oggetto (per esempio versioni HTML, RTF o PDF dello stesso documento), o quando l'algoritmo di costruzione di un oggetto è complesso ed è preferibile considerare la costruzione dell'oggetto complessivo come creazione e assemblaggio delle singole parti che lo compongono.

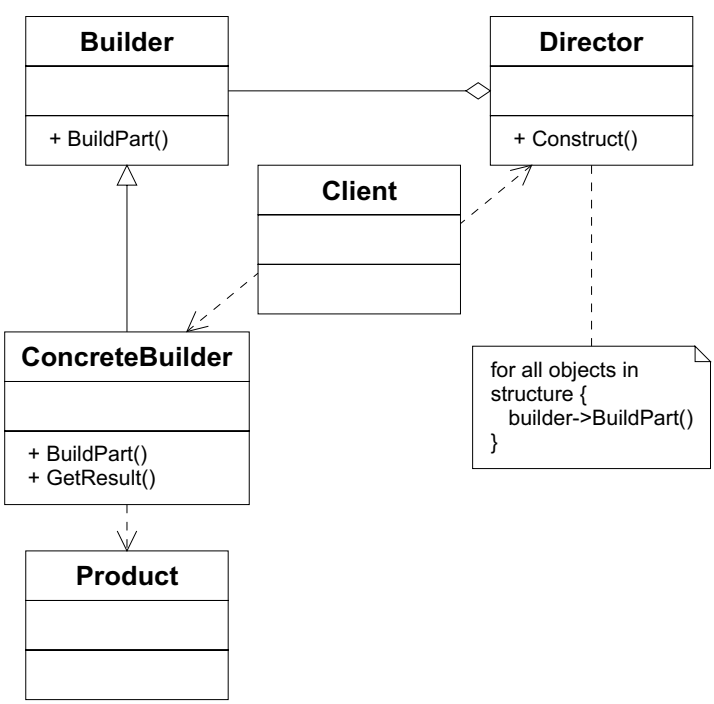


Figura 12.23 *Diagramma UML del design pattern Builder*

I partecipanti di questo pattern sono:

Builder	interfaccia o classe astratta per la creazione delle parti che costituiscono l'oggetto da costruire;
ConcreteBuilder	implementazione di *Builder* per la costruzione di ciascuna parte dell'oggetto complessivo; definisce un metodo per la costruzione *buildPart()* e uno per l'accesso alla singola parte *getResult*() ;

Director	assembla l'oggetto mediante il metodo *construct()*;
Product	rappresenta l'oggetto complessivo, risultato dell'operazione di costruzione e assemblaggio.

ESEMPIO 12.8

Un esempio molto semplice di applicazione del pattern può essere la costruzione di un'automobile a partire da un telaio, quattro ruote e un motore.

L'algoritmo di costruzione, il metodo *creaAuto()* (*construct()*) dell'oggetto *CostruttoreAuto* (*Director*), costruisce l'*Auto* (*Product*), assemblando gli oggetti *Telaio*, *Motore* e un array di quattro elementi *Ruota* (ciascuno costruito dal metodo *buildPart()* di un *ConcreteBuilder*).

Ogni componente (*ConcreteBuilder*) ha un costruttore che equivale al metodo *buildPart()* e un sovraccarico dell'operatore << che equivale al metodo *getResult()*.
La classe *CostruttoreAuto* incaricata di costruire l'oggetto *Auto* definisce il metodo *creaAuto()* (*construct()*); *creaAuto()* è un metodo statico che accetta i parametri di costruzione validi per le diverse parti e richiama i costruttori per la generazione delle istanze dei componenti. Il sovraccarico dell'operatore << della classe *Auto* richiama implicitamente i sovraccarichi relativi alle parti costituenti per ottenere una rappresentazione completa dell'oggetto.

```
//Telaio.h
#include <iostream>
using namespace std;

class Telaio{
    public:
        friend ostream& operator<<(ostream&, const Telaio&);
}; //Telaio
```

```
//Telaio.cpp
#include "Telaio.h"

ostream& operator<<(ostream& os, const Telaio& t){
    os << "Telaio";
    return os;
}//operator<<
```

```
//Ruota.h
#include <iostream>
using namespace std;

class Ruota{
    private:
        double diametro;
    public:
        friend ostream& operator<<(ostream& os, const Ruota& r);
        Ruota(double =0);
}; //Ruota
```

```
//Ruota.cpp
#include "Ruota.h"
Ruota::Ruota(double diametro){
    this->diametro = diametro;
```

```
}//Ruota

ostream& operator<<(ostream& os, const Ruota& r){
    os << "\nRuota: " << r.diametro;
    return os;
}//operator<<
- - - - - - -
//Motore.h
#include <iostream>
using namespace std;

class Motore{
    private:
        double potenza;
    public:
        friend ostream& operator<<(ostream&, const Motore&);
        Motore(double =0);
}; //Motore
- - - - - - - - -
//Motore.cpp
#include "Motore.h"

Motore::Motore(double potenza){
    this->potenza = potenza;
}//Motore

ostream& operator<<(ostream& os, const Motore& m){
    os << "\nMotore: " << m.potenza;
    return os;
}//operator<<
- - - - - -
//Auto.h
#include "Telaio.h"
#include "Motore.h"
#include "Ruota.h"

class Auto{
    private:
        Telaio telaio;
        Motore motore;
        Ruota ruote[4];
    public:
        void setRuote(double);
        void setTelaio(void);
        void setMotore(double);
        friend ostream& operator<<(ostream&, const Auto&);
}; //Auto
- - - - - - -
//Auto.cpp
#include "Auto.h"

void Auto::setTelaio(void){
    telaio = Telaio();
}//setTelaio
```

```
void Auto::setMotore(double p){
    motore = Motore(p);
}//setMotore

void Auto::setRuote(double d){
    for(int i=0; i<4; i++)
        ruote[i] = Ruota(d);
}//setRuote

ostream& operator<<(ostream& os, const Auto& a){
    os << a.telaio << " " << a.motore;
    for(int i=0; i<4; i++)
        os << " " << a.ruote[i];
    return os;
}//operator<<
```

```
//CostruttoreAuto.h
#include "Auto.h"

class CostruttoreAuto{
    public:
        static Auto creaAuto(double, double);
}; //CostruttoreAuto
```

```
//CostruttoreAuto.cpp
#include "CostruttoreAuto.h"

Auto CostruttoreAuto::creaAuto(double diametro, double potenza){
    Auto aut;
    aut.setRuote(diametro);
    aut.setTelaio();
    aut.setMotore(potenza);
    return aut;
}//creaAuto
```

```
//ProvaAuto.cpp
#include <iostream>
#include "CostruttoreAuto.h"
using namespace std;
int main(void){
    Auto automobile = CostruttoreAuto::creaAuto(80.2, 147.4);
    cout << automobile << endl;
    getchar();
    return 0;
}//main
```

Il design pattern Composite

Composite è un pattern strutturale; gestisce una gerarchia di oggetti in cui ogni oggetto può essere un oggetto singolo o a sua volta una composizione di altri oggetti.

È utile quando si devono gestire strutture ad albero per rappresentare gerarchie tutto-parti, mantenendo la possibilità di trattare in modo uniforme sia gli oggetti che le composizioni di oggetti.

Il client può trattare i due tipi di oggetti in maniera uniforme; non ha bisogno di sapere se si tratta di un oggetto singolo o composto.

Nota È molto usato nella progettazione di interfacce grafiche con contenitori e componenti.

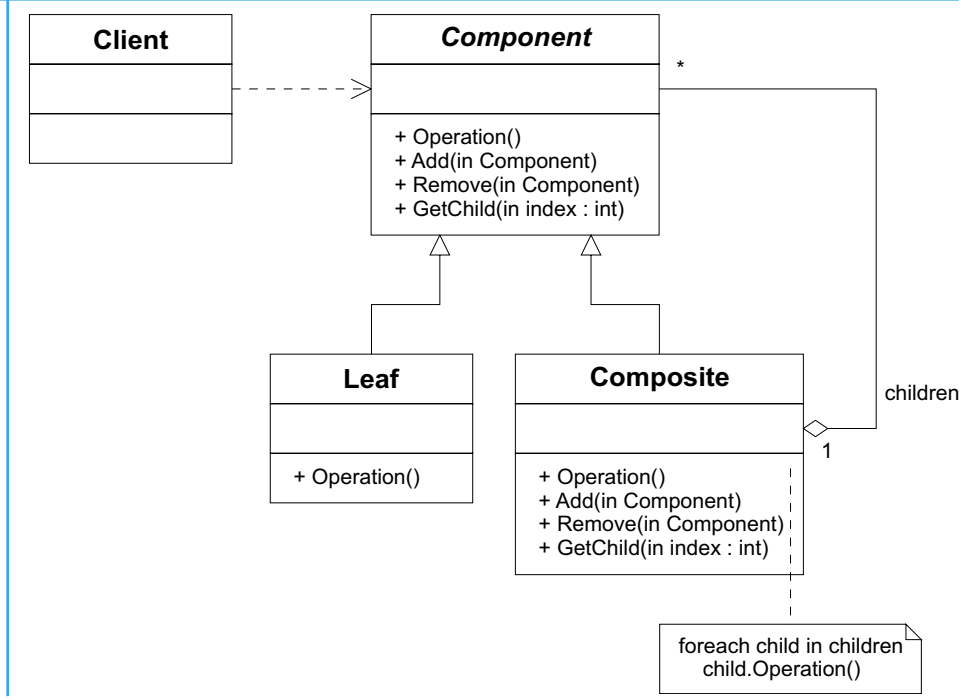


Figura 12.24 *Diagramma UML del design pattern Composite*

I partecipanti di questo pattern sono:

Component	interfaccia o classe astratta che fa da base comune sia ai nodi foglia che ai nodi composti; dichiara sia operazioni tipiche delle primitive (*operation()*) che quelle del contenitore, come ad esempio delle operazioni per accedere e gestire gli oggetti contenuti (figli);
Leaf	rappresenta un oggetto singolo, una singola parte;
Composite	rappresenta un oggetto composto (che può essere composto da altri Composite in modo ricorsivo); si può usare qualsiasi struttura per mantenere gli oggetti figli; non implementa *operation()* direttamente ma delega l'operazione a tutti gli oggetti che lo compongono (se per esempio il *Composite* usa un *vector* con i figli, lo scorre chiamando su ognuno di essi il metodo *operation()*).

Il design pattern Iterator

Iterator è un design pattern comportamentale; fornisce un modo per accedere sequenzialmente agli elementi di un contenitore senza dover conoscere la rappresentazione del contenitore stesso.

Per uno stesso contenitore si possono definire più iteratori che permettono di attraversare il contenitore con modalità diverse.
Si possono effettuare anche più attraversamenti contemporanei su una stessa struttura.
Non bisogna però modificare il contenitore durante l'attraversamento perché gli effetti sono imprevedibili.

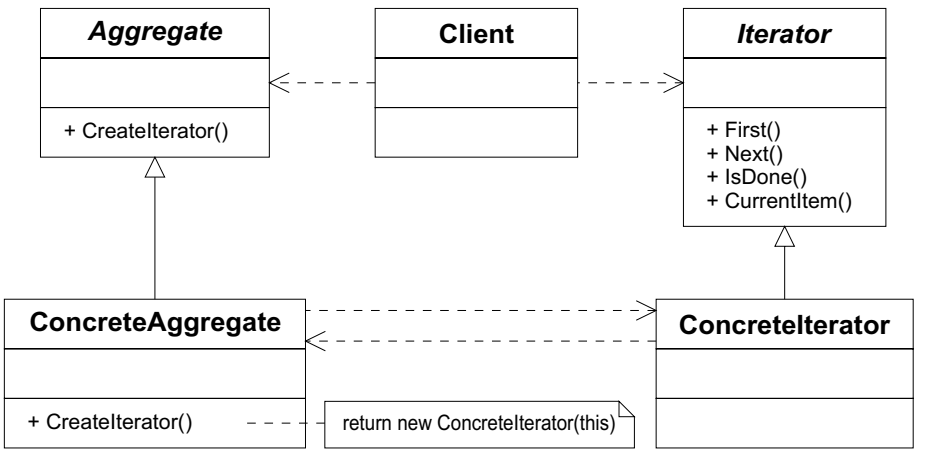


Figura 12.25 *Diagramma UML del design pattern Iterator*

I partecipanti di questo pattern sono:

Iterator	interfaccia o classe astratta che definisce almeno i metodi *first()* per posizionarsi sul primo elemento, *currentItem()* per accedere all'elemento corrente, *next()* per posizionarsi sul successivo e *isDone()* per verificare che l'attraversamento è concluso;
ConcreteIterator	definizione di un iteratore concreto; un iteratore ha tipicamente due variabili di istanza: un puntatore al contenitore e un cursore che permette di identificare e accedere a un elemento; il costruttore di solito posiziona il cursore sul primo elemento;
Aggregate	interfaccia o classe astratta che rende obbligatoria la definizione del metodo *newIterator()* per creare un iteratore;
ConcreteAggregate	implementa il metodo *newIterator()* che restituisce una istanza di *ConcreteIterator* per l'attraversamento del contenitore, passando *this* come parametro al suo costruttore.

Per ogni *ConcreteAggregate* si avrà in corrispondenza il *ConcreteIterator* in grado di attraversarlo (iterazione polimorfica). **Nota**

Si può trovare una dimostrazione dell'uso del pattern Iterator nell'esempio 13.9. **Nota**

VERIFICA

QUESTIONARIO

1. Quali sono le possibili relazioni tra le classi?
2. In che cosa consiste la relazione di ereditarietà?
3. In che cosa consiste la relazione di associazione?
4. Che differenza c'è tra le associazioni per composizione e per aggregazione?
5. In che cosa consiste la relazione di dipendenza?
6. Come viene rappresentata un'associazione in un diagramma UML?
7. Come vengono rappresentate composizioni e aggregazioni in un diagramma UML?
8. Come viene rappresentata una dipendenza in un diagramma UML?

9. Che cosa descrive un diagramma di collaborazione?
10. Che cosa descrive un diagramma di sequenza?
11. In che cosa consiste l'accoppiamento tra le classi?
12. Qual è la differenza tra un oggetto della classe *vector* e un array?
13. Che cosa sono i design pattern?

Esercizi

1 ..... Nell'esempio 12.1 sostituire il metodo *stampaVertici()* della classe *Poligono* con il sovraccarico dell'operatore << in modo da stampare i vertici uno dopo l'altro, ordinati rispetto all'ascissa.

2 ..... Aggiungere alla classe *Punto* dell'esempio 12.1 un metodo per spostare un punto e utilizzarlo per modificare i vertici di un *Poligono*.

3 ..... Nell'esempio 12.3 nel metodo *prodotto()* controllare le precondizioni e lanciare un'eccezione nel caso non siano soddisfatte.

Individuare le relazioni tra le classi e descriverle mediante un diagramma UML:

4 ..... un *Mazzo* di carte come insieme di *oggetti Carta*;

5 ..... un *Pixel* (un *Punto* con un colore) in relazione a un *Punto*;

6 ..... la relazione tra *Prodotti* e *Fornitori* di prodotti;

7 ..... un *Autore* di *Libri*;

8 ..... un *Dirigente* in relazione a un *Dipendente* in un'azienda;

9 ..... la relazione *Voto* che ogni *Studente* prende in ciascuna *Materia*;

10 ..... un *Triangolo* individuato da tre oggetti *Lato*, ognuno individuato da due oggetti *Punto*;

11 ..... una classe *Persona* e una classe *Matrimonio* che indica una coppia di persone sposate.

PROPOSTE DI LAVORO

Realizzare le classi

1 ..... ***Biblioteca***
attributo: array o *vector* di oggetti *Libro*;
metodi per:
- aggiungere un libro,
- togliere un libro,
- elencare i libri presenti.

2 ..... ***Famiglia***
attributo: array o *vector* di oggetti *Persona*, gestendo il ruolo della persona nella famiglia;
metodi per:
- aggiungere un componente alla famiglia,
- visualizzare i componenti della famiglia,
- togliere un componente.

3 ..... ***Geografia***
attributo: array o *vector* di oggetti *Paese* (nome, popolazione e area);
metodi per restituire:
- il paese con l'area più grande,

- il paese con la popolazione più numerosa,
- il paese con la maggior densità abitativa.

4 ..... ***Byte***
attributo: array o *vector* di oggetti *Bit*;
metodi per:
- impostare o azzerare un determinato bit,
- eseguire lo shift o la rotazione dei bit,
- eseguire le operazioni *OR* o *AND* con un altro byte.

5 ..... ***Classe*** (di una scuola)
attributi:
- nome,
- numero alunni,
- array o *vector* di oggetti *Studente*;

metodi per:
- aggiungere o togliere uno studente,
- elencare gli studenti della classe in ordine alfabetico.

6 ..... ***Scuola***
attributi:
- nome,
- numero delle classi,
- array o *vector* di oggetti *Classe*;

metodi per:
- aggiungere o togliere una classe,
- elencare le classi,
- elencare gli studenti della scuola.

7 ..... ***Agenda*** di appuntamenti
attributo: array o *vector* di oggetti *Appuntamento* (data, ora e descrizione); l'array deve essere ordinato;
metodi per:
- visualizzare tutti gli appuntamenti,
- visualizzare gli appuntamenti tra due orari dati,
- cercare un appuntamento dato l'orario (usare una ricerca binaria),
- inserire un nuovo appuntamento (individuare dove deve essere inserito un appuntamento e non aggiungerlo se è in conflitto con altri appuntamenti),
- cancellare un appuntamento.

8 ..... ***Spezzata*** (rappresenta una spezzata di n lati)
attributo: array o *vector* di oggetti *Lato* (ognuno individuato da due oggetti *Punto*);
metodi per:
- aggiungere un lato,
- ottenere il numero di lati,
- ottenere le coordinate dei punti dell'i-esimo lato,
- verificare se la spezzata è chiusa (l'ultimo punto coincide col primo),
- stampare la sequenza di punti.

9 ..... ***Mazzo*** di carte
attributo: array o *vector* di oggetti *Carta* (seme, valore);
metodi per:
- inizializzare il mazzo in modo ordinato,
- mescolare il mazzo,
- estrarre la prima carta dal mazzo,

- rimettere una carta in fondo al mazzo.

Scrivere un programma per giocare a carte contro il computer: il valore più alto vince; se i valori sono uguali vince il rosso, se sono uguali e dello stesso colore, fiori vincono su picche e cuori su quadri.

Realizzare le seguenti applicazioni: progettare le classi, eventualmente proponendo schemi diversi, e realizzarne l'implementazione in C++

10 ..... Gestione dei libri in una libreria.
Per esempio:
- classe *Libreria* come array di *Libro*,
- classe *Libro*,
- classe *Autore* (può ereditare da *Persona*),
- classe *Editore*.

Deve essere possibile tra l'altro sapere quali libri di un autore ci sono nella libreria, quali libri di un editore, le caratteristiche degli autori e dell'editore di un libro e così via.

11 ..... Gestione di una collezione di CD musicali con informazioni sui *CD* e gli *Artisti*.

12 ..... Gestione degli ordini dei clienti: un *Ordine* è relativo a un *Cliente* ed è costituito da più di una *LineaOrdine*, ognuna relativa a un *Articolo*.

13 ..... Gestione degli ordini ai fornitori: un *Ordine* è relativo a un *Fornitore* ed è costituito da più di una *LineaOrdine*, ognuna relativa a un *Articolo*; ogni *Articolo* può essere ordinato da un insieme di vari fornitori tra cui scegliere.

14 ..... Gestione dei voti degli studenti: si può usare una classe *Voto* in cui ogni voto è relativo a uno *Studente* e a una certa materia.

15 ..... Realizzare una classe *NumeroBinario* con le operazioni sui numeri binari.
Realizzare una classe *RappresentazioneNumeroIntero* con la rappresentazione in complemento alla base e le operazioni sulle rappresentazioni dei numeri.

16 ..... Realizzare un gioco per insegnare l'aritmetica con tre livelli di difficoltà:
- livello 1 addizioni tra numeri <10 con somma <10,
- livello 2 addizioni qualsiasi su numeri di 1 cifra,
- livello 3 sottrazioni su numeri di una sola cifra con risultati non negativi.

Generare problemi casuali e acquisire la risposta; dopo due errori visualizzare la risposta corretta; dopo cinque risposte esatte passare al livello superiore.

Design patterns

17 ..... Usare il design pattern Builder per costruire un documento multimediale a partire da vari tipi di media.

18 ..... Usare il design pattern Builder per creare una figura geometrica a partire da figure geometriche elementari.

19 ..... Usare il design pattern Composite per determinare il volume di un solido complesso a partire dalle figure solide elementari che lo costituiscono.

20 ..... Usare il design pattern Composite per determinare il tempo necessario al completamento di una prova (scolastica, sportiva, ecc.) o di un lavoro in base ai tempi necessari al completamento delle singole parti.

21 ..... Usare il design pattern Composite per calcolare le calorie ingerite in un pasto in base ai piatti scelti e ai loro ingredienti.

22 ..... Usare il design pattern Composite per stampare un libro dall'insieme dei singoli capitoli, insiemi di paragrafi.

23 ..... Studiare altri design pattern e realizzare applicazioni che li utilizzino.

13 La genericità

Prerequisiti

- Classi ed ereditarietà

Obiettivi

Conoscenze

- Scopo e vantaggi della genericità
- Diagrammi UML per la rappresentazione di classi generiche
- La classe generica *vector*
- Definizione di classi generiche
- Relazione di ereditarietà tra classi generiche
- Definizione di funzioni generiche
- Definizione di classi generiche con parametri di tipo predefinito e con parametri valore

Abilità/capacità

- Utilizzare la classe generica *vector*
- Definire ed utilizzare classi e funzioni generiche

13.1 LA GENERICITÀ

La **genericità** è la costruzione di una classe (classe generica o parametrica, o template class) che usa al suo interno una o più classi specificate solo in fase di esecuzione.

Usando la genericità si può creare per esempio una classe contenitore generica che può contenere oggetti di qualche classe specifica precisata durante l'esecuzione (per esempio un vettore di persone, articoli ecc.).

I tipi parametrici si possono usare nella definizione di classi e metodi.
I tipi generici usati come parametri di metodi realizzano il polimorfismo parametrico.

VANTAGGI DELLA GENERICITÀ

L'uso della genericità permette di evitare i cast rendendo il programma:

- più leggibile;
- più veloce: i cast rendono i programmi più lenti per la verifica del tipo;
- più sicuro: si sa già il tipo di oggetti da trattare e non si possono usare oggetti di tipo diverso; i tipi diversi sono riconosciuti dal compilatore che impedisce la compilazione del programma.

13.2 CLASSI GENERICHE IN UML

La **classe parametrica** viene indicata con un rettangolo tratteggiato in alto a destra che specifica un tipo parametrico.

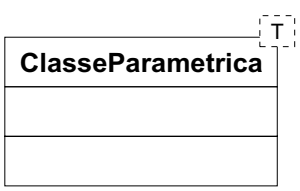


Figura 13.1 *Classe parametrica*

La classe che usa un tipo specifico indica nel rettangolo il nome completo del tipo specifico.

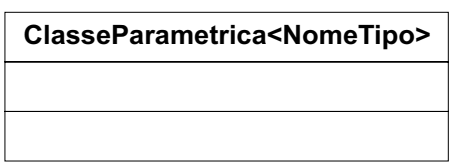


Figura 13.2 *Classe che usa un tipo specifico*

LA CLASSE VECTOR

C++ 13.1

La classe **vector** è una classe generica definita nel file ***<vector>*** della libreria **STL**, che deve essere incluso con la direttiva *#include*.

COSTRUTTORI

`vector()` crea un vettore vuoto;
`vector(size_t capacita, Tipo el)`
crea un vettore con capacità iniziale *capacita* riempito con elementi di valore *el*;
`vector(size_t capacita)`
crea un vettore con la capacità iniziale specificata.

Metodi principali

Metodi per esaminare il contenuto del vettore:

`Tipo front()`	restituisce il primo elemento (all'indice 0) del vettore;
`Tipo back()`	restituisce l'ultimo elemento del vettore;
`Tipo at(size_t index)`	restituisce l'elemento alla posizione specifica;
`iterator begin()`	restituisce l'iteratore che punta al primo elemento del vettore;
`iterator end()`	restituisce l'iteratore che punta alla posizione successiva a quella dell'ultimo elemento del vettore;
`reverse_iterator rbegin()`	restituisce l'iteratore che punta all'ultimo elemento del vettore;
`reverse_iterator rend()`	restituisce l'iteratore che punta alla posizione precedente a quella del primo elemento.

Metodi per modificare il contenuto del vettore:

`void push_back(Tipo o)` aggiunge l'elemento in fondo al vettore;

`iterator insert(iterator index, Tipo el)`
inserisce l'elemento alla posizione indicata dall'iteratore e sposta tutti gli elementi da qui alla fine di una posizione verso destra; restituisce un oggetto *iterator* all'elemento inserito;

`void assign(size_t n, Tipo elemento)`
riempie il vettore con *n* elementi di valore *elemento*;

`iterator erase(iterator index)`
rimuove l'elemento alla posizione indicata dall'iteratore e sposta tutti gli elementi da qui alla fine di una posizione verso sinistra (e decrementa la dimensione); restituisce un oggetto *iterator* all'elemento successivo;

`void pop_back()` rimuove l'ultimo elemento del vettore;

`void clear()` rimuove tutti gli elementi dal vettore.

Metodi per gestire la lunghezza del vettore:

`bool empty()` controlla se il vettore contiene elementi;

`size_t size()` restituisce il numero di elementi;

`void resize(size_t newSize)`
stabilisce la dimensione del vettore; se è minore di quella corrente gli elementi in più vengono persi.

Nota

Per cercare un elemento all'interno del vettore si può usare la funzione ***find()*** definita nel file ***<algorithm>*** della libreria **STL**.

Un oggetto della classe *vector* è un **vettore** di **oggetti** a **dimensione variabile**.

La dichiarazione di un oggetto *vector* è:

```
vector<TipoElemento> nomeVettore;
```

La classe *vector* sovraccarica l'operatore [ ]; un oggetto *vector* può essere usato con la notazione degli array.

Nota

Per accedere all'elemento di indice *i* di un vettore v si possono usare le notazioni *v[i]* oppure *v.at(i)*; quest'ultima è da preferire in quanto il metodo *at()* lancia l'eccezione *out_of_range* se *i* non è un valore valido.

```
vector<int> numeri;
numeri.insert(numeri.begin(),500);
```
crea il vettore vuoto *numeri* per contenere elementi di tipo *int* e inserisce come primo elemento il valore 500.

La classe *vector* supporta anche il tipo *const_iterator* per definire un iteratore in grado di scorrere in sola lettura i valori del vettore.

Per esempio per stampare tutti gli elementi di un vettore *v* di *int* usando un iteratore di tipo *const_iterator* si può fare:

```
vector<int>::const_iterator i;
    for(i = v.begin(); i != v.end(); i++)
        cout << *i << endl;
```

ESEMPIO 13.1

Classe *Poligono* che ha come attributo un *vector* di oggetti *Punto*.

```
//Poligono.h
#include <vector>
#include "Punto.h"
class Poligono {
    private:
        vector<Punto> vertici;
    public:
        void aggiungiVertice(const Punto&);
        void stampaVertici(void) const;
        double getPerimetro(void) const;
        Punto getVertice(int) const;
};//Poligono
```

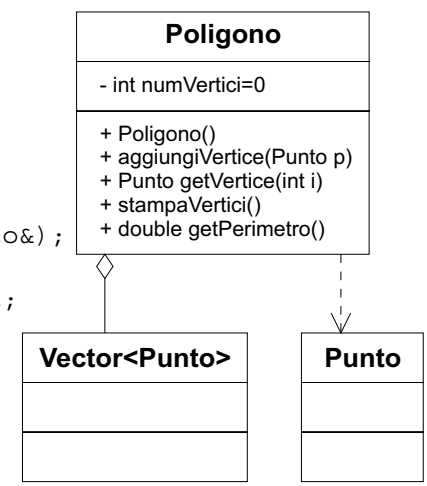


Figura 13.3 *Diagramma UML delle classi dell'esempio*

```
//Poligono.cpp
#include <iostream>
#include "Poligono.h"
using namespace std;

Punto Poligono::getVertice(int i) const{
    if(i<vertici.size())
        return vertici[i];
}//getVertice

void Poligono::aggiungiVertice(const Punto& p){
    vertici.push_back(p);
}//aggiungiVertice

double Poligono::getPerimetro(void) const{
    double perimetro = 0;
    for(int i=0; i<vertici.size()-1; i++)
        perimetro += vertici[i].distanza(vertici[i+1]);
    perimetro += vertici[vertici.size()-1].distanza(vertici[0]);
    return perimetro;
}//getPerimetro

void Poligono::stampaVertici() const{
    for(int i=0; i<vertici.size(); i++)
```

```
        cout << vertici[i] << endl;
}//stampaVertici
```

con un iteratore di tipo *const_iterator*:

```
void Poligono::stampaVertici() const{
    vector<Punto>::const_iterator i;
    for(i = vertici.begin(); i != vertici.end(); i++)
        cout << *i << endl;
}//stampaVertici
```

C++ 13.2

TEMPLATE

La genericità in C++ è realizzata tramite l'adozione di un **template** (modello).

Vi sono due tipi di template: di classe e di funzione.

Un **template di classe** è definito dallo schema

```
template <class T>
class nomeClasse{
   //attributi e metodi
};
```

T rappresenta una variabile di tipo.

Al posto della lettera T potrebbe essere utilizzato qualsiasi nome, a scelta; di solito si usano nomi di una sola lettera maiuscola.

Nota

Per **usare la classe generica** si deve sostituire alla variabile di tipo il nome di una classe specifica:

```
NomeClasseGenerica<NomeClasseSpecifica>
```

Il compilatore sostituisce tutte le occorrenze della variabile di tipo con il tipo degli oggetti che si vogliono utilizzare, specificato tra parentesi angolari.
In questo modo è il compilatore che controlla che vengano usati solo oggetti del tipo specificato e si evitano errori di esecuzione.

Il **tipo** degli oggetti può essere una **classe** o anche un tipo primitivo.

La classe risultante dall'applicazione del template mediante l'istruzione *typedef* può diventare un tipo per nascondere la presenza del template.

La classe *string* è definita come tipo sinonimo della classe generica ***basic_string*** dopo l'applicazione del template con il tipo *char*:

```
typedef basic_string<char> string;
```

Una classe generica può usare anche più tipi parametrici:

```
template <class T1, ..., class TN>
```

Per i nomi dei tipi generici si usano le seguenti convenzioni:

T	tipo generico
S,U,V	altri tipi generici
N	tipo numerico
E	elemento di una collezione

Per le mappe (collezioni di tipo *map*) si usano *K* per indicare la chiave e *V* per indicare il valore.

Nota

Il tipo parametrico in una classe generica si può utilizzare per dichiarare variabili di istanza o come tipo dei parametri o del valore di ritorno dei metodi.

Il tipo parametrico può essere usato anche nel corpo del metodo. **Nota**

Un metodo della classe viene implementato con la sintassi:

```
template <class T> TipoRitorno
NomeClasse<T>::nomeMetodo(listaParametri){
    //istruzioni
}//nomeMetodo
```

Una classe generica solitamente è salvata in un unico file **.h**. La separazione in due file distinti porta alla segnalazione di errori in fase di link da parte di molti compilatori. **Nota**

Esempio 13.2

Realizzazione di una classe generica *Scatola*, che può contenere un oggetto di qualsiasi tipo.

```
//Scatola.h
template <class T>
class Scatola {
    private:
        T valore;
    public:
        T getValore(void) const;
        Scatola(T);
};//Scatola
```

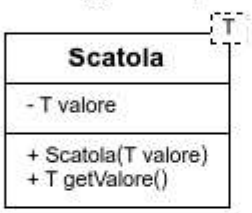


Figura 13.4 *Diagramma UML delle classi dell'esempio*

```
template <class T> Scatola<T>::Scatola(T valore){
    this->valore = valore;
}//Scatola

template <class T> T Scatola<T>::getValore(void) const{
    return valore;
}//getValore
```

Figura 13.5 *Esecuzione dell'esempio*

```
- - - - - - - - - - - - - -
//ProvaScatola.cpp
#include <iostream>
#include <string>
#include "Scatola.h"
using namespace std;

int main(void){
    Scatola<string> scatolas = Scatola<string>("Uno");
    cout << scatolas.getValore() << endl;
    Scatola<int> scatolai = Scatola<int>(1234);
    cout << scatolai.getValore() << endl;
    getchar();
    return 0;
}//main
```

Esempio 13.3

Realizzazione di una classe generica *Coppia*, che può contenere una coppia di oggetti di qualsiasi tipo.

```
//Coppia.h
template <class T, class S>
class Coppia {
    private:
        T primo;
        S secondo;
    public:
        T getPrimo(void) const;
        S getSecondo(void) const;
        Coppia(T, S);
};//Coppia
```

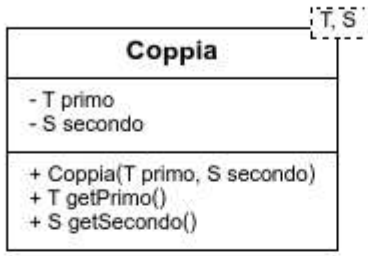


Figura 13.6 *Diagramma UML delle classi dell'esempio*

```
template <class T, class S> Coppia<T, S>::Coppia(T p, S s){
    primo = p;
    secondo = s;
}//Coppia

template <class T, class S> T Coppia<T, S>::getPrimo(void) const{
    return primo;
}//getPrimo

template <class T, class S> S Coppia<T, S>::getSecondo(void)
const{
    return secondo;
}//getSecondo

- - - - - - - - - - - - - - -
class ProvaCoppia {
//ProvaCoppia.cpp
#include <iostream>
#include <string>
#include "Coppia.h"
using namespace std;

int main(void){
    Coppia<string, int> coppia1("Uno", 200);
    Coppia<long, string> coppia2(123456789, "Due");
    cout << coppia1.getPrimo() << " "
         << coppia1.getSecondo() << endl;
    cout << coppia2.getPrimo() << " "
         << coppia2.getSecondo() << endl;
    getchar();
    return 0;
}//main
```

Figura 13.7 *Esecuzione dell'esempio*

```
Coppia<String, int> coppia1("Uno", 200);
```

può essere sostituita da

```
Coppia<String, int> coppia1 = Coppia<String, int>("Uno", "Due");
```

ma non da

```
Coppia coppia1 = Coppia<String, int>("Uno", "Due");
```

segnalato dal compilatore come un errore di omissione del template.

EREDITARIETÀ DELLE CLASSI GENERICHE

La relazione di **ereditarietà tra classi generiche** è determinata dalle **classi** e non dal tipo dei parametri.

```
Classe<Tipo>
SottoClasse<Tipo>
```
SottoClasse e *Classe* hanno una relazione gerarchica.

L'ereditarietà tra i tipi dei parametri invece non genera nessun legame di ereditarietà fra le classi generiche.

```
Classe<Tipo>
Classe<SottoTipo>
```
non hanno alcuna relazione gerarchica tra di loro anche se *SottoTipo* è una sottoclasse di *Tipo*.

Scatola<char> non è una sottoclasse di *Scatola<int>* anche se *char* è un sottotipo di *int*.
Il seguente codice non è valido:

```
vector<char> car;
vector<int> numeri = car;
```
infatti l'operatore = richiama il costruttore copia che va a buon fine solo se i due oggetti sono dello stesso tipo oppure di classi derivate.

Template di classe con parametri di tipo predefinito

Una classe generica può indicare come tipo generico un tipo predefinito. A seconda dell'istanziazione del template verrà creata l'opportuna classe.

```
template <class T =TipoPredefinito>
class nomeClasse{
    //attributi e metodi
};
```

Un oggetto di *nomeClasse* può essere creato anche con:

```
nomeClasse<> classe;
```

in cui l'istanziazione del template avviene con il tipo *TipoPredefinito*.

Template di classe con parametri valore

Il tipo parametrico di un template può essere anche un valore. Questo risulta utile nel caso di classi generiche con attributi e metodi statici.

Il template risultante viene chiamato **template con parametri valore**; può essere specificato anche un valore predefinito.

```
template <Tipo VT =valore>
class nomeClasse{
    //attributi e metodi
};
```

Un oggetto di *nomeClasse* può essere creato anche con:

```
nomeClasse<valore> classe;
```

Metodi e attributi statici nelle classi generiche

Nelle **classi generiche** è possibile dichiarare metodi e attributi statici. Gli attributi statici sono inizializzati alla creazione del primo oggetto; oggetti di tipi diversi sono distinti, cioè viene creata una copia dell'attributo statico per ogni tipo utilizzato.

Esempio 13.4

Modifica della classe generica *Scatola*. Vengono utilizzati un attributo statico, inizializzato mediante un parametro valore, e un metodo statico.

```
//ScatolaVS.h
template <class T, unsigned int I>class Scatola {
    private:
        T valore;
        static unsigned int numScatole;
    public:
        T getValore(void) const;
        static unsigned int getNumScatole(void);
        Scatola(T);
};//Scatola
template <class T, unsigned int I>
unsigned int Scatola<T, I>::numScatole = I;

template <class T, unsigned int I>
Scatola<T, I>::Scatola(T valore){
    this->valore = valore;
    numScatole++;
}//Scatola

template <class T, unsigned int I>
T Scatola<T, I>::getValore(void) const{
    return valore;
}//getValore

template <class T, unsigned int I>
unsigned int Scatola<T, I>::getNumScatole(void){
    return numScatole;
}//getNumScatole
```

```
//ProvaScatolaVS.cpp
#include <iostream>
#include "ScatolaVS.h"
using namespace std;

int main(void){
    Scatola<string, 10> scas1("Uno");
    Scatola<string, 10> scas2("Due");
    Scatola<int, 10> scai1(1234);
    Scatola<int, 20> scai2(12345);
    Scatola<string, 10> scas3("Tre");
    cout << scas1.getValore() << endl;
    cout << "numScatole: "
```

Figura 13.8 *Esecuzione dell'esempio*

```
        << Scatola<string, 10>::getNumScatole() << endl;
    cout << scai1.getValore() << endl;
    cout << "numScatole: "
         << Scatola<int, 10>::getNumScatole() << endl;
    cout << scas2.getValore() << endl;
    cout << "numScatole: "
         << Scatola<string, 10>::getNumScatole() << endl;
    cout << scai2.getValore() << endl;
    cout << "numScatole: "
         << Scatola<int, 20>::getNumScatole() << endl;
    cout << scas3.getValore() << endl;
    cout << "numScatole: "
         << Scatola<string, 10>::getNumScatole() << endl;
    getchar();
    return 0;
}//main
```

TEMPLATE DI FUNZIONI

Un **template di funzione** permette la costruzione di una funzione (funzione generica) che usa al suo interno uno o più tipi (anche di classe) specificati solo in fase di esecuzione.

```
template <tipo Tipo1, ..., tipo TipoN>
TipoDiRitorno nomeFunzione(listaParametri);
```

Tipo1, ..., *TipoN* sono i tipi parametrici che possono comparire come *TipoDiRitorno* e nella *listaParametri*;

tipo può essere ***typename*** se il tipo parametrico è primitivo o ***class*** se è una classe.

La chiamata di una funzione generica è detta **istanziazione del template**; la corrispondenza nell'ordine dei parametri determina il tipo sostituito al tipo generico.

ESEMPIO 13.5

Realizzazione e collaudo della funzione generica *piuGrande*, che ha un tipo parametrico primitivo.

```
#include <iostream>
using namespace std;

template <typename Tipo> string piuGrande(Tipo, Tipo);

int main(void){
    int x = 6, y = 7;
    double v = 34.56, z = 23.53;
    cout << x << piuGrande(x, y) << y << endl;
    cout << v << piuGrande(v, z) << z << endl;
    getchar();
    return 0;
}//main

template <typename Tipo>
string piuGrande(Tipo a, Tipo b){
```

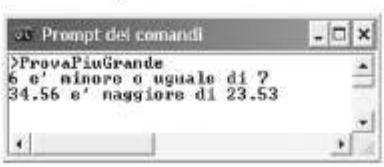


Figura 13.9 *Esecuzione dell'esempio*

```
    if(a > b)
        return " e' maggiore di ";
    else
        return " e' minore o uguale di ";
}//piuGrande
```

Esempio 13.6

Realizzazione e collaudo della funzione generica *stampa*, che riceve un array di qualsiasi tipo e ne visualizza il contenuto.

Nota: La classe *Punto* deve sovraccaricare l'operatore <<.

```
#include <iostream>
#include "Punto.h"
using namespace std;

template <class T> void stampa(T[], int);

int main(void){
    long a[] = {1234, 2345, 3456, 4567, 5678};
    string b[] = {"aaaa", "bbb", "ccccc"};
    Punto c[] = {Punto(4,6), Punto(6,-7), Punto(), Punto(-5,-3)};
    stampa(a, 5);
    stampa(b, 3);
    stampa(c, 4);
    getchar();
    return 0;
}//main

template <class T> void stampa(T array[], int dim){
    for(int i=0; i<dim; i++)
        cout << array[i] << " ";
    cout << endl;
}//stampa
```

Figura 13.10 *Esecuzione dell'esempio*

C++ 13.3 Realizzazione di classi contenitori generiche

Per realizzare una classe contenitore **generica** basta specificare nella dichiarazione della classe un tipo generico e usare il tipo generico per ricevere, restituire o memorizzare elementi.
Se la classe usa classi ausiliarie come nodi o iteratori che non siano classi interne, anche le classi ausiliarie devono diventare generiche.

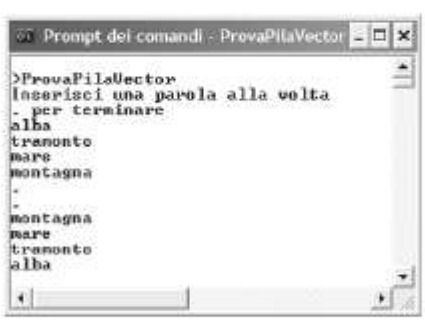
Esempio 13.7

Realizzazione di una pila con *vector*.

```
//PilaVector.h
#include <vector>

template <class E> class PilaVector {
    private:
        vector<E> pila;
```

```
    public:
        void push(E);
        E top(void) const;
        E pop(void);
        int cima(void) const;
        bool vuota(void) const;
};//PilaVector
template <class E> void PilaVector<E>::push(E elemento){
    pila.push_back(elemento);
}//push
template <class E> E PilaVector<E>::top(void) const{
    if (vuota())
        cerr << "Pila vuota\n";
    else
        return pila.at(cima());
}//top
template <class E> E PilaVector<E>::pop(void){
    if (vuota())
        cerr << "Pila vuota\n";
    else{
        E elemento = pila.back();
        pila.erase(pila.begin()+cima());
        return elemento;
    }//else
}//pop
template <class E> int PilaVector<E>::cima() const{
    return pila.size() - 1;
}//cima
template <class E> bool PilaVector<E>::vuota() const {
    return pila.size() == 0;
}//vuota
```

Figura 13.11
Esecuzione dell'esempio




```
-----------------
//ProvaPilaVector.cpp
#include <iostream>
#include "PilaVector.h"
using namespace std;
int main(void){
    PilaVector<string> pila;
    string parola;
    cout << "Inserisci una parola alla volta"
         << "\n. per terminare\n";
    do{
        cin >> parola;
        pila.push(parola);
    }while(parola != ".");
    while(!pila.vuota()){
        parola = pila.pop();
        cout << parola << endl;
    }//while
```

```
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Esempio 13.8

Realizzazione di una lista concatenata generica.

```
//Nodo.h
using namespace std;

template <class E> class Nodo {
    private:
        E elemento;
        Nodo<E>* successivo;
    public:
        void setElemento(E);
        E getElemento(void) const;
        Nodo<E>* getSuccessivo(void) const;
        void setSuccessivo(Nodo<E>*);
        Nodo(E);
        Nodo(E, Nodo<E>*);
};//Nodo

template <class E> Nodo<E>::Nodo(E el){
    elemento = el;
    successivo = NULL;
}//Nodo

template <class E> Nodo<E>::Nodo(E el, Nodo<E>* succ){
    elemento = el;
    successivo = succ;
}//Nodo

template <class E> E Nodo<E>::getElemento(void) const{
    return elemento;
}//getElemento

template <class E> void Nodo<E>::setElemento(E el){
    elemento = el;
}//setElemento

template <class E> Nodo<E>* Nodo<E>::getSuccessivo() const{
    return successivo;
}//getSuccessivo

template <class E> void Nodo<E>::setSuccessivo(Nodo<E>* succ) {
    successivo = succ;
}//setSuccessivo
```

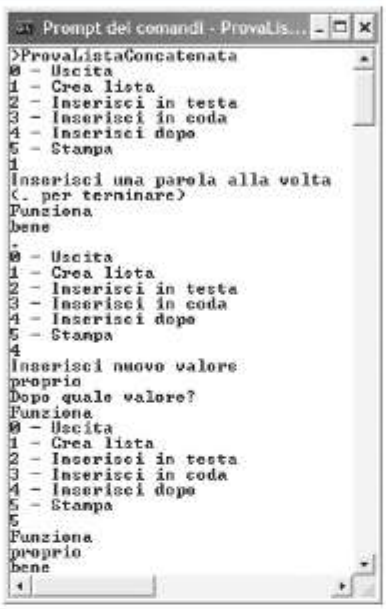


Figura 13.12 *Esecuzione dell'esempio*

```
- - - - - - - - - - - - - - -
//ListaConcatenata.h
#include "Nodo.h"

template <class E> class ListaConcatenata {
    private:
```

```
        Nodo<E>* testa;
    public:
        void inserisciInTesta(E);
        void inserisciInCoda(E);
        Nodo<E>* cerca(E) const;
        void inserisciDopo(E, E);
        void stampa(void) const;
        Nodo<E>* getTesta(void) const;
        ListaConcatenata(void);
};//ListaConcatenata

template <class E> ListaConcatenata<E>::ListaConcatenata(void){
    testa = NULL;
}//ListaConcatenata

template <class E>
Nodo<E>* ListaConcatenata<E>::getTesta(void) const{
    return testa;
}//getTesta

template <class E> void ListaConcatenata<E>::inserisciInTesta(E el){
    Nodo<E>* nuovo = new Nodo<E>(el);
    if(testa == NULL)
        testa = nuovo;
    else{
        Nodo<E>* succ = testa;
        nuovo->setSuccessivo(succ);
        testa = nuovo;
    }//else
}//inserisciInTesta

template <class E> void ListaConcatenata<E>::inserisciInCoda(E el){
    Nodo<E>* nuovo = new Nodo<E>(el);
    if(testa == NULL)
        testa = nuovo;
    else{
        Nodo<E>* succ = testa;
        while(succ->getSuccessivo() != NULL)
            succ = succ->getSuccessivo();
        succ->setSuccessivo(nuovo);
    }//else
}//inserisciInCoda

template <class E>
Nodo<E>* ListaConcatenata<E>::cerca(E valore) const{
    Nodo<E>* succ = testa;
    while((succ != NULL) && (succ->getElemento() != valore))
        succ = succ->getSuccessivo();
    return succ;
}//cerca

template <class E>
void ListaConcatenata<E>::inserisciDopo(E el, E valore){
    Nodo<E>* n = cerca(valore);
```

```
    if(n != NULL){
        Nodo<E>* nuovo = new Nodo<E>(el);
        nuovo->setSuccessivo(n->getSuccessivo());
        n->setSuccessivo(nuovo);
    }//if
    else
        inserisciInCoda(el);
}//inserisciDopo

template <class E> void ListaConcatenata<E>::stampa(void) const{
    Nodo<E>* succ = testa;
    while(succ != NULL){
        cout << succ->getElemento() << endl;
        succ = succ->getSuccessivo();
    }//while
}//stampa
```

```
//ProvaListaConcatenata.cpp
#include <iostream>
#include "ListaConcatenata.h"
using namespace std;

void provaCrea(ListaConcatenata<string>& lis) {
    string parola;
    cout << "Inserisci una parola alla volta"
         << " (. per terminare)\n";
    do {
        cin >> parola;
        if (parola != ".")
            lis.inserisciInCoda(parola);
    }while(parola != ".");
}//provaCrea

void stampa(const ListaConcatenata<string>& lis) {
    lis.stampa();
}//stampa

void provaInserisciInTesta(ListaConcatenata<string>& lis) {
    string parola;
    cout << "Inserisci nuovo valore\n";
    cin >> parola;
    lis.inserisciInTesta(parola);
}//provaInserisciInTesta

void provaInserisciDopo(ListaConcatenata<string>& lis) {
    string parola, valore;
    cout << "Inserisci nuovo valore\n";
    cin >> parola;
    cout << "Dopo quale valore?\n";
    cin >> valore;
    lis.inserisciDopo(parola, valore);
}//provaInserisciDopo

void provaInserisciInCoda(ListaConcatenata<string>& lis) {
```

```
    string parola;
    cout << "Inserisci nuovo valore\n";
    cin >> parola;
    lis.inserisciInCoda(parola);
}//provaInserisciInCoda

int main(void){
    ListaConcatenata<string> lista;
    ...
}//main
```

ESEMPIO 13.9

Creazione di un iteratore per una lista concatenata generica.
Viene utilizzato il design pattern Iterator: una *ListaAttraversabile* (*ConcreteAggregate*) estende la classe astratta *Attraversabile* (*Aggregate*) e definisce il metodo *newIterator()* per ottenere un *IteratoreLista* (*ConcreteIterator*) che estende la classe astratta *Iteratore* (*Iterator*) e definisce i metodi *first()*, *currentItem()*, *next()* e *isDone()*.

```
//Iteratore.h
template <class E> class Iteratore {
    public:

// porta il cursore sul primo elemento
        virtual void first(void) =0;

// elemento corrente indicato dal cursore
        virtual E currentItem(void) const =0;

// porta il cursore sull'elemento successivo
        virtual void next(void) =0;

// vero se non ci sono altri elementi
        virtual bool isDone(void) =0;
};//Iteratore
- - - - - - - - - - - - - -
//IteratoreLista.h
#include "Iteratore.h"
#include "ListaConcatenata.h"

template <class E>
class IteratoreLista : public Iteratore<E>{
    private:
        ListaConcatenata<E>* lista;
        Nodo<E>* cursore;
    public:
        void first(void);
        E currentItem(void) const;
        void next(void);
        bool isDone(void);
        IteratoreLista(ListaConcatenata<E>*);
};//IteratoreLista

template <class E>
IteratoreLista<E>::IteratoreLista(ListaConcatenata<E>* li){
```

```
    lista = li;
    first();
}//IteratoreLista

template <class E> void IteratoreLista<E>::first(void){
   cursore = lista->getTesta();
}//first

template <class E> E IteratoreLista<E>::currentItem(void) const{
   return cursore->getElemento();
}//currentItem

template <class E> void IteratoreLista<E>::next(void){
   cursore = cursore->getSuccessivo();
}//next

template <class E> bool IteratoreLista<E>::isDone(void){
   return cursore == NULL;
}//isDone
```

```
//Attraversabile.h
#include "IteratoreLista.h"

template <class E> class Attraversabile{
    public:

// crea un iteratore per attraversare il contenitore
        virtual IteratoreLista<E> newIterator(void) =0;
}; //Attraversabile
```

```
//ListaAttraversabile.h
#include "Attraversabile.h"

template <class E>
class ListaAttraversabile : public ListaConcatenata<E>,
public Attraversabile<E> {
    public:
        IteratoreLista<E> newIterator(void);
        ListaAttraversabile(void);
}; //ListaAttraversabile

template <class E>
ListaAttraversabile<E>::ListaAttraversabile(void) :
ListaConcatenata<E>(){
}//ListaAttraversabile

template <class E> IteratoreLista<E>
ListaAttraversabile<E>::newIterator(void) {
    return IteratoreLista<E>(this);
}//newIterator
```

```
//ProvaListaAttraversabile.h
#include <iostream>
#include "ListaAttraversabile.h"
using namespace std;
```

```
int main(void){
    ListaAttraversabile<string> lista;
    lista.inserisciInCoda("uno");
    lista.inserisciInCoda("due");
    lista.inserisciInCoda("tre");
    IteratoreLista<string> it = lista.newIterator();
    while (!it.isDone()) {
        cout << it.currentItem() << endl;
        it.next();
    }//while
    getchar();
    return 0;
}//main
```

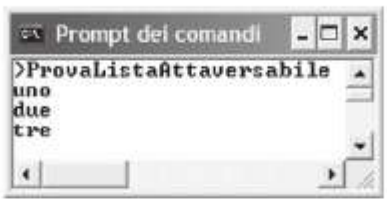


Figura 13.13 *Esecuzione dell'esempio*

VERIFICA

QUESTIONARIO

1. Che cos'è la genericità?
2. Come si rappresentano con UML le classi parametriche?
3. Come si specificano i tipi parametrici e di quale tipo possono essere?
4. Che cosa sono i template di classe con parametri valore?
5. Che cosa sono i template di funzione?

TEST

1 ..... *Indicare se le seguenti affermazioni sono vere o false.*

V	F	
☐	☐	1) La genericità permette di decidere la classe da usare durante l'esecuzione invece che durante la compilazione.
☐	☐	2) Un metodo generico è un metodo in cui è presente un parametro.
☐	☐	3) In una classe generica è possibile dichiarare un metodo statico generico.
☐	☐	4) In una classe generica gli attributi statici sono unici per tutti i tipi.
☐	☐	5) *vector<unsigned char>* è una sottoclasse di *vector<short>*.
☐	☐	6) La seguente istanziazione può essere corretta `Coppia<> c('A',5);`
☐	☐	7) Il seguente prototipo di funzione generica è corretto `template <class T, typename S> S elabora(T, S);`

☐	☐	8) Le seguenti definizioni sono equivalenti
		`Scatola<> s("contenuto");`
		`Scatola<> s = Scatola("contenuto");`

Esercizi

1 ..... Aggiungere alla gestione della pila dell'esempio 13.7 metodi per:
- contare il numero di elementi;
- invertire gli elementi (il primo elemento deve diventare l'ultimo e l'ultimo il primo);
- concatenare un'altra struttura dello stesso tipo;
- cercare la posizione di un valore;
- fondere due strutture ordinate dello stesso tipo.

2 ..... Aggiungere alla classe *ListaConcatenata* dell'esempio 13.8 metodi per:
- contare il numero di elementi della lista;
- concatenare alla lista un'altra lista;
- restituire il riferimento al nodo che contiene il valore che precede immediatamente un valore specificato;
- inserire un elemento prima di un nodo che contiene un valore specificato;
- eliminare il nodo che contiene un valore specificato;
- sostituire un nuovo valore in un nodo che contiene un valore specificato.

3 ..... Realizzare una implementazione di una pila usando la classe *ListaConcatenata* dell'esempio 13.8.

4 ..... Realizzare una implementazione di una coda usando la classe *ListaConcatenata* dell'esempio 13.8. Trasferire i dati da una coda realizzata con array a una coda realizzata usando la classe *ListaConcatenata*.

5 ..... Realizzare una implementazione di una coda usando un *vector*.

6 ..... Realizzare un albero binario generico e un iteratore per l'albero.

PROPOSTE DI LAVORO

1 ..... Realizzare la classe *Vettore* con attributo un *vector* generico per aggiungere, togliere ed elencare oggetti dello stesso tipo.

2 ..... Realizzare una classe *Numero* che possa contenere un numero di qualsiasi tipo.

3 ..... Realizzare la classe *Coppie* con due attributi *vector* generici che contengono oggetti dello stesso tipo, accoppiati in base alla posizione.

4 ..... Realizzare funzioni generiche per eseguire la ricerca e l'ordinamento su array di qualsiasi tipo.

5 ..... Realizzare una simulazione di una coda di persone a due o più sportelli analoghi tra loro (posta o banca ecc.), con un'unica coda o una coda per ogni sportello.

6 ..... Realizzare una implementazione di una lista concatenata bidirezionale (usando la classe *Nodo* dell'esempio 13.8).

7 ..... Realizzare una lista concatenata con un array o un *vector* di oggetti *Nodo*.
Gestire all'interno dell'array anche una lista dello spazio libero che colleghi tutti gli elementi liberi dell'array, in modo che gli elementi liberati in seguito a cancellazioni possano essere riutilizzati.
(All'inizio tutto l'array contiene elementi disponibili; bisogna inizializzare i puntatori in modo da formare una catena che colleghi tutti gli elementi ed inizializzare il puntatore alla prima posizione libera in modo che punti al primo elemento dell'array).

8 ..... Rappresentare un albero binario tramite un array o un *vector* di oggetti; ogni oggetto rappresenta un elemento (costituito dal dato e da due valori che indicano l'indice corrispondente al figlio di sinistra e l'indice corrispondente al figlio di destra).

La libreria STL

14

Prerequisiti

- Saper utilizzare una classe
- Conoscere i meccanismi dell'ereditarietà

Obiettivi

Conoscenze

- Conoscere le caratteristiche dei principali tipi di contenitori
- Conoscere le classi per la gestione delle strutture dati (collezioni o contenitori)

  In particolare le classi:

  stack

  list
- Conoscere i tipi di iteratore presenti nei contenitori
- Conoscere le funzioni generiche (algoritmi) per operare sui contenitori

Abilità/capacità

- Scandire gli elementi di un contenitore mediante iteratori di tipo *iterator* e *const_Iterator*
- Usare le classi delle collezioni, in particolare *stack* e *list*
- Usare gli algoritmi per operare sulle collezioni

C++

La libreria STL

14.1

La libreria STL (Standard Template Library) è una raccolta di classi per la gestione di strutture di dati (chiamate anche **collezioni** o contenitori), di classi iteratore, di classi contenenti algoritmi generici (per la ricerca, per l'ordinamento e altro) e funzioni varie di appoggio (soprattutto per l'overloading degli operatori).

Le classi STL e la genericità

Tutte le classi della libreria STL sono **generiche**.
La definizione include le parentesi angolari delle variabili di tipo.

La libreria STL è composta da un certo numero di file. Per utilizzare una classe definita in un file è necessario includere il file con la direttiva *#include* e specificare la clausola *using namespace std*.

File di libreria	Contenuto
<algorithm>	algoritmi generici (come *sort()*, *for_each()*, *merge()*, *copy()*, *min_element()*, *max_element()*, *find()* e *count()*) per operare sui contenitori;
<deque>	contenitore *deque*;
<functional>	funzioni e oggetti di appoggio;
<iterator>	iteratori;
<list>	contenitore *list*;
<map>	contenitori *map* e *multimap*;
<memory>	array di dimensioni variabili; è anche chiamato coda bifronte;
<numeric>	contenitori *map* e *multimap*;
<queue>	contenitore *queue*;
<set>	contenitori *set* e *multiset*;
<stack>	contenitore *stack*;
<utility>	definizione di *pair* (coppia chiave-valore) e degli operatori relazionali;
<vector>	contenitore *vector*.

Caratteristiche dei contenitori di STL

Ogni classe contenitore è indipendente, cioè non esiste nessuna classe base da cui derivano le altre; questo significa che non ci sono metodi virtuali e che ogni classe definisce i suoi metodi, usando però gli stessi nomi per funzionalità analoghe.

Un singolo contenitore può definire proprie funzionalità (metodi e overloading di operatori) oppure inibirne alcune di comuni, perché non adatte.

Dato che le collezioni sono classi generiche è possibile creare una collezione specificando il tipo di oggetti che deve contenere. Cercare di aggiungere alla collezione un oggetto di tipo diverso causa un errore di compilazione.

Gli iteratori

Un iteratore permette di accedere sequenzialmente agli elementi di un contenitore senza dover conoscere la rappresentazione del contenitore stesso.

Per uno stesso contenitore si possono definire più iteratori che permettono di attraversare il contenitore con modalità diverse.
Si possono effettuare anche più attraversamenti contemporanei su una stessa struttura.

In C++ gli iteratori sono visti come una generalizzazione del concetto di puntatore; possiedono infatti alcune proprietà dei puntatori e in più conservano gli stessi operatori.

Nota

Diversamente dai puntatori gli iteratori non dipendono dal tipo dell'elemento del contenitore.

Gli iteratori sono divisi in tipi in base al modo di scorrere gli elementi del contenitore:

`iterator`	per scandire gli elementi del contenitore, permettendo di modificarli;
`const_iterator`	per scandire gli elementi del contenitore, senza permettere modifiche;
`reverse_iterator`	per scandire gli elementi del contenitore in ordine inverso, permettendo di modificarli;
`const_reverse_iterator`	in grado di scandire ma non modificare gli elementi del contenitore in ordine inverso.

Per tutti i tipi di iteratore è definito **l'operatore *** per accedere al contenuto dell'elemento puntato; è necessario, con apposita notazione, associare l'iteratore scelto al contenitore.

Un contenitore possiede uno o più tipi di iteratore secondo le proprie specifiche e può ridefinire (sovraccaricando i relativi operatori) opportune operazioni su di essi.

L'associazione di un oggetto iteratore ad un contenitore avviene con la sintassi

```
contenitore<Tipo>::tipoIteratore nomeIteratore
```

quando *Tipo* non è un tipo parametrico e con la sintassi

```
typename contenitore<Tipo>::tipoIteratore nomeIteratore
```

quando *Tipo* è un tipo parametrico.

Classificazione funzionale dei contenitori

Una classe contenitore possiede:

- il costruttore di copia;
- il sovraccarico dell'operatore = (assegnazione);
- il distruttore;
- i tipi *iterator* e *const_iterator*;
- i metodi:

`max_size()`	restituisce il numero massimo di elementi memorizzabili;
`size()`	restituisce il numero di elementi memorizzati;
`empty()`	restituisce *true* se non vi sono elementi;
`begin()`	restituisce l'iteratore che punta al primo elemento;
`end()`	restituisce l'iteratore che punta alla posizione successiva a quella dell'ultimo elemento; questo iteratore è chiamato anche **past-the-end**.

Contenitore Sequenza

Un contenitore è detto **contenitore sequenza** quando è in grado di inserire o rimuovere elementi in qualsiasi posizione specificata da un iteratore.

Un contenitore sequenza possiede inoltre:

- i metodi:

`insert(posIter, elemDaInserire)`
: inserisce in una posizione specificata dall'iteratore *posIter* l'elemento *elemDaInserire*; restituisce un iteratore all'elemento inserito;

`insert(posIter, n, elemDaInserire)`
: inserisce, a partire dalla posizione specificata dall'iteratore *posIter*, *n* copie dell'elemento *elemDaInserire*;

`erase(posIter)`
: rimuove l'elemento specificato dall'iteratore *posIter*; restituisce un iteratore all'elemento successivo a quello rimosso;

`erase(posIterInizio, posIterFine)`
: rimuove tutti gli elementi dell'intervallo specificato degli iteratori *posIterInizio* e *posIterFine*; restituisce un iteratore all'elemento successivo all'ultimo rimosso;

`clear()` rimuove tutti gli elementi;

- l'overloading degli operatori:

++ (prefisso e postfisso) per incrementare di una posizione l'iteratore, facendolo puntare all'elemento successivo;

-- (prefisso e postfisso) per decrementare di una posizione l'iteratore, facendolo puntare all'elemento precedente;

+= e -= per incrementare/decrementare di un certo numero di posizioni l'iteratore;

<, <=, >, >= in analogia con l'aritmetica dei puntatori.

Contenitore Sequenza ad accesso casuale

Un contenitore sequenza è detto **ad accesso casuale** quando permette di accedere direttamente ad un elemento attraverso un indice, cioè quando sovraccarica l'operatore [].

Contenitore Associativo

Un contenitore è detto **contenitore associativo** quando è possibile accedere agli elementi tramite delle **chiavi**. La chiave può coincidere o meno con l'elemento. Alcuni contenitori associativi pretendono l'unicità della chiave altri consentono più elementi con la stessa chiave.

Un contenitore associativo possiede oltre ai metodi *erase()* e *clear()* del contenitore sequenza, anche i metodi:

`erase(valoreChiave)` rimuove l'elemento con chiave *valoreChiave*; restituisce il numero di elementi rimossi;

`count(valoreChiave)` restituisce il numero di elementi con chiave *valoreChiave*;

`find(valoreChiave)` restituisce l'iteratore all'elemento con chiave *valoreChiave*; se l'elemento non c'è restituisce l'iteratore past-the-end.

Se il contenitore associativo è ordinato rispetto alle chiavi allora sovraccarica l'operatore <.

Adattatori

Un **adattatore** è un caso particolare di un contenitore sequenza o associativo, che riduce le funzionalità del contenitore di partenza limitando le operazioni possibili; non possiede iteratori.

Descrizione delle principali classi contenitori

`vector`	array di dimensioni variabili; contenitore sequenza ad accesso casuale;
`stack`	adattatore pila (accesso LIFO);
`queue`	adattatore coda (accesso FIFO);
`priority queue`	adattatore coda ordinata (in modo decrescente); l'inserimento e la rimozione di elementi avvengono in modo da conservare l'ordinamento;
`list`	lista concatenata bidirezionale; contenitore sequenza;

`slist`	lista concatenata monodirezionale; contenitore sequenza (definita nel file *<slist>* estensione di STL);
`deque`	array di dimensioni variabili; contenitore sequenza ad accesso casuale; differisce da *vector* solo per il modo (interno) di inserire o rimuovere il primo elemento;
`set`	contenitore associativo ordinato che non consente elementi duplicati; rappresenta il concetto di insieme matematico;
`multiset`	contenitore associativo ordinato che consente elementi duplicati;
`map`	contenitore associativo ordinato di coppie chiave-valore; non consente elementi con la stessa chiave;
`multimap`	contenitore associativo ordinato di coppie chiave-valore; consente elementi con la stessa chiave.

CONTENITORE VECTOR

Il contenitore ***vector*** è un contenitore sequenza che possiede gli iteratori *iterator*, *const_iterator*, *reverse_iterator* e *const_reverse_iterator*.

Costruttori e metodi di *vector* sono stati descritti nel paragrafo C++ 13.1. **Nota**

ESEMPIO 14.1

Modifica dell'esempio 13.7. Uso degli iteratori di tipo *const_iterator* e *reverse_iterator* per stampare gli elementi della pila in ordine di inserimento o in ordine di estrazione.

```
//PilaVector.h
#include <vector>
using namespace std;

template <class E> class PilaVector {
    private:
        vector<E> pila;
    public:
        void push(E);
        E top(void) const;
        E pop(void);
        int cima(void) const;
        bool vuota(void) const;
        void stampaOrdIns(void) const;
        void stampaOrdEstr(void);
};//PilaVector

//altri metodi

template <class E> void PilaVector<E>::stampaOrdIns(void) const{
    typename vector<E>::const_iterator it;
    cout << "Elementi: ";
    for(it = pila.begin(); it != pila.end(); it++)
        cout << *it << "  ";
    cout << endl;
}//stampaOrdIns
```

```
template <class E> void PilaVector<E>::stampaOrdEstr(void){
    typename vector<E>::reverse_iterator it;
    cout << "Elementi: ";
    for(it = pila.rbegin(); it != pila.rend(); it++)
        cout << *it << "  ";
    cout << endl;
}//stampaOrdEstr
```

```
//ProvaPilaVector.cpp
#include <iostream>
#include "PilaVector.h"
using namespace std;
int main(void){
    PilaVector<string> pila;
    string parola;
    cout << "Inserisci una parola alla volta, . per terminare\n";
    do{ cin >> parola;
        pila.push(parola);
    }while(parola != ".");
    pila.stampaOrdIns();
    pila.stampaOrdEstr();
    fflush(stdin);
    getchar();
    return 0;
}//main
```

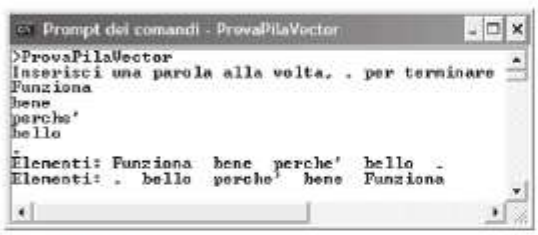


Figura 14.1 *Esecuzione dell'esempio*

Alla classe *PilaVector* sono stati aggiunti i metodi *stampaOrdIns()* e *stampaOrdEstr()*. Poiché *reverse_iterator* è un tipo di iteratore che permette la modifica degli elementi scanditi, il metodo *stampaOrdEstr()* non può essere un metodo costante.

C++ 14.3 Contenitore stack

Il contenitore ***stack*** è un adattatore predisposto alla gestione LIFO.
Possiede i metodi:

`push(el)`	inserisce l'elemento specificato nello stack;
`pop()`	rimuove (senza restituirlo) l'elemento sulla cima dello stack;
`top()`	restituisce (senza rimuoverlo) l'elemento sulla cima dello stack;
`empty()`	restituisce *true* se lo stack è vuoto.

Esempio 14.2

Classe *ProvaStack* che permette di inserire una serie di parole e le stampa in ordine inverso.

```
public class ProvaStack {
#include <iostream>
#include <stack>
using namespace std;
int main(void){
    stack<string> pila;
    string parola;
```

```
    cout << "Inserisci una parola alla volta, . per terminare\n";
    do{ cin >> parola;
        pila.push(parola);
    }while(parola != ".");
    cout << "Le parole inserite, dall'ultima alla prima, sono:\n";
    while(!pila.empty()){
        parola = pila.top();
        pila.pop();
        cout << parola << endl;
    }//while
    fflush(stdin);
    getchar();
    return 0;
}//main
```

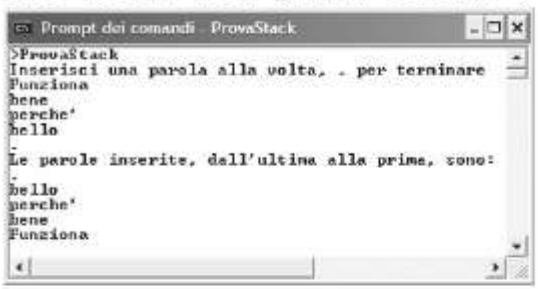


Figura 14.2 *Esecuzione dell'esempio*

CONTENITORE LIST

C++ 14.4

Il contenitore ***list*** è un contenitore sequenza e rappresenta una lista concatenata bidirezionale; *list* non sovraccarica l'operatore [], pertanto per raggiungere un elemento è necessario scorrere la lista; inoltre non sovraccarica gli operatori += e -=, quindi non è possibile accedere ad un elemento anche se si conosce la posizione relativa rispetto ad un iteratore (senza usare la funzione generica *advance()* definita nel file *<iterator>*).

La classe *list* possiede oltre ai metodi di *vector* anche

- i costruttori:

`list(n)`	crea una lista di *n* elementi;
`list(n, el)`	crea una lista di *n* elementi tutti uguali a *el*;
`list(primoIter, secIter)`	crea una lista prendendo gli elementi da un altro contenitore, nell'intervallo individuato dagli iteratori *primoIter* e *secIter*;

- i metodi:

`sort()`	ordina la lista; questo è possibile solo se per il tipo di elemento della lista è definito (o è stato sovraccaricato) l'operatore <;
`push_front(el)`	inserisce l'elemento specificato come primo elemento;
`pop_front()`	rimuove il primo elemento;
`splice(it, altraL)`	inserisce nella lista su cui è invocato prima dell'elemento individuato dall'iteratore *it* tutti gli elementi della lista *altraL*;
`reverse()`	inverte l'ordine degli elementi;
`merge(altraLista)`	fonde in modo ordinato (merge) la lista su cui è invocato con la lista *altraLista*; questo è possibile solo se le liste hanno lo stesso tipo di elemento e per esso è definito (o è stato sovraccaricato) l'operatore <;
`swap(altraLista)`	scambia gli elementi della lista su cui è invocato con la lista *altraLista*; questo è possibile solo se le liste hanno lo stesso tipo di elemento.

ESEMPIO 14.3

Classe *ListaConcatenata* che offre vari metodi per gestire l'inserimento di parole digitate dall'utente in una lista concatenata.

Nota Per individuare la posizione di un elemento all'interno della lista concatenata è stata utilizzata la funzione generica *find()* definita nel file di libreria *<algorithm>*.

```
//ListaConcatenata.h
#include <string>
#include <list>
using namespace std;

class ListaConcatenata {
    private:
        list<string> lista;
        friend string leggi(string);
    public:
        void crea(void);
        void inserisciInTesta(void);
        void inserisciInCoda(void);
        void inserisciDopo(void);
        void cerca(void) const;
        void stampa(void) const;
};//ListaConcatenata
```

```
//ListaConcatenata.cpp
#include <iostream>
#include <algorithm>
#include "ListaConcatenata.h"
using namespace std;

void ListaConcatenata::crea(void){
    string parola;
    cout << "Inserisci una parola alla volta"
         << " (. per terminare)\n";
    do {
        cin >> parola;
        if (parola != ".")
            lista.push_back(parola);
    }while(parola != ".");
}//crea

string leggi(string s){
    string parola;
    cout << s;
    cin >> parola;
    return parola;
}//leggi

void ListaConcatenata::inserisciInTesta(void){
    string parola = leggi("Inserisci nuovo valore\n");
    lista.push_front(parola);
}//inserisciInTesta

void ListaConcatenata::inserisciInCoda(void){
    string parola = leggi("Inserisci nuovo valore\n");
    lista.push_back(parola);
}//inserisciInCoda

void ListaConcatenata::cerca(void) const{
    string parola = leggi("Inserisci il valore da cercare\n");
    list<string>::const_iterator it = lista.begin();
```

Figura 14.3 *Esecuzione dell'esempio*

```
    bool trovato = false;
    int pos = 0;
    while((it != lista.end()) && (!trovato)){
        if( *it == parola)
            trovato = true;
        it++;
        pos++;
    }//while
    if(trovato)
        cout << parola << " posizione " << pos << endl;
    else
        cout << "non trovato\n";
}//cerca

void ListaConcatenata::inserisciDopo(void){
    string parola = leggi("Inserisci nuovo valore\n");
    string valore = leggi("Dopo quale valore?\n");
    list<string>::iterator it = find(lista.begin(), lista.end(),
valore);
    lista.insert(++it, parola);
}//inserisciDopo

void ListaConcatenata::stampa(void) const{
   list<string>::const_iterator it;
    for(it = lista.begin(); it != lista.end(); it++)
       cout << *it << endl;
}//stampa
```

```
//ProvaListaConcatenata.cpp
#include <iostream>
#include "ListaConcatenata.h"
using namespace std;

int main(void){
    ListaConcatenata listaC;
   ...
}//main
```

GLI ALGORITMI DI STL

C++ 14.5

La libreria STL offre anche molte funzioni template per operare sui contenitori.

Nel file di libreria ***<iterator>*** sono definite due funzioni equivalenti al sovraccarico degli operatori *+=* e *-=* e alla differenza tra iteratori, da usare nei contenitori sequenza privi dell'accesso casuale.

`void` **`advance`**`(TipoIteratore& it, int n)`
simula l'operazione *it += n* (anche con *n* negativo);

`int` **`distance`**`(TipoIteratore itPrimo, TipoIteratore itSec)`
simula l'operazione *itSec - itPrimo*.

Molte funzioni template operanti sui contenitori sono definite nel file di libreria ***<algorithm>*** e per questo sono anche chiamate **algoritmi**. Solitamente agiscono sul contenitore attraverso

due iteratori, associati al contenitore stesso, che individuano un intervallo chiuso a sinistra e aperto a destra: appartengono all'intervallo gli elementi la cui posizione è uguale o maggiore di quella del primo iteratore e strettamente minore di quella del secondo iteratore. Con il termine sequenza si indicano tutti gli elementi dell'intervallo.

Nota

Per indicare l'intervallo contenente tutti gli elementi del contenitore si usano gli iteratori restituiti dalle chiamate ai metodi *begin()* e *end()* sull'oggetto contenitore.

Nota

Al posto degli iteratori possono essere usati i puntatori; in questo modo gli algoritmi possono essere applicati agli array.

for_each`(itPrimo, itSec, nomeFunzione)`
: permette l'esecuzione di una funzione su ciascun elemento di una sequenza; come terzo parametro è possibile specificare anche un oggetto-funzione.

find`(itPrimo, itSec, elemento)`
: permette la ricerca di un elemento all'interno di una sequenza; restituisce un iteratore alla posizione della prima (o unica) occorrenza di *elemento* oppure l'iteratore *itSec*;

count`(itPrimo, itSec, elemento)`
: restituisce il numero di occorrenze di *elemento* nel contenitore;

binary_search`(itPrimo, itSec, elemento)`
: permette di stabilire se un elemento è presente o meno in una sequenza **ordinata**: restituisce *true* se *elemento* è presente, *false* altrimenti; rispetto agli algoritmi *find()* e *count()*, *binary_search()* è più efficiente;

adjacent_find`(itPrimo, itSec)`
: controlla se in una sequenza sono presenti due elementi uguali in posizioni contigue: restituisce un iteratore alla posizione del primo elemento uguale al successivo oppure l'iteratore *itSec*;

equal`(itPrimoSeqUno, itSecSeqUno, itPrimoSeqDue, itSecSeqDue)`
: permette il confronto di due sequenze: restituisce *true* se gli elementi della prima sequenza sono ordinatamente uguali a quelli della seconda sequenza, *false* altrimenti; i tipi degli elementi delle due sequenze possono essere di tipo diverso ma compatibile; le due sequenze possono risiedere in contenitori distinti.

copy`(itPrimo, itSec, itAltraSeq)`
: permette la copia di una sequenza in un'altra a partire dalla posizione specificata; solitamente l'iteratore *itAltraSeq* appartiene ad un contenitore diverso da quello della sequenza; può essere anche un iteratore per un flusso; si può usare anche per copiare due array con una sola istruzione:

```
double a1[Max], a2[Max];
//istruzioni di caricamento dell'array a1
copy(a1, a1+Max, a2)
```

unique`(itPrimo, itSec)`
: agisce su una sequenza ordinata spostando nelle posizioni finali gli elementi uguali successivi: restituisce un iteratore alla posizione del primo elemento uguale spostato; è necessario usare un metodo *erase()* del contenitore a cui appartiene la sequenza per eliminare gli elementi spostati;

replace`(itPrimo, itSec, vecchioEl, nuovoEl)`
: permette in una sequenza la sostituzione di tutte le occorrenze di un elemento con un altro;

remove`(itPrimo, itSec, elemento)`
: permette in una sequenza la rimozione di tutte le occorrenze di un elemento:

restituisce un iteratore alla posizione dell'ultima occorrenza dell'elemento rimosso oppure l'iteratore *itSec*;

sort`(itPrimo, itSec)`
: permette l'ordinamento in ordine crescente degli elementi di una sequenza; il tipo degli elementi della sequenza, se è una classe, deve possedere un sovraccarico dell'operatore <;

reverse`(itPrimo, itSec)`
: agisce su una sequenza invertendo l'ordine degli elementi:

merge`(itPSUno, itSSUno, itPSDue, itSSDue, itFusione)`
: permette la fusione (merge) di due sequenze ordinate in un'altra ancora ordinata; restituisce un iteratore alla fine della sequenza fusione; le tre sequenze coinvolte possono risiedere in tre contenitori distinti;

min_element`(itPrimo, itSec)`
max_element`(itPrimo, itSec)`
: permettono di ottenere rispettivamente l'elemento minimo e massimo di una sequenza; restituiscono un iteratore alla posizione dell'elemento minimo/massimo.

VERIFICA

QUESTIONARIO

1. Che cos'è la libreria STL?
2. Quali sono le principali classi relative alle collezioni?
3. A che cosa servono gli iteratori?
4. Cos'è l'iteratore *past-the-end*?
5. Che cosa rappresenta la classe *list*?
6. Come si possono attraversare gli elementi di un oggetto *list*?
7. Quali sono le differenze significative tra le classi *list* e *vector*?

TEST

1 ..... *Individuare la risposta esatta.*

1) Il metodo *rbegin()* di un contenitore restituisce:
   - ☐ un iteratore di tipo *iterator* che individua il primo elemento del contenitore;
   - ☐ un iteratore di tipo *reverse_iterator* che individua l'ultimo elemento del contenitore;
   - ☐ un iteratore di tipo *reverse_iterator* che individua il primo elemento del contenitore.

2) Per inserire un elemento in un oggetto di tipo *stack* si usa il metodo:
   - ☐ *push()*;
   - ☐ *top()*;
   - ☐ *pop()*.

3) La classe *list* non sovraccarica:
   - ☐ l'operatore <;
   - ☐ l'operatore ==;
   - ☐ l'operatore *[]*.

Esercizi

1 ..... Realizzare una implementazione di una pila e di una coda usando una *list*.

2 ..... Scrivere un programma di collaudo per un oggetto della classe *queue*.

3 ..... Esaminare il file di libreria *<set>* per reperire la documentazione relativa alla classe *set* e il file di libreria *<algorithm>* per reperire la documentazione relativa alla funzione *sort()*.

4 ..... Aggiungere alla classe *ListaConcatenata* dell'esempio 14.3 un metodo per ordinare le parole della lista sfruttando il metodo *sort()* della classe *list*.

PROPOSTE DI LAVORO

1 ..... Creare un array di numeri e da questo creare un oggetto *vector*. Trovare il massimo e il minimo dopo aver ordinato il vettore con la funzione *sort()* (file di libreria *<algorithm>*).

2 ..... Usare uno *stack* per controllare se una stringa è palindroma: inserire ogni carattere nello stack e poi controllare con i caratteri della stringa.

3 ..... Inserire in una stringa un'espressione con parentesi tonde, quadre e graffe. Utilizzare un oggetto *stack* per controllare se le parentesi sono usate correttamente: inserire nello stack ogni parentesi aperta e ad ogni parentesi chiusa, controllare se la parentesi sulla cima dello stack è dello stesso tipo e in tal caso estrarla. Se lo stack rimane vuoto l'espressione è corretta.

Realizzare le classi

4 ..... ***Poligono*** che memorizza i vertici di un poligono come oggetti *Punto* in:
- uno *stack*,
- una *list*.

5 ..... ***Agenda*** di appuntamenti come *list* di *Appuntamento* (data, ora e descrizione).
La lista deve essere ordinata. Inserire un appuntamento nel giusto ordine e non aggiungerlo se è in conflitto con altri appuntamenti.

6 ..... ***Mazzo*** di carte come *list* di *Carta* (seme e valore).
Si deve poter mescolare il mazzo, estrarre la prima carta e rimettere una carta in fondo al mazzo.

7 ..... ***Classe*** (di una scuola)
con attributi:
- nome,
- numero alunni,
- *list* di oggetti *Studente*,

e con metodi per:
- aggiungere o togliere uno studente,
- elencare gli studenti della classe.

8 ..... ***Scuola***
con attributi:
- nome,
- numero delle classi,
- *list* di oggetti *Classe*,

e con metodi per:
- aggiungere o togliere una classe,
- elencare le classi,
- elencare gli studenti della scuola.

GESTIONE DELLE ECCEZIONI

15

PREREQUISITI

- Saper utilizzare una classe
- Conoscere i meccanismi dell'ereditarietà

OBIETTIVI

CONOSCENZE

- Classi relative alle eccezioni
- Istruzione *throw*
- Blocco *try catch*
- Clausola *throw*

ABILITÀ/CAPACITÀ

- Sollevare eccezioni
- Gestire eccezioni
- Definire nuove classi di eccezioni

Riassumendo

Un'**eccezione** può essere:

- un valore di un tipo predefinito; il tipo può essere indicato nella dichiarazione della funzione e viene utilizzato per riconoscere l'eccezione stessa;
- un oggetto che viene creato e lanciato nel codice di una funzione o di un metodo con l'istruzione *throw*; il tipo dell'oggetto deve essere una classe derivata della classe ***exception*** definita nel file di libreria ***<exception>*** (da includere).

Le funzioni o i metodi che generano eccezioni possono indicarle nella loro dichiarazione con la clausola *throw(listaEccezioni)*.

È possibile impedire che una funzione o un metodo lancino eccezioni indicando la clausola *throw()* senza nulla tra parentesi.

Il modo più semplice per gestire le eccezioni è intercettarle con l'istruzione ***try catch***.

C++ 15.1

Gerarchia delle eccezioni

Tutte le eccezioni oggetto sono sottoclassi della classe ***exception***.

```
exception
    bad_alloc
    bad_cast
    bad_exception
    bad_typeid
    logic_error
        domain_error
        invalid_argument
        length_error
        out_of_range
    runtime_error
        overflow_error
        range_error
        underflow_error
    ios_base::failure
```

Ogni eccezione è definita in un file di libreria:

<exception>	*exception*, *bad_exception*
<new>	*bad_alloc*
<stdexcept>	*logic_error*, *runtime_error* e tutte le loro sottoclassi
<ios_base.h>	*ios_base::failure* (classe interna)
<typeinfo>	*bad_cast*, *bad_typeid*

Nota

Il file di libreria *<ios_base.h>* è incluso automaticamente in tutti i file di libreria della gerarchia stream, quindi non è necessario includerlo direttamente con la direttiva *#include*.

Significato delle principali eccezioni:

`bad_exception`	definisce l'eccezione che viene generata da un metodo che lancia un'eccezione non presente nell'elenco della clausola *throw*;

`logic_error`	e le sue sottoclassi descrivono errori derivanti dalla non osservanza delle precondizioni sui metodi e sul contenuto delle variabili;
`runtime_error`	e le sue sottoclassi descrivono errori che possono verificarsi in qualsiasi punto del programma;
`bad_alloc`	è generata dall'operatore ***new*** quando non riesce ad allocare la memoria richiesta;
`ios_base::failure`	descrive gli errori derivanti da operazioni di I/O.

La classe ***exception*** non viene utilizzata direttamente, ma contiene la dichiarazione del metodo virtuale ***what()***.

`virtual char* what() throw()`
restituisce la descrizione associata all'eccezione al momento della creazione dell'oggetto.

Una delle operazioni più comuni in un gestore di eccezioni è la stampa delle informazioni sull'eccezione che si può ottenere con:

```
cout << eccezione.what();
```

Le classi derivate da *exception* possiedono un attributo *string* per la descrizione dell'eccezione, inizializzato tramite il costruttore, che riceve come parametro la stringa relativa. La descrizione viene restituita dal metodo virtuale *what()* sovrascritto.

Sollevare eccezioni predefinite

C++ 15.2

Il programmatore può sollevare una eccezione oggetto con l'istruzione ***throw*** che riceve come parametro un oggetto di una delle classi derivate da ***exception***.
Bisogna creare un oggetto di una classe eccezione adeguata e lanciarlo:

```
throw eccezionePredefinita("descrizione eccezione");
```

Molti metodi predefiniti lanciano eccezioni in questo modo; la lista delle eccezioni che possono lanciare è indicata nella clausola ***throw***.

Le eccezioni possono essere lanciate da qualsiasi metodo o funzione, anche dai costruttori.

Quando si richiama un metodo che lancia un'eccezione si devono gestire tutte le eccezioni che vengono lanciate.

Il costrutto try catch

C++ 15.3

Per **intercettare** una eccezione lanciata da un metodo basta inserire la chiamata del metodo nel costrutto ***try catch***; l'istruzione *try catch* installa un gestore di eccezioni.

La sintassi di un blocco *try catch* con le eccezioni oggetto è:

```
try {
    // istruzioni che possono generare eccezioni
}
catch (eccezione& e) {
    // gestione dell'eccezione
}
```

Le istruzioni del blocco ***try*** vengono eseguite una alla volta; se tutto il blocco *try* viene eseguito correttamente la clausola *catch* non viene eseguita; se però un'istruzione solleva un'eccezione le altre istruzioni del blocco *try* non vengono eseguite e si passa ad eseguire la clausola *catch*.

La clausola ***catch*** viene eseguita solo se l'eccezione sollevata è compatibile con il tipo del parametro, cioè dello stesso tipo o di un tipo derivato.
L'eccezione viene passata come parametro alla clausola *catch*; la clausola *catch* può utilizzare i metodi della classe dell'eccezione (di solito il metodo *what()* ereditato da *exception*).

Ci possono essere più clausole *catch*, una per ogni tipo di eccezione che il blocco *try* può gestire.

```
try {
    // istruzioni che possono generare eccezioni
}
catch (UnaClasseException& e) {
    // gestione dell'eccezione di tipo UnaClasseException
}
catch (AltraClasseException& e) {
    // gestione dell'eccezione di tipo AltraClasseException
}
```

Quando si verifica una eccezione le clausole *catch* vengono esaminate nell'ordine in cui si presentano provando se una di queste può intercettare il tipo dell'eccezione, cioè se nella clausola *catch* è indicata la classe dell'eccezione o una sua superclasse.
Se si trova una clausola *catch* di questo tipo si esegue il suo blocco di istruzioni, dopo aver passato come parametro un riferimento all'oggetto eccezione. Le clausole *catch* successive non vengono eseguite.

Non ha senso scrivere due clausole *catch* di cui la prima gestisca una superclasse di eccezioni rispetto alla seconda: la seconda non verrebbe mai eseguita.

Se una clausola *catch* lancia a sua volta una eccezione, le clausole *catch* non vengono riesaminate. Queste eccezioni possono essere gestite da un blocco *try catch* che racchiuda quello in cui si trovano le clausole *catch* che hanno lanciato l'eccezione.

Se non si trova nessuna clausola *catch* appropriata l'eccezione viene propagata lungo la catena dei metodi che ha generato la chiamata; potrebbe esserci un blocco *try catch* più esterno che può gestire l'eccezione.
L'eccezione cioè può essere gestita nel codice che richiama il metodo che la lancia o più indietro nello stack delle chiamate. Le eccezioni permettono di comunicare informazioni lungo una catena di metodi finché uno di questi può gestirle.

Dopo le clausole *catch* può esserci una clausola ***catch****(...)* per catturare un qualsiasi tipo di eccezione. Le istruzioni della clausola *catch(...)* vengono eseguite, se l'eccezione non è stata gestita da nessuna delle clausole *catch* precedenti.

Esempio 15.1

Lancio dell'eccezione predefinita *domain_error* se il valore inserito non è nel dominio di definizione del tipo della variabile; nel costruttore dell'eccezione si può specificare il messaggio desiderato.

```
#include <iostream>
#include <stdexcept>
using namespace std;

int leggi(void) throw(domain_error){
```

```
    int n;
    cout << "Inserisci un numero positivo o nullo\n";
    cin >> n;
    if(n < 0)
        throw domain_error("Introdotto numero negativo!!!");
    return n;
}//leggi

int main(void){
    int numero;
    try{
        numero = leggi();
    }catch(domain_error& e){
        cerr << e.what() << endl;
        return 1;
    }//catch
    cout << "Numero letto: " << numero << endl;
    return 0;
}//main
```

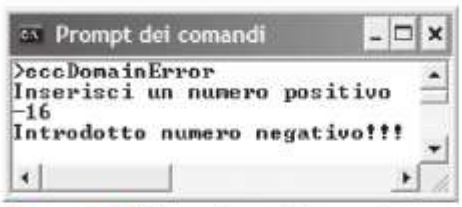


Figura 15.1 *Esecuzione dell'esempio*

ESEMPIO 15.2

Gestione di più eccezioni predefinite con un blocco *try catch* avente due clausole *catch*. La funzione *leggi()* è quella dell'esempio 15.1.

```
#include <iostream>
#include <stdexcept>
using namespace std;

int leggi(void) throw(domain_error){
    int n;
    cout << "Inserisci un numero positivo o nullo\n";
    cin >> n;
    if(n < 0)
        throw domain_error("Introdotto numero negativo!!!");
    return n;
}}//leggi

double reciproco(int n) throw(out_of_range){
    if(n==0)
        throw out_of_range("Divisione per 0");
    return 1.0/n;
}//reciproco

int main(void){
    int numero;
    double r;
    try{
        numero = leggi();
        r = reciproco(numero);
    }catch(domain_error& e){
        cerr << e.what() << endl;
        return 1;
```

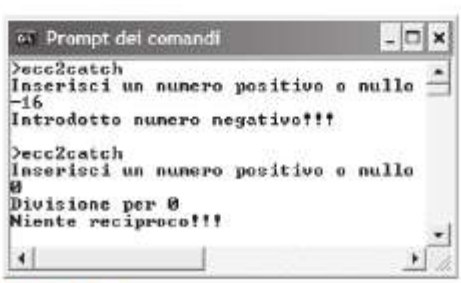


Figura 15.2 *Esecuzione dell'esempio*

```
    }catch(out_of_range& e){
        cerr << e.what() << endl;
        cout << "Niente reciproco!!!";
        return 2;
    }//catch
    cout << "Il reciproco di " << numero
         << " e' " << r << endl;
    return 0;
}//main
```

Invocazione di metodi che lanciano eccezioni

Per **gestire un'eccezione** un metodo può:

- catturare e gestire l'eccezione con un blocco ***try catch***;
- non intercettare l'eccezione ma elencarla nella propria clausola ***throw***, lasciando così che si propaghi lungo la catena dei metodi che ha generato la chiamata; in questo modo si può avere una classe specializzata per la gestione delle eccezioni; un'eccezione può verificarsi in più punti ma essere gestita sempre nello stesso punto;
- catturare l'eccezione e, nella clausola *catch*, **lanciare un'altra eccezione** (dichiarata nella propria clausola *throw*).

C++

15.4 Creazione di nuove eccezioni

Si possono definire delle **eccezioni personalizzate** creando nuove classi di eccezioni.
È utile definire nuove classi di eccezioni perché le eccezioni vengono catturate in una clausola *catch* in base al tipo.

Per creare nuove classi di eccezioni è possibile creare una sottoclasse di ***exception***, ma bisogna definire un attributo *string* (per la descrizione dell'eccezione) e un costruttore parametrico per l'inizializzazione dell'attributo e ridefinire il metodo virtuale *what()*.

È preferibile creare la classe eccezione come derivata di *logic_error* oppure di *runtime_error* perché tali classi (sottoclassi di *exception*) forniscono già l'attributo, il costruttore e l'implementazione del metodo *what()*; basta definire il costruttore della classe eccezione personalizzata richiamando il costruttore della superclasse.

```
class NuovaEcc : public logic_error {
    public:
        NuovaEcc(const string&);
}; //NuovaEcc

NuovaEcc::NuovaEcc(const string& m) : logic_error(m){}//NuovaEcc
```

Per lanciare l'eccezione si crea un oggetto della nuova classe e lo si lancia con l'istruzione *throw*.

```
void metodo throw(NuovaEcc) {
    ...
    throw NuovaEcc("Messaggio di errore");
    ...
}//metodo
```

Esempio 15.3

Creazione di una classe apposita per l'eccezione di divisione per zero.

```
//DivisionePerZero.h
#include <string>
#include <stdexcept>
using namespace std;

class DivisionePerZero : public runtime_error {
    public:
        DivisionePerZero(const string&);
}; //DivisionePerZero

DivisionePerZero::DivisionePerZero(const string& m) :
runtime_error(m){}//DivisionePerZero
- - - - - - - - - - - - - - - -
//ProvaDivPerZero.cpp
#include <iostream>
#include "DivisionePerZero.h"
using namespace std;

int leggi(void) throw(domain_error){
    int n;
    cout << "Inserisci un numero positivo o nullo\n";
    cin >> n;
    if(n < 0)
        throw domain_error("Introdotto numero negativo!!!");
    return n;
}//leggi

double reciproco(int n) throw(DivisionePerZero){
    if(n==0)
        throw DivisionePerZero("Divisione per 0");
    return 1.0/n;
}//reciproco

int main(void){
    int numero;
    double r;
    try{
        numero = leggi();
        r = reciproco(numero);
    }catch(domain_error& e){
        cerr << e.what();
        return 1;
    }catch(DivisionePerZero& e){
        cerr << e.what() << endl;
        cout << "Niente reciproco!!!";
        return 2;
    }//catch
    cout << "Il reciproco di " << numero
         << " e' " << r << endl;
    return 0;
}//main
```

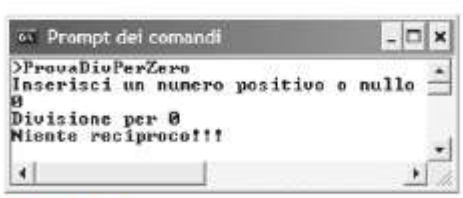


Figura 15.3 *Esecuzione dell'esempio*

VERIFICA

QUESTIONARIO

1. Cosa sono le eccezioni valore e le eccezioni oggetto?
2. Da quale classe derivano tutte le eccezioni oggetto e qual è il metodo principale di tale classe?
3. Come vengono sollevate le eccezioni?
4. Che cosa fa l'istruzione *try catch* e quali sono i suoi formati ?
5. A che cosa serve la clausola *throw*?
6. A che cosa serve la clausola *catch(...)*?
7. Come si può creare una nuova classe di eccezione?
8. A che cosa serve creare nuove classi di eccezioni?

TEST

1 ..... *Indicare se le seguenti affermazioni sono vere o false.*

V	F	
☐	☐	1) Le eccezioni possono essere sottoclassi di *exception*.
☐	☐	2) Le eccezioni sono sempre oggetti.
☐	☐	3) Le eccezioni possono essere sottoclassi di *runtime_error*.
☐	☐	4) Le eccezioni sono sempre definite in classi.
☐	☐	5) Una eccezione personalizzata non deve essere lanciata con l'istruzione *throw*.

PROPOSTE DI LAVORO

Nota È possibile svolgere ogni proposta di lavoro con eccezioni valore o con eccezioni oggetto.

1 ..... Scrivere una funzione che riceva come parametri due numeri interi e lanci due diverse eccezioni, una se il primo parametro è 0, l'altra se il primo parametro è minore del secondo. Provare la funzione inserendo la chiamata in un ciclo e contare quanti errori del primo tipo e quanti del secondo si sono verificati.

2 ..... Definire una classe *Data* con anno, mese e giorno che lanci apposite eccezioni se si cerca di creare un oggetto con valori non corretti.
Scrivere un programma che crei vari oggetti *Data* permettendo in caso di errore, di correggere e continuare.

3 ..... Scrivere una classe *Orario* con ora, minuti e secondi che lanci apposite eccezioni se si cerca di creare un oggetto con valori non corretti.

4 ..... Modificare la classe *Data* della proposta di lavoro 2 in modo che abbia come attributo un oggetto della classe *Orario*, definita come nella proposta di lavoro 3.
Scrivere un'applicazione che crei vari oggetti *Data* e tenga conto di quanti errori complessivamente sono stati commessi nella creazione di oggetti *Data* e *Orario*.

GESTIONE DELL'INPUT/OUTPUT 16

PREREQUISITI

- Saper utilizzare una classe
- Conoscere i meccanismi dell'ereditarietà

OBIETTIVI

CONOSCENZE

- Classi per la gestione dell'I/O, in particolare le classi:
  ios
  ostream, *istream* e *iostream*
  ofstream, *ifstream* e *fstream*
  ostringstream, *istringstream* e *stringstream*
- Iteratori per i flussi

ABILITÀ/CAPACITÀ

- Stampare dati
- Leggere l'input dalla tastiera
- Scrivere e leggere file di testo e binari su disco
- Gestire file con accesso diretto
- Serializzare oggetti
- Usare le classi *ostringstream* e *stringstream* per le conversioni di tipo numero-stringa e stringa-numero
- Usare gli iteratori per scandire i flussi

C++ 16.1

La gerarchia delle classi stream

Gli **stream** (o **flussi**) sono sequenze di byte in cui ciascun byte è individuato da una posizione.

Le classi per la gestione dell'input/output definiscono oggetti per operare sui flussi.
Queste classi sono derivate della classe base astratta virtuale ***ios***.
Le librerie e le classi principali sono:

<iostream>	*istream, ostream, iostream*
<fstream>	*ifstream, ofstream, fstream*
<sstream>	*istringstream, ostringstream, stringstream*

Nota

Tutte le classi derivate da *ios* sono tipi sinonimi di classi generiche il cui nome inizia sempre con *basic_* dopo l'applicazione del template con tipo *char*.

La classe ios

La classe *ios* è tipo sinonimo della classe generica *basic_ios* dopo l'applicazione del template con il tipo *char*.

La classe ***basic_ios*** è una derivata della classe ***ios_base***; gran parte dei moduli di *ios* sono in realtà definiti in *ios_base*. Per comodità si possono considerare come definiti in *ios*.

La classe *ios* tratta sempre sequenze di caratteri e mette a disposizione, oltre a manipolatori per la formattazione dei caratteri, anche attributi e metodi per controllare la correttezza delle operazioni.

Tipi enumerativi

`iostate` definisce i quattro stati assunti da un flusso dopo un'operazione:

```
enum iostate {goodbit, eofbit, failbit, badbit=4};
```

`goodbit`	l'operazione è andata a buon fine;
`eofbit`	è raggiunta la fine del file; ha senso solo per i flussi di input;
`failbit`	l'attuale operazione non è andata a buon fine, tuttavia è possibile continuare con nuove operazioni;
`badbit`	l'attuale operazione non è andata a buon fine e non è possibile continuare.

`openmode` definisce le modalità di apertura di un flusso:

```
enum openmode{in, out, ate, app, trunc, binary, nocreate,
noreplace};
```

`in`	in lettura;
`out`	in scrittura;
`ate`	alla fine del file (EOF);
`app`	in append, cioè in scrittura in coda al file;
`trunc`	in sovrascrittura; se il file non esiste viene creato, altrimenti il suo contenuto viene cancellato;
`binary`	in formato binario; se non viene specificato, l'apertura avviene in modalità testo;
`nocreate`	per un file esistente; se non esiste è assunto lo stato *failbit*;
`noreplace`	per la creazione di un file; se esiste già è assunto lo stato *failbit*.

Nota

Si possono combinare più opzioni di apertura mediante l'operatore |.

seekdir definisce particolari posizioni all'interno di un flusso:

```
enum seekdir {beg, cur, end};
```

`beg`	inizio del flusso; vale 0;
`cur`	posizione corrente;
`end`	fine del flusso (EOF).

Nota

In realtà sono definite costanti omonime per ogni valore del tipo *seekdir* e *openmode*.

Attributi

state di tipo *int*, individua lo stato del flusso; *state* può assumere solo 8 valori che vengono rappresentati con 3 bit; gli stati *eofbit failbit* e *badbit* sono individuati rispettivamente dal valore del primo, secondo e terzo bit; lo stato *goodbit* è assunto quando *state* è nullo.

Metodi

good(), **eof**(), **fail**() e **bad**()
restituiscono il valore booleano (o intero) *true* se il flusso si trova rispettivamente allo stato *goodbit*, *eofbit*, *failbit* e *badbit*, *false* altrimenti.

clear(valStato)
imposta lo stato del flusso al valore di *valStato*; se invocato senza parametri, imposta lo stato a *goodbit*.

rdstate() restituisce il valore di *state*.

exceptions()
permette di impostare dei flag di stato per gestire tramite un'eccezione della classe *ios_base::failure*, l'assunzione del relativo stato da parte del flusso.

```
nomeFlusso.exceptions(ios::eofbit | ios::failbit | ios::badbit);
```

imposta i flag dei tre stati *eofbit*, *failbit* e *badbit*; se il flusso *nomeFlusso* assume uno di questi stati, viene lanciata un'eccezione di classe *ios_base::failure* catturabile con un blocco *try catch*.

Solitamente l'eccezione specificata nella clausola *catch* è ***ios::failure*** e non *ios_base::failure*; infatti essendo *ios* tipo sinonimo di una classe derivata di *ios_base*, può essere considerata come una classe derivata di *ios_base*.

La classe *ios* sovraccarica l'operatore **di negazione !** in modo da restituire il valore di *fail()*; questo è utile per controllare se un flusso *nomeFlusso* è nello stato *failbit*. Le scritture *!nomeFlusso* e *nomeFlusso.fail()* sono equivalenti.

La classe *ios* mette a disposizione anche l'operatore **di conversione esplicita a *void**** che restituisce *NULL* se il valore del metodo *fail()* è *true*. Questo è particolarmente utile per rilevare la fine del flusso o per controllare la correttezza di una condizione.

```
double numero;
while(cin >> numero){
    //istruzioni
}//while
```

usa in modo nascosto l'operatore di conversione esplicita a *void**. La condizione diventa falsa quando si scrive una sequenza di caratteri che non rappresenta un numero di tipo *double*.

C++ 16.2

Le classi per la gestione dei flussi standard

La classe ostream

Nella classe ***ostream***, derivazione virtuale della classe *ios* che identifica il flusso di output standard, sono definiti gli oggetti:

- ***cout*** usato per inviare valori allo standard output (normalmente il video);
- ***cerr*** e ***clog*** usati per inviare messaggi allo standard error (normalmente il video).

L'invio allo standard output e allo standard error avviene tramite l'operatore <<.

La classe *ostream* mette a disposizione molti sovraccarichi di <<, uno per ogni tipo primitivo.

La classe istream

Nella classe ***istream***, derivazione virtuale della classe *ios* che identifica il flusso di input standard, è definito l'oggetto ***cin*** usato per acquisire valori tramite l'operatore >>.

La classe *istream* mette a disposizione molti sovraccarichi di >>, uno per ogni tipo primitivo.

Se il valore letto non è compatibile con il tipo della variabile specificata, viene lasciato nello stream che assume lo stato *failbit*.

Tutti i sovraccarichi di >> leggono solo stringhe prive di spaziature (cioè "spazi", tabulazioni, invii). Se è necessario includere nella lettura le spaziature bisogna usare idonei metodi.

I metodi principali sono:

`int get()` restituisce il carattere letto convertito in intero o -1 se è raggiunta la fine del flusso;

`istream& get(int& c)`
salva in *c* il carattere letto convertito in intero o -1 se è raggiunta la fine del flusso;

`istream& get(char* p, int nc, char delim='\n')`
memorizza nell'area puntata da *p* gli *nc* caratteri estratti oppure tanti caratteri fino al raggiungimento del carattere *delim* (valore di default '\n'); *delim* non è estratto;

`istream& getline(char* p, int nc, char delim='\n')`
memorizza nell'area puntata da *p* gli *nc* caratteri estratti oppure tanti caratteri fino al raggiungimento del carattere *delim* (valore di default '\n'); *delim* è estratto ma non memorizzato;

`istream& ignore(int nc, int delim=EOF)`
estrae *nc* caratteri oppure tanti caratteri fino al raggiungimento del carattere *delim* (valore di default *EOF* cioè la fine del flusso);

`int gcount() const`
restituisce il numero di caratteri estratti nella precedente invocazione di *getline()*.

Esempio 16.1

Realizzazione della funzione *leggi()* per acquisire un numero intero (di tipo *int*). L'acquisizione viene ripetuta finché vengono digitati caratteri che non formano un numero di tipo *int*.

```
#include <iostream>
using namespace std;

int leggi(void){
    int dato;
    bool inputOk = false;
```

```
    cout << "Inserisci un numero intero:\n";
    do{
        try{
            cin >> dato;
            inputOk = true;
        }catch(ios::failure& e){
            cerr << "Non e' un numero intero, riprova!!\n";
            cin.clear();
            cin.ignore(80,'\n');
        }//catch
    }while(!inputOk);
    return dato;
}//leggi

int main(void){
    int numero;
/*imposta il lancio dell'eccezione ios::failure
quando il flusso assume lo stato failbit*/
    cin.exceptions(ios::failbit);
    numero = leggi();
    cout << "Il numero inserito e' " << numero << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

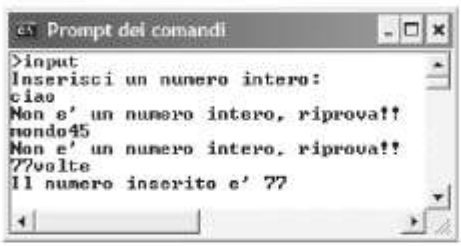


Figura 16.1 *Esecuzione dell'esempio*

La classe iostream

La classe ***iostream*** è una derivazione per ereditarietà multipla delle classi *istream* e *ostream*; non aggiunge nulla di nuovo, riunisce solo le caratteristiche delle due classi base.

Le classi per la gestione dei flussi associati ai file

C++ 16.3

I dati possono essere memorizzati in formato **binario**, cioè come sequenze di **byte**, o in formato **testo**, cioè come sequenze di **caratteri**; il formato binario è più compatto ed efficiente, mentre il formato testo è più comodo e semplice.

La classe ofstream

La classe ***ofstream*** è una classe derivata della classe *ostream* e identifica il flusso di output associato ad un file.
L'accesso ad un flusso *ofstream* è solitamente sequenziale.

Costruttori

`ofstream()`	costruttore di default; crea un flusso non associato a nessun file; è necessario l'uso del metodo *open()*;
`ofstream(const char* nomeFile, openmode modApertura=ios::out)`	crea un flusso associato al file *nomeFile*; *modApertura* indica la modalità di apertura ed è un valore del tipo enumerativo *openmode*; è automaticamente impostato a *out* (scrittura).

Nota

Il nome del file è visto come una sequenza di caratteri; come variabile deve essere usato un array di *char* e non un oggetto della classe *string*.

La scrittura avviene, come nella superclasse *ostream*, con l'operatore <<.

ALTRI METODI

`void` **`open`**`(const char* nomeFile, openmode modApertura=ios::out)`
: associa il flusso al file *nomeFile*; *modApertura* è un valore del tipo enumerativo *openmode*; è automaticamente impostato a *out* (scrittura).

`bool` **`isopen`**`(void)`
: restituisce *true* se il flusso è associato ad un file aperto;

`ofstream&` **`write`**`(const char* p, int nc)`
: scrive sul flusso gli *nc* byte dell'area puntata da *p*;

`ofstream&` **`put`**`(char car)`
: scrive sul flusso il carattere *car*;

`long` **`tellp`**`(void)`
: restituisce l'attuale posizione; è utilizzato nei file con modalità di accesso casuale;

`ifstream&` **`seekp`**`(long pos)`
: imposta il flusso alla posizione *pos*; è utilizzato nei file con modalità di accesso casuale;

`ifstream&` **`seekp`**`(long offset, seekdir dir)`
: imposta il flusso alla posizione distante *offset* byte dalla posizione indicata da *dir* (che può essere *ios::beg* o *ios::cur* o *ios:end*); se *dir* è *ios::end* allora *offset* deve essere negativo;

`void` **`close`**`(void)`
: chiude il file; il flusso associato può essere riutilizzato per un altro file tramite il metodo *open*; è invocato implicitamente dal distruttore alla fine dell'esecuzione.

WRITE()

Il metodo *write()* è particolarmente adatto a scrivere dati in formato binario (sequenze di byte). Se i dati da scrivere sono memorizzati in una variabile *nomeVar* di tipo *TipoVar* (primitivo o strutturato), il metodo *write()* assume la forma:

```
write((char*)&nomeVar, sizeof(TipoVar))
```

ESEMPIO 16.2

Scrittura su un flusso di output associato a un file.

```
#include <iostream>
#include <fstream>
using namespace std;

int main(void){
    ofstream fileo;
    string s;
    fileo.exceptions(ios::failbit | ios::badbit);
    cout << "Inserisci delle frasi (\"fine\" per finire)\n";
    getline(cin, s);
    try {
        fileo.open("prova.txt", ios::out);
        while(s != "fine"){
```

```
            fileo << s;
            fileo.put('\n');
            getline(cin, s);
         }//while
      }catch(ios::failure& e){
         cerr << "Errore di I/O\n";
         return 1;
      }//catch
      cout << "Scrittura eseguita correttamente!!!\n";
      fflush(stdin);
      getchar();
      return 0;
   }//main
```

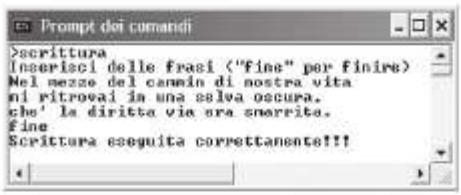


Figura 16.2 *Esecuzione dell'esempio*

La classe ifstream

La classe ***ifstream*** è una classe derivata della classe *istream* e identifica il flusso di input associato ad un file.
L'accesso ad un flusso *ifstream* è solitamente sequenziale.

Costruttori

`ifstream()` costruttore di default; crea un flusso non associato a nessun file; è necessario l'uso del metodo *open()*;

`ifstream(const char* nomeFile, openmode modApertura=ios::in)` crea un flusso associato al file *nomeFile*; *modApertura* è un valore del tipo enumerativo *openmode*; è automaticamente impostato a *in* (lettura).

La lettura avviene sempre con l'operatore >>.

In alternativa, la classe *ifstream* ridefinisce, mantenendone le caratteristiche, i metodi *get()*, *getline()*, *ignore()* e *gcount()* della superclasse *istream*.

Altri metodi

`void open(const char* nomeFile, openmode modApertura=ios::in)`
associa il flusso al file *nomeFile*; *modApertura* è un valore del tipo enumerativo *openmode*; è automaticamente impostato a *in* (lettura).

`bool isopen(void)`
restituisce *true* se il flusso è associato ad un file aperto;

`ifstream& read(char* p, int nc)`
memorizza nell'area puntata da *p* gli *nc* byte estratti dal flusso;

`int readsome(char* p, int nc)`
memorizza nell'area puntata da *p* gli *nc* byte estratti dal flusso e restituisce il numero di byte effettivamente letti;

`long tellg(void)`
restituisce l'attuale posizione; è utilizzato nei file con modalità di accesso casuale;

`ifstream& seekg(long pos)`
imposta il flusso alla posizione *pos*; è utilizzato nei file con modalità di accesso casuale;

`ifstream& seekg(long offset, seekdir dir)`
imposta il flusso alla posizione distante *offset* byte dalla posizione indicata da *dir*

(che può essere *ios::beg* o *ios::cur* o *ios:end*); se *dir* è *ios::end* allora *offset* deve essere negativo;

`void close(void)`
chiude il file; il flusso associato può essere riutilizzato per un altro file tramite il metodo *open*; è invocato implicitamente dal distruttore alla fine dell'esecuzione.

READ() E READSOME()

I metodi *read()* e *readsome()* sono particolarmente adatti a leggere dati in formato binario (sequenze di byte). Se i dati da leggere devono essere memorizzati in una variabile *nomeVar* di tipo *TipoVar* (primitivo o strutturato), il metodo *read()* (e analogamente *readsome()*) assume la forma:

```
read((char*)&nomeVar, sizeof(TipoVar))
```

ESEMPIO 16.3

Lettura di un flusso di input associato al file *prova.txt* dell'esempio 16.2.

La lettura avviene con l'operatore <<; i caratteri "spazio" e "invio" sono dei delimitatori di <<, pertanto si avrà una lettura (e successiva visualizzazione) parola per parola.

```
#include <iostream>
#include <fstream>
using namespace std;

int main(void){
    ifstream file1;
    string s;
    file1.exceptions(ios::badbit);
    cout << "Contenuto del file:\n";
    try {
        file1.open("prova.txt");
        if(!file1){
            cerr << "Il file non esiste!!!\n";
            return 1;
        }//if
        while(file1 >> s){
            cout << s << endl;
        }//while
    }catch(ios::failure& e){
        cerr << "Errore di I/O\n";
        return 2;
    }//catch
    getchar();
    return 0;
}//main
```

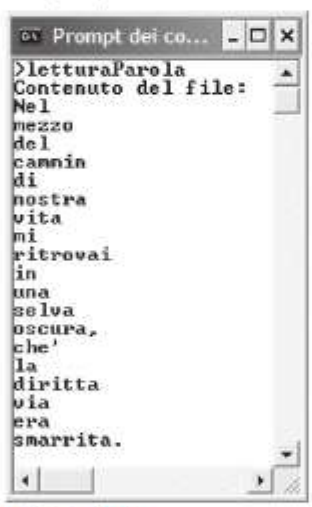


Figura 16.3
Esecuzione dell'esempio

ESEMPIO 16.4

Modifica dell'esempio 16.3. Lettura del flusso carattere per carattere mediante il metodo *get()*.

```
#include <iostream>
#include <fstream>
```

```
using namespace std;
int main(void){
    ifstream filei;
    char c;
    filei.exceptions(ios::badbit);
    cout << "Contenuto del file:\n";
    try {
        filei.open("prova.txt");
        if(!filei){
            cerr << "Il file prova.txt non esiste!!!\n";
            return 1;
        }//if
        while((c = filei.get()) != -1){
            cout << c;
        }//while
    }catch(ios::failure& e){
        cerr << "Errore di I/O\n";
        return 2;
    }//catch
    getchar();
    return 0;
}//main
```

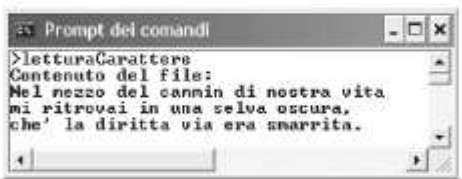


Figura 16.4 *Esecuzione dell'esempio*

La classe fstream

La classe ***fstream*** è una classe derivata della classe *iostream* e identifica un flusso associato ad un file.
Riunisce le funzionalità delle classi *ifstream* e *ofstream*, quindi su un oggetto *fstream* si possono invocare sia i metodi definiti in *ifstream* che in *ofstream*.
L'accesso ad un flusso *fstream* è solitamente ad accesso casuale in quanto è possibile effettuare operazioni sia di lettura che di scrittura. Questa possibilità dipende dalla modalità di apertura del flusso.

Costruttori

`fstream()`	costruttore di default; crea un flusso non associato a nessun file; è necessario l'uso del metodo *open*;
`fstream(const char* nomeFile, openmode modApertura)`	crea un flusso associato al file *nomeFile*; *modApertura* indica la modalità di apertura; solitamente è in lettura e scrittura (*ios::in* / *ios::out*).

Nota

Se il file da aprire in lettura e scrittura non esiste, la modalità di apertura può essere: *ios::in* / *ios::out* / *ios::trunc*.

La lettura e la scrittura possono avvenire con gli operatori >> e << oppure con i metodi di *ifstream* e *ofstream*.

Esempio 16.5

Scrittura e lettura di un file di testo mediante un flusso *fstream*.

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
int scrittura(fstream& f);
int lettura(fstream& f);
int main(void){
    fstream file;
    file.open("prova.txt", ios::out);
    if(!file){
        cerr << "Errore di I/O\n";
        return 1;
    }//if
    scrittura(file);
    file.close();
    file.open("prova.txt", ios::in);
    if(!file){
        cerr << "Il file prova.txt non esiste!!!\n";
        return 3;
    }//if
    lettura(file);
    getchar();
    return 0;
}//main
int scrittura(fstream& f){
    string s;
    f.exceptions(ios::failbit | ios::badbit);
    cout << "Scrittura sul file\n\n";
    cout << "Inserisci delle frasi (\"fine\" per finire)\n";
    getline(cin, s);
    try {
        while(s != "fine"){
            f << s;
            f.put('\n');
            getline(cin, s);
        }//while
    }catch(ios::failure& e){
        cerr << "Errore di I/O\n";
        return 2;
    }//catch
    cout << "\nScrittura eseguita correttamente!!!\n";
}//scrittura
int lettura(fstream& f){
    char c;
    f.exceptions(ios::badbit);
    cout << "\nLettura del file\n\n";
    cout << "Contenuto del file:\n";
    try {
        while((c = f.get()) != -1){
            cout << c;
        }//while
    }catch(ios::failure& e){
        cerr << "Errore di I/O\n";
        return 4;
    }//catch
```

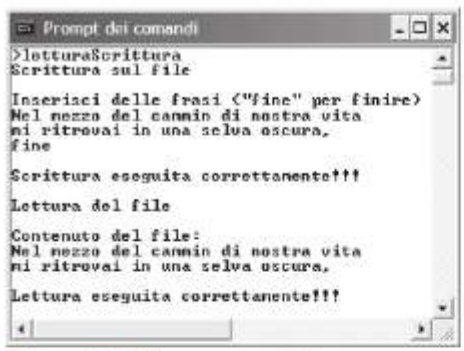


Figura 16.5 *Esecuzione dell'esempio*

```
    cout << "\nLettura eseguita correttamente!!!\n";
}//lettura
```

Esempio 16.6

Scrittura di alcuni valori di vario tipo nel file e lettura in ordine diverso da quello di scrittura.

```
#include <iostream>
#include <fstream>
using namespace std;

int main(void){
    fstream file;
    bool bw = true, br;
    char cw = 'h', cr;
    int iw = 1360, ir;
    double dw = 1300.55, dr;
    file.exceptions(ios::failbit | ios::badbit);
    try {
        file.open("prova.txt", ios::in | ios::out | ios::trunc);
        file.write((char*)&bw, sizeof(bool));
        file.write((char*)&iw, sizeof(int));
        file.write((char*)&cw, sizeof(char));
        file.write((char*)&dw, sizeof(double));
        file.seekg(sizeof(bool)+sizeof(int));
        file.read((char*)&cr, sizeof(char));
        file.read((char*)&dr, sizeof(double));
        file.seekg(ios::beg);
        file.read((char*)&br, sizeof(bool));
        file.read((char*)&ir, sizeof(int));
    }catch(ios::failure& e){
        cerr << "Errore di I/O\n";
        return 1;
    }//catch
    cout << "carattere: " << cr << endl;
    cout << "numero reale: " << dr << endl;
    cout << "booleano: " << br << endl;
    cout << "numero intero: " << ir << endl;
    getchar();
    return 0;
}//main
```

Figura 16.6 *Esecuzione dell'esempio*

Serializzazione di oggetti

L'operazione di scrittura di oggetti in un flusso è chiamata **serializzazione**.
Scrivendo gli oggetti in un file si rendono **persistenti**, cioè è possibile riutilizzare gli oggetti dopo il termine dell'applicazione.

Nota

In C++ non esistono classi e metodi specifici per la serializzazione.

Solitamente, per ottenere la serializzazione, nella classe che definisce l'oggetto si implementa un metodo apposito per scrivere su un flusso tutti gli attributi che compongono l'oggetto. Se un attributo è a sua volta un oggetto anche la sua classe deve prevedere un metodo di scrittura sul flusso.

È conveniente salvare gli oggetti in formato binario. Se un attributo è una stringa conviene memorizzarla come array di char in quanto la lunghezza certa dell'array agevola la rilettura dell'oggetto e la sua ricostruzione.

Per **rileggere** un oggetto si dichiara nella classe che definisce l'oggetto un metodo per estrarre da un flusso gli attributi e assegnarli all'oggetto implicito.

ESEMPIO 16.7

Scrittura e lettura di oggetti in un file.

```
//Punto.h
#include <iostream>
#include <fstream>
using namespace std;

class Punto {
    private:
        int x, y;
    public:
        int getX(void) const;
        int getY(void) const;
        double distanza(const Punto&) const;
        void serializzaOut(ofstream&);
        void serializzaIn(ifstream&);
        friend ostream& operator<<(ostream&, Punto);
        Punto(int =0, int =0);
};//Punto
```

```
//Punto.cpp
#include <cmath>
#include "Punto.h"

//altri metodi

void Punto::serializzaOut(ofstream& oss){
    oss.write((char*)&x, sizeof(int));
    oss.write((char*)&y, sizeof(int));
}//serializzaOut

void Punto::serializzaIn(ifstream& iss){
    iss.read((char*)&x, sizeof(int));
    iss.read((char*)&y, sizeof(int));
}//serializzaIn
```

```
//leggiScriviPunti.cpp
#include <iostream>
#include <fstream>
#include "Punto.h"
using namespace std;

void scrittura(void){
    Punto p;
    ofstream fileout("prova.dat");
    for(int i=0; i<10; i++){
```

```
        p = Punto(i, i);
        p.serializzaOut(fileout);
    }//for
}//scrittura

void lettura(void){
    Punto p;
    ifstream filein("prova.dat");
    while(!filein.eof()){
        p.serializzaIn(filein);
        cout << p << endl;
    }//while
}//lettura

int main(void){
    scrittura();
    lettura();
    getchar();
    return 0;
}//main
```

Figura 16.7 *Esecuzione dell'esempio*

Le classi per la gestione dei flussi associati a stringhe

C++ 16.4

I dati possono essere letti o scritti in formato **binario**, cioè come sequenze di **byte**, o in formato **testo**, cioè come sequenze di **caratteri**.

La classe ostringstream

La classe ***ostringstream*** è una classe derivata della classe *ostream* e identifica il flusso di output associato ad una stringa.
Un flusso *ostringstream* è visto come una sequenza di byte memorizzati in memoria centrale su cui è possibile effettuare operazioni che modificano la sequenza stessa.
L'accesso ad un flusso *ostringstream* è solitamente sequenziale.

Costruttori

ostringstream(openmode modApertura=ios::out)
costruttore di default; crea un flusso di uscita vuoto; il parametro è di default e può essere omesso;

ostringstream(const string& s, openmode modApertura=ios::out)
costruttore di copia; crea un flusso di uscita copiando i caratteri dalla stringa *s*.

La scrittura avviene, come nella superclasse *ostream*, con l'operatore <<.

La classe *ostringstream* eredita tutti i metodi delle superclassi ed è utile per le conversioni tra tipi primitivi.

Altri metodi

string **str**(void)
restituisce il contenuto del flusso visto come oggetto della classe *string*.

void **str**(const string& s)
imposta il contenuto del flusso al valore della stringa *s*.

Esempio 16.8

Scrittura di alcuni valori di vario tipo in un flusso *ostringstream*.

```
#include <iostream>
#include <sstream>
using namespace std;

int main(void){
    string nome;
    int annoN;
    double oraN;
    ostringstream oss;
    cout << "Inserisci il nome:\n";
    cin >> nome;
    cout << "Inserisci l'ora di nascita (es. 12.00):\n";
    cin >> oraN;
    cout << "Inserisci l'anno di nascita (es. 1995):\n";
    cin >> annoN;
    oss << nome << " " << oraN << " " << annoN << endl;
    cout << "Il flusso scritto ha contenuto:\n"
         << oss.str() << endl;
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 16.8 *Esecuzione dell'esempio*

La classe istringstream

La classe ***istringstream*** è una classe derivata della classe *istream* e identifica il flusso di input associato ad una stringa.
Un flusso *istringstream* è visto come una sequenza di byte memorizzati in memoria centrale su cui è possibile effettuare operazioni per accedere (e non modificare) il contenuto della sequenza stessa.

L'accesso ad un flusso *istringstream* è solitamente sequenziale.

Costruttori

istringstream(openmode modApertura=ios::in)
: costruttore di default; crea un flusso di input vuoto; il parametro è di default e può essere omesso;

istringstream(const string& s, openmode modApertura=ios::in)
: costruttore di copia; crea un flusso di input copiando i caratteri dalla stringa *s*.

La lettura avviene, come nella superclasse *istream*, con l'operatore >>.

La classe *istringstream* eredita tutti i metodi delle superclassi.

Esempio 16.9

Estrazione di alcuni valori di vario tipo da un flusso *istringstream*.

```
#include <iostream>
#include <sstream>
using namespace std;
```

```
int main(void){
    string s = "Chiara 18.11 1991", nome;
    int annoN;
    double oraN;
    istringstream iss(s);
    iss >> nome >> oraN >> annoN;
    cout << nome << " e' nata\nnell'anno "
         << annoN << "\nalle ore " << oraN << endl;
    getchar();
    return 0;
}//main
```

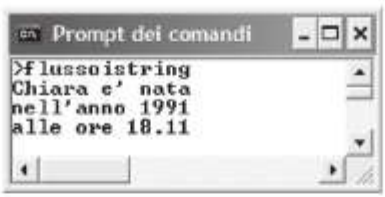


Figura 16.9 *Esecuzione dell'esempio*

La classe stringstream

La classe ***stringstream*** è una classe derivata della classe *iostream* e identifica un flusso associato ad una stringa.

Riunisce le funzionalità delle classi *istringstream* e *ostringstream*, quindi su un oggetto *stringstream* si possono invocare sia i metodi definiti in *istringstream* che in *ostringstream*.

L'accesso ad un flusso *stringstream* è solitamente ad accesso casuale in quanto è possibile effettuare operazioni sia di lettura che di scrittura.

Costruttori

stringstream(openmode modApertura=ios::in|ios::out)
: costruttore di default; crea un flusso vuoto; il parametro è di default e può essere omesso;

stringstream(const string& s, openmode modApertura=ios::in|ios::out)
: costruttore di copia; crea un flusso copiando i caratteri dalla stringa *s*.

La lettura e la scrittura possono avvenire con gli operatori >> e << oppure con i metodi ereditati da *istream* e *ostream*.

Esempio 16.10

Scrittura di alcuni valori di vario tipo su un flusso *stringstream* e successiva estrazione.

```
#include <iostream>
#include <sstream>
using namespace std;

void lettura(stringstream &ss);
void scrittura(stringstream &ss);

int main(void){
    stringstream stringa;
    cout << "Scrittura flusso\n\n";
    scrittura(stringa);
    cout << "\nLettura flusso\n\n";
    lettura(stringa);
    fflush(stdin);
    getchar();
    return 0;
}//main
```

Figura 16.10 *Esecuzione dell'esempio*

```
void lettura(stringstream &ss){
    string nome;
    int annoN;
    double oraN;
    ss >> nome >> oraN >> annoN;
    cout << "Nome: " << nome << endl;
    cout << "Anno: " << annoN << endl;
    cout << "Ora: " << oraN << endl;
}//lettura

void scrittura(stringstream &ss){
    string nome;
    int annoN;
    double oraN;
    cout << "Inserisci il nome:\n";
    cin >> nome;
    cout << "Inserisci l'ora di nascita (es. 12.00):\n";
    cin >> oraN;
    cout << "Inserisci l'anno di nascita (es. 1995):\n";
    cin >> annoN;
    ss << nome << " " << oraN << " " << annoN << endl;
}//scrittura
```

C++ 16.5

ITERATORI PER I FLUSSI

Per scorrere un flusso si possono usare gli iteratori.
Il file ***<iterator>*** della libreria STL definisce oltre agli iteratori per i contenitori anche due tipi di iteratori per i flussi:

- ***istream_iterator*** per scorrere un flusso senza modificarne il contenuto;
- ***ostream_iterator*** per scorrere un flusso con la possibilità di modificarne il contenuto e di aggiungerne elementi.

COSTRUTTORI

`istream_iterator``(void)`
: definisce un iteratore che si riferisce alla posizione dopo l'ultimo elemento del flusso; è utilizzato per verificare se si è raggiunta la fine del flusso;

`istream_iterator``(tipoFlusso nomeFlusso)`
: definisce un iteratore che si riferisce alla prima posizione del flusso specificato;

`ostream_iterator``(tipoFlusso nomeFlusso)`
: definisce un iteratore che si riferisce alla prima posizione del flusso specificato; è possibile inserire elementi nel flusso;

`ostream_iterator``(tipoFlusso nomeFlusso, TipoElFlusso separatore)`
: definisce un iteratore che si riferisce alla prima posizione del flusso specificato; è possibile inserire elementi nel flusso che saranno separati da *separatore*.

Nota

Gli iteratori sono classi generiche, quindi è necessario, al momento della dichiarazione/associazione al flusso, specificare il tipo di elementi del flusso.

`ofstream fileout("prova.txt");`
flusso di uscita associato al file di testo *prova.txt*;
`ostream_iterator<string> ito(fileout, "\n");`
definisce *ito* come iteratore per scrivere elementi di tipo *string* separati dalla stringa "*\n*" nel flusso *fileout*;

`ifstream filein("prova.txt");`
flusso di ingresso associato al file di testo *prova.txt*;
`istream_iterator<string> iti(filein);`
definisce *iti* come iteratore per scorrere gli elementi di tipo *string* del flusso *filein*;
`istream_iterator<string> itFine;`
definisce *itFine* come iteratore da usare in un confronto per individuare la fine del file.

Per accedere all'elemento individuato da un iteratore si usa ancora l'operatore di **indirezione** *.

Nota

Gli iteratori per i flussi sono utilizzati soprattutto nelle applicazioni in cui sono presenti anche i contenitori di STL.

Esempio 16.11

Modifica dell'esempio 16.2. Scrittura su un flusso di output associato a un file mediante un iteratore *ostream_iterator*.

```
#include <iostream>
#include <fstream>
#include <iterator>
using namespace std;

int main(void){
    ofstream fileo;
    ostream_iterator<string> ito(fileo, "\n");
    string s;
    fileo.exceptions(ios::failbit | ios::badbit);
    cout << "Inserisci delle frasi (\"fine\" per finire)\n";
    getline(cin, s);
    try {
        fileo.open("prova.txt");
        while(s != "fine"){
            *ito++ = s;
            getline(cin, s);
        }//while
    }catch(ios::failure& e){
        cerr << "Errore di I/O\n";
        return 1;
    }//catch
    cout << "Scrittura eseguita correttamente!!!\n";
    fflush(stdin);
    getchar();
    return 0;
}//main
```

ESEMPIO 16.12

Modifica dell'esempio 16.3. Lettura da un flusso di input associato a un file mediante un iteratore *istream_iterator*. La lettura avviene parola per parola.

```
#include <iostream>
#include <fstream>
#include <iterator>
using namespace std;

int main(void){
    ifstream filei("prova.txt");
    if(!filei){
        cerr << "Il file prova.txt non esiste!!!\n";
        return 1;
    }//if
    istream_iterator<string> iti(filei);
    istream_iterator<string> itFine;
    string s;
    filei.exceptions(ios::badbit);
    cout << "Contenuto del file:\n";
    try {
        while(iti != itFine){
            cout << *iti++ << endl;
        }//while
    }catch(ios::failure& e){
        cerr << "Errore di I/O\n";
        return 2;
    }//catch
    getchar();
    return 0;
}//main
```

VERIFICA

QUESTIONARIO

1. In quale file di libreria si trovano le classi per la gestione dell'input/output?
2. Qual è la differenza tra flussi di byte e flussi di caratteri?
3. Quali sono le classi principali?
4. Quali sono le modalità di apertura di un flusso?
5. Quali stati può assumere un flusso e come si determina in quale stato si trova?
6. Come si gestiscono gli errori derivanti da operazioni su un flusso?
7. Come si leggono sequenze di caratteri da un flusso di input?
8. Come si leggono sequenze di byte da un flusso di input?
9. Come si scrivono sequenze di caratteri in un flusso di output?
10. Come si scrivono sequenze di byte in un flusso di output?
11. Come si determina il raggiungimento della fine di un flusso di input?
12. In un file con modalità di accesso casuale quali operazioni si possono fare?

13. Che cosa vuol dire serializzazione di oggetti?
14. Cosa sono i flussi associati a stringhe e a cosa servono?
15. Quali sono gli iteratori per i flussi?

TEST

1 ..... *Indicare se le seguenti affermazioni sono vere o false.*

V	F	
☐	☐	1) I flussi possono essere associati solo a file di testo.
☐	☐	2) Le classi *ostream* e *istream* sono entrambe derivazioni virtuali della classe *ios*.
☐	☐	3) La classe *ifstream* è una derivata della classe *iostream*.
☐	☐	4) L'accesso ad un flusso *fstream* può essere sia sequenziale che casuale.
☐	☐	5) La classe *fstream* è una derivata della classe *iostream*.
☐	☐	6) Per serializzazione si intende l'operazione di scrittura di oggetti in un flusso.
☐	☐	7) La classe *ostringstream* è una derivata della classe *stringstream*.
☐	☐	8) La classe *stringstream* identifica un flusso associato ad un file di stringhe.
☐	☐	9) Un iteratore di tipo *istream_iterator* non può modificare gli elementi del flusso.
☐	☐	10) Un iteratore di tipo *ostream_iterator* non può aggiungere elementi al flusso.

PROPOSTE DI LAVORO

1 ..... Visualizzare le righe di un file precedute da un numero.

2 ..... Dato un file di testo calcolare la frequenza di ciascuna lettera, degli spazi e dei simboli di punteggiatura.

3 ..... Dato un file di testo calcolare la frequenza delle parole di 1 lettera, 2 lettere, 3 lettere e così via.

4 ..... Dato un file di testo calcolare il numero di caratteri, il numero di parole e il numero di righe.

5 ..... Cercare una parola in un file di testo e stampare le righe che la contengono.

6 ..... Scrivere in un file una serie di operazioni una per riga (operando, operazione, operando); poi leggere il file ed eseguire le operazioni.

7 ..... Scrivere in un file nome città, temperatura minima e massima;
- dare le temperature di una città richiesta,
- calcolare le escursioni termiche di tutte le città.

8 ..... Un file di testo contiene domande a risposta del tipo vero o falso, prima la risposta (*v* o *f*), uno spazio, poi sulla stessa riga la domanda; porre la domanda e controllare la risposta. Alla fine fornire la percentuale di risposte esatte.

9 ..... Dato un file di testo calcolare la frequenza delle parole che vi compaiono e scrivere in un secondo file, una parola per riga, parola e relativa frequenza.

10 ..... Scrivere un programma che accetti come parametri alcuni nomi di file, che devono esistere tutti tranne l'ultimo, che può esistere o meno. Aggiungere il contenuto di

ciascun file, fino al penultimo, all'ultimo file. In pratica concatenare vari file in uno solo (per esempio i capitoli in un libro, ultimo parametro).

11 ..... Dati due file ordinati creare un file con la fusione dei dati, prendendo una sola volta gli elementi comuni.

12 ..... Controllare in un testo che tutte le parole dopo il punto comincino con la lettera maiuscola ed apportare le necessarie correzioni (copiando il file in un file temporaneo che poi deve sostituire l'originale).

13 ..... Date due stringhe, una da cercare e una da sostituire, leggere un file, cercare la prima stringa e sostituirla con la seconda (copiando il file in un file temporaneo che poi deve sostituire l'originale).
In più si può visualizzare la riga che contiene la stringa e chiedere l'autorizzazione all'utente prima di effettuare l'aggiornamento.

14 ..... Memorizzare in un file ad accesso diretto per ogni mese nome (tutti i nomi della stessa lunghezza) e numero dei giorni. Poi dato il numero di un mese visualizzare nome e numero di giorni ricavandoli dal file.

15 ..... Memorizzare in un file ad accesso diretto il ricavato delle vendite nei 12 mesi dell'anno di 10 anni consecutivi. Dato mese e anno visualizzare il dato corrispondente.

16 ..... Memorizzare in un file ad accesso diretto le coordinate di un insieme di punti.
Gestire in base al numero del punto:
- inserimento,
- elenco dei punti,
- visualizzazione di un punto,
- modifica di una coordinata o di tutte e due,
- visualizzazione dei punti sull'asse x.

17 ..... Memorizzare in un file ad accesso diretto i dati sui prodotti:
- descrizione memorizzata con un numero fisso di caratteri, aggiungendo spazi o troncando caratteri,
- prezzo (double).

Deve essere possibile:
- aggiungere prodotti,
- elencare i prodotti,

e, in base alla posizione del prodotto:
- visualizzare i dati di un prodotto,
- variare i dati di un prodotto.

18 ..... Salvare in un file le informazioni relative a una *Persona*.

19 ..... Rileggere e stampare le informazioni relative a una *Persona*.

20 ..... Gestire una rubrica memorizzando in un file i dati di ogni *Contatto* (nome, numero di telefono, indirizzo e-mail). Rileggere i dati e sistemarli ordinatamente in un array.
Prevedere operazioni di:
- inserimento,
- elenco,
- ricerca,
- modifica.

Al termine riscrivere i dati nel file.

21 ..... Salvare in un file le informazioni sugli studenti della scuola, usando le classi *Classe*, *Scuola* e *Studente* (vedi cap. 12 Proposte di lavoro: realizzare le classi n. 5 e n. 6).
Rileggere le informazioni sugli studenti della scuola memorizzate e visualizzare l'elenco delle classi.

APPENDICE

A COMPLESSITÀ COMPUTAZIONALE

A COMPLESSITÀ COMPUTAZIONALE

A.1 LA COMPLESSITÀ COMPUTAZIONALE

Si dice **complessità di un algoritmo** la quantità di tempo che occorre alla CPU per eseguire il programma relativo.

Per risolvere un problema possono essere disponibili algoritmi diversi (per esempio per ordinare un array sono disponibili molti metodi diversi).

Si dice **complessità computazionale di un problema** la complessità dell'algoritmo più efficiente per risolvere il problema, cioè di quello che viene eseguito più velocemente.

Il calcolo del tempo di esecuzione di un algoritmo viene fatto in modo approssimato, determinandone l'**ordine di grandezza** e non il valore effettivo (che non potrebbe essere calcolato in modo generale, poiché dipende per esempio dal linguaggio scelto per la codifica, dal compilatore o interprete usato, dal processore su cui viene eseguito il programma ecc.).

La misura del tempo di esecuzione di un algoritmo e quindi la complessità dell'algoritmo viene espressa in funzione della **dimensione dei dati di input** (numero di elementi di un array, numero di cifre di un valore ecc.) o in certi casi addirittura in funzione dello stesso valore di input.
La complessità dell'algoritmo può essere espressa dalla funzione ***T(n)*** che rappresenta il tempo impiegato per l'esecuzione dell'algoritmo applicato a dati di input di dimensione *n*.

Costo delle istruzioni di un algoritmo

La complessità di un algoritmo si può valutare calcolando il **costo dell'algoritmo**.

Il costo dell'algoritmo si può esprimere in base al **numero di operazioni** da effettuare con le seguenti regole:

- il costo di istruzioni semplici, come quelle di lettura, scrittura o assegnazione, è pari a 1;
- il costo di una sequenza di istruzioni è la somma dei costi delle singole istruzioni;
- il costo di una struttura condizionale è il costo massimo tra i due rami;
- il costo di un ciclo è dato dal costo delle istruzioni del ciclo moltiplicato per il numero di ripetizioni del ciclo effettuate.

Analisi del caso ottimo, pessimo e medio

Il tempo impiegato da un algoritmo può dipendere non solo dalla dimensione dei dati di input ma anche dalla loro **disposizione**.

Per esempio il tempo necessario per l'ordinamento di un array di dimensione *n* dipende dalla posizione dei dati.

In tal caso la complessità può essere calcolata sul **caso ottimo**, sul **caso pessimo** o **in media**.

Per l'ordinamento di un array il caso ottimo è che l'array sia già ordinato, il caso pessimo che i dati siano disposti nell'ordine inverso, il caso medio che i dati siano disposti in ordine casuale.

Complessità asintotica

Quello che interessa è l'ordine di grandezza della complessità, cioè la complessità per valori molto grandi della dimensione del problema (**complessità asintotica** nel tempo).

La notazione con cui si esprime l'ordine di grandezza di una funzione viene detta notazione **O-grande**.

La complessità asintotica nel tempo di un algoritmo viene determinata dalla funzione $f(n)$ che è O-grande per la funzione di complessità $T(n)$.

Notazione O-grande

Date due funzioni $f(x)$ e $g(x)$ si dice che $f(x)$ è $O(g(x))$ (si legge $f(x)$ è O-grande di $g(x)$) quando si può determinare una costante c tale che, per valori abbastanza grandi di x, cioè da un certo x_0 in poi, si ha che $f(x) \leq cg(x)$.

Ciò significa che da un certo punto in poi i valori assunti dalla funzione $f(x)$ sono più piccoli o al più uguali a quelli della funzione $cg(x)$, o anche che, rappresentando le due funzioni su un piano cartesiano, da un certo valore di x in poi, il grafico di $f(x)$ sta sempre sotto al grafico di $cg(x)$.

Date le funzioni $f(x) = x+2$ e $g(x) = x$ si ha che $f(x) = O(g(x))$, infatti esistono una costante $c=2$ e un valore $x_0=2$ tali che $f(x) \leq cg(x)$ per $x>x_0$ cioè $x+2 \leq 2x$ per $x>2$.

x	0	1	2	3	4	5	6	7
f(x) = x+2	2	3	4	5	6	7	8	9
2g(x) = 2x	0	2	4	6	8	10	12	14

Per una funzione $f(x)$ esistono molte funzioni $g(x)$ tali che $f(x) \leq cg(x)$.
Come funzione O-grande deve essere scelta quella che meglio approssima la funzione $f(x)$, quindi la più piccola possibile, quella che cresce più lentamente.

Classi di complessità

In base agli ordini di grandezza si possono individuare le **classi di complessità**:

- **complessità costante**: $T(n)=O(1)$ il tempo di esecuzione dell'algoritmo non dipende dall'input ma si mantiene costante; è il miglior caso possibile;
- **complessità logaritmica**: $T(n)=O(\log_2 n)$ il tempo di esecuzione aumenta secondo il logaritmo della dimensione dei dati di input; il logaritmo è una funzione i cui valori aumentano molto lentamente, quindi un algoritmo con complessità logaritmica è uno tra i casi migliori;
- **complessità lineare**: $T(n)=O(n)$ il tempo di esecuzione aumenta in modo proporzionale alla dimensione dei dati di input;
- **complessità polinomiale**: $T(n)=O(n^k)$ il tempo di esecuzione aumenta in modo proporzionale a una potenza della dimensione dei dati di input; se $k=2$ si parla di complessità **quadratica**;

Nota: La complessità lineare è un caso particolare della complessità polinomiale.

- **complessità esponenziale**: $T(n)=O(k^n)$ (con $k>1$) la dimensione dei dati di input compare nell'esponente della funzione per il calcolo del tempo; è il caso peggiore; il tempo aumenta molto rapidamente anche per piccoli incrementi della dimensione dei dati di input.

COMPLESSITÀ DI ALCUNI ALGORITMI

RICERCA SEQUENZIALE IN UN ARRAY DI N ELEMENTI

caso ottimo: il valore cercato si trova nella prima posizione dell'array $T_{ottimo}(n) = 1$;
caso pessimo: il valore cercato si trova nell'ultima posizione o non è presente $T_{pessimo}(n) = n$;
caso medio: è dato dalla media del numero di confronti necessari per ogni posizione del valore cercato $T_{medio}(n) = (n+1)/2 = O(n)$ infatti $(1+2+...+(n-1)+n)/2 = (n+1)/2$.

RICERCA BINARIA IN UN ARRAY DI N ELEMENTI

caso ottimo: il valore cercato si trova al centro dell'array $T_{ottimo}(n) = 1$;
caso pessimo: si ha quando si arriva ad eseguire la ricerca su insiemi di un elemento;
$T_{pessimo}(n) = 1+INT(\log_2 n) = O(\log n)$;
caso medio: $T_{medio}(n)=O(\log n)$.

La complessità del caso pessimo della ricerca binaria risulta nettamente inferiore anche alla complessità del caso medio della ricerca sequenziale; l'algoritmo di ricerca binaria risulta molto più veloce e la differenza risulta sempre più notevole al crescere di *n*.

ORDINAMENTO PER BUBBLE SORT DI UN ARRAY DI N ELEMENTI

caso ottimo: gli elementi sono già ordinati; si hanno $n-1$ confronti e quindi $T_{ottimo}(n) = n-1 = O(n)$;
caso pessimo: gli elementi sono in ordine inverso; si hanno $(n-1)+(n-2)+....+2+1=n(n-1)/2$ confronti e $n(n-1)/2$ scambi e quindi $n(n-1)$ operazioni cioè $T_{pessimo}(n) = n(n-1) = n^2-n$ che è $O(n^2)$;
caso medio: $T_{medio}(n) = O(n^2)$.

Nota

Gli algoritmi di ordinamento più efficienti sono $O(n\log n)$.

A.2 PROBLEMI INTRATTABILI

In base alla complessità computazionale i problemi vengono separati in:

- problemi la cui complessità è **al più polinomiale**;
- problemi con complessità **esponenziale**.

I problemi con complessità esponenziale vengono detti **intrattabili** poiché anche per input di piccole dimensioni il tempo necessario per risolverli risulta troppo elevato (quindi non sono effettivamente risolvibili).

Il fatto che questi problemi siano intrattabili non dipende dall'attuale velocità dei processori ma dalla natura stessa del problema. Anche aumentando di migliaia di volte la velocità del processore, i valori di *n* per cui il problema potrebbe essere risolto aumenterebbero solo di qualche unità.
Pur essendo **computabili**, i problemi risolvibili con algoritmi di complessità esponenziale non sono effettivamente risolvibili in pratica, poichè richiederebbero tempi di elaborazione troppo elevati (anche dell'ordine di secoli).

Un problema si dice **effettivamente computabile** se la sua complessità è al più polinomiale.

Alcuni problemi ammettono soluzioni con complessità polinomiale, anche se le migliori soluzioni hanno complessità esponenziale (sono intrinsecamente esponenziali; un esempio di questo tipo è il problema del gioco degli scacchi).
Altri vengono ritenuti di complessità esponenziale perché non si è ancora trovato un algoritmo di complessità polinomiale, anche se non si è dimostrata rigorosamente la complessità esponenziale del problema (si dovrebbe cioè "dimostrare" che non esiste una soluzione polinomiale).

INDICE ANALITICO

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

In copertina: *Mens sana*... © ICPonline

Informatica
Programmazione in C++

Questo volume è destinato a tutti i corsi di Informatica in cui si studiano gli elementi fondamentali della programmazione utilizzando il linguaggio **C++**.
La prima parte del libro descrive la **programmazione imperativa**, mentre la seconda parte affronta la **programmazione a oggetti** illustrando sia le tecniche di programmazione e di progettazione di applicazioni che la libreria standard **STL**.
Sono affrontati anche argomenti complessi come le **eccezioni**, la **genericità** e la gestione dei **flussi**.
Per un approfondimento della programmazione a oggetti è proposta un'introduzione all'utilizzo di alcuni **design pattern** (schemi che descrivono soluzioni riutilizzabili di particolari problematiche di progettazione).
Come ambiente di programmazione si è scelto **wxDev-C++**, un ambiente di sviluppo integrato gratuito, che utilizza il compilatore **MingW** (GNU C++).
Per tutti gli argomenti affrontati si è cercato di dare una interpretazione molto attuale, non soltanto dal punto di vista teorico ma anche con riferimenti concreti agli strumenti utilizzati nella pratica lavorativa.

Il libro è diviso in **moduli**, a loro volta ripartiti in **capitoli**.

- **Modulo 1** – Lo sviluppo del software
- **Modulo 2** – La programmazione imperativa
- **Modulo 3** – La programmazione a oggetti
- **Appendice** – Complessità computazionale

Per consentire un utilizzo più agevole dei volumi, che costituiscono l'opera, ogni capitolo comprende una sezione di **teoria** corredata di numerosi esempi, e una sezione di verifica con domande, test, esercizi e proposte di lavoro. Gli argomenti per l'effettuazione di esercitazioni di **laboratorio** sono integrati strettamente nel testo.

QUESTO CORSO È COSTITUITO DA:
ISBN 978-88-8433-**025**-3 IL COMPUTER E LA PROGRAMMAZIONE IN VISUALBASIC
ISBN 978-88-8433-**026**-0 IL COMPUTER E LA PROGRAMMAZIONE IN PASCAL
ISBN 978-88-8433-**630**-9 SISTEMI OPERATIVI, RETI E INTERNET PER L'AZIENDA
ISBN 978-88-8433-**031**-4 CD-ROM PER L'INSEGNANTE
ISBN 978-88-8433-**027**-7 DATABASE
ISBN 978-88-8433-**029**-1 PROGRAMMAZIONE IN C ++
ISBN 978-88-8433-**030**-7 PROGRAMMAZIONE IN JAVA

J029
SCORZONI, COSTA
INFORMATICA
PROGRAMMAZIONE IN C++
NELL'ELENCO DEI LIBRI DI TESTO

EDUCAZIONE
TECNICA
SUPERIORE