UNIVERSITÀ DI GENOVA
FACOLTÀ DI SCIENZE M.F.N.

# Dispense del corso di Laboratorio di Calcolo A
a.a. 2002-2003

Paolo Morettini, Patrizia Boccacci
Alessandro Brunengo, Claudia Gemme, Fabrizio Parodi

20th December 2002

# Contents

# Capitolo 1

# Dispositivi elettronici digitali

## 1.1 Introduzione

Con il termine "segnale" indichiamo una grandezza fisica che viene utilizzata per trasmettere o manipolare informazioni concernenti un determinato sistema. In generale lo stato del sistema in esame sarà vaiabile nel tempo, e così anche i segnali che lo rappresentano saranno variabili. Si definisce segnale digitale un segnale che può, nel corso della sua evoluzione temporale, assumere due soli valori, che vengono identificati con i numeri 0 e 1 o con il concetto di "vero" e "falso". L' unità di informazione trasportata da un segnale digitale si dice "bit". Un singolo segnale digitale può rappresentare due soli stati distinti di un sistema (Fig. 1.1); al contrario un segnale analogico, che può assumere un valore qualsiasi, è in grado di rappresentare un numero molto elevato di stati differenti, limitato essenzialmente dal range dinamico (ovvero dall' ampiezza dell' intervallo di valori che il segnale può assumere) e dalla precisione dello strumento con cui si effettua la misura. Tuttavia un segnale analogico risulta essere molto soggetto al rumore e, in generale, ai disturbi esterni; di conseguenza è difficile da utilizzare in tutti quei casi, come ad esempio i calcoli matematici, in cui è essenziale che ogni stato di un sistema venga identificato in modo univoco e riproducibile da uno stesso valore del segnale che lo rappresenta; inoltre la decodifica di un segnale analogico richiede l' utilizzo di un sistema di misura preciso, e quindi costoso. I segnali digitali al contrario possono facilmente essere resi immuni dai disturbi esterni, a patto che si selezionino due intervalli di valori ammessi rispettivamente per lo zero e per l' uno; inoltre il dispositivo necessario ad identificare lo stato di un segnale digitale (discriminatore) è molto semplice ed economico. Siccome in generale un sistema avrà più di due stati possibili saranno necessari più segnali digitali per descriverlo completamente.
Nella pratica i segnali che si utilizzano nei sistemi digitali sono molto spesso differenze di potenziale; nel seguito utilizzeremo quindi come sinonimi i termini "sistema digitale" e "circuito digitale", sebbene il primo si riferisca ad un generico dispositivo che manipola un segnale digitale, mentre il secondo indica specificamente dispositivi elettronici che manipolano segnali elettrici digitali. Inoltre le operazioni su segnali digitali possono essere pensate come operazioni algebriche su particolari entità matematiche dette variabili booleane o logiche; parleremo quindi spesso di "operazioni logiche". Vale la pena di ricordare che comunque il nostro studio sui sistemi logici è finalizzato alla comprensione dei circuiti digitali.
Esistono in commercio diverse famiglie di componenti elettronici che utilizzano segnali digitali; le varie famiglie si distinguono per i valori di tensioni corrispondenti ai livelli logici: nella famiglia TTL i valori tra 0 e 0.8 Volts corrispondono allo zero, mentre i valori tra 2 e 5 Volts corrispondono all' uno. Nella famiglia ECL le tensioni tra -1.95 e -1.60 Volts indicano lo zero, quelle tra -1.00 e -0.65 Volts l' uno.

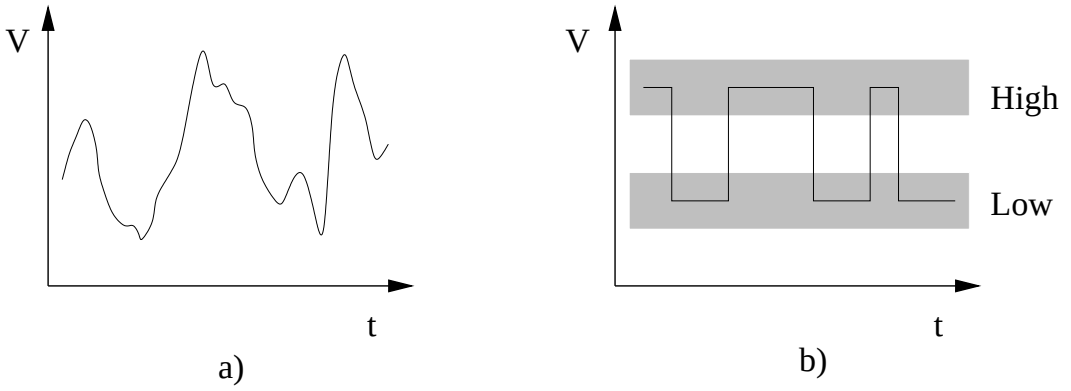


Figura 1.1: Evoluzione temporale di un segnale analogico (a) e di un segnale digitale (b).

### 1.1.1 Algebra di Boole

Un insieme sul quale siano definite due operazioni, somma ($+$) e prodotto ($\times$), commutative e distributive l' una rispetto all' altra, dotate entrambe di elemento neutro e tali che, se $N^{\times}$ è l' elemento neutro del prodotto e $N^{+}$ è l' elemento neutro della somma, esiste per ogni $x$ un elemento $\overline{x}$ ($x$ negato) tale che

$$\overline{x} \times x = N^{+} \quad e \quad \overline{x} + x = N^{\times},$$

si dice algebra di Boole.
È facile dimostrare che l' insieme $\{0, 1\}$ è un' algebra di Boole se si utilizza l' OR come somma e l' AND come prodotto.
In un' algebra di Boole si può dimostrare una relazione di particolare interesse, nota come teorema di De Morgan, e riassunto dalle due relazioni:

$$\overline{x + y} = \overline{x} \times \overline{y}$$

$$\overline{x \times y} = \overline{x} + \overline{y}$$

A parole si può dire che il negato dell' AND è uguale all' OR dei negati, e che il negato dell' OR è uguale all' AND dei negati.
Vista la corrispondenza tra segnali digitali ed elementi dell'algebra di Boole, si utilizza spesso tale formalismo per studiare il comportamento di sistemi digitali. In particolare si realizzano circuiti elettronici che ricevono in ingresso due segnali digitali e producono in uscita un segnale digitale che è l'AND o l'OR degli ingressi. Tali circuti possono essere combinati per realizzare funzioni logiche più complesse.

## 1.2 Rappresentazione dei numeri interi

Abbiamo visto che i segnali digitali possono essere utilizzati per rappresentare variabili logiche, e quindi i sistemi digitali possono trovare utilizzo in dispositivi di controllo nei quali il processo da eseguire sia controllato dal verificarsi o meno di determinate condizioni.
Un' altra applicazione importante dei segnali digitali è la rappresentazione dei numeri interi. In questo contesto i sistemi digitali possono essere utilizzati per realizzare operazioni matematiche.

### 1.2.1 Numerazione binaria in complemento a due

Quando scriviamo un numero intero sottointendiamo che ne realizziamo una rappresentazione in base 10. Questo significa che il numero che scriviamo più a destra è il coefficiente che moltiplica $10^0$, il successivo è il coefficiente di $10^1$ e così via; i coefficienti sono simboli che rappresentano i numeri interi da 0 a 9. Se indichiamo con l' indice $_{10}$ la rappresentazione in base 10 avremo quindi:

$$abcd_{10} = a \cdot 10^3 + b \cdot 10^2 + c \cdot 10^1 + d \cdot 10^0 \; a, b, c, d \in \{0, ..., 9\}$$

Ovviamente la rappresentazione dei numeri interi si può realizzare in una base qualunque: se vogliamo utilizzare segnali digitali per rappresentare i numeri interi la scelta ovvia sarà quella di utilizzare come base 2. Avremo bisogno di due soli simboli, per lo zero (0) e per l' uno (1), e la corrispondenza tra un numero e la sua rappresentazione sarà data da:

$$abcd_2 = a \cdot 2^3 + b \cdot 2^2 + c \cdot 2^1 + d \cdot 2^0 \; a, b, c, d \in \{0, 1\}$$

Ad esempio:

$$10011_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 =$$

$$1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 =$$

$$19_{10}$$

È facile convincersi del fatto che utilizzando $N$ cifre binarie (o, se preferite, $N$ segnali digitali) si possono rappresentare $2^N$ numeri interi, quelli compresi tra 0 e $2^N - 1$. Questo significa che utilizzando 8 linee (8 bits) si possono rappresentare i numeri tra 0 e 255; con 32 bits si va da 0 a $2^{32} - 1 = 4\ 294\ 967\ 295$

### 1.2.2 Numerazione esadecimale

La numerazione esadecimale, cioè in base sedici, è molto utilizzata in ambito informatico. Infatti è molto più compatta della numerazione binaria (servono meno cifre per rappresentare lo stesso numero, il che aiuta a ridurre gli errori e a migliorare la leggibilità), e possiede la proprietà che a ogni cifra esadecimale corrispondono 4 cifre binarie, per cui la conversione binario-esadecimale risulta molto semplice. La numerazione esadecimale richiede 16 cifre, che sono i numeri da 0 a 9 e le lettere da $A$ a $F$ che rappresentano i numeri da $10_{10}$ a $15_{10}$.
Consideriamo ad esempio il numero binario composto dalle cifre $b_0...b_N$, che indicheremo con $(b_N...b_1b_0)_2$. Scriveremo

$$(b_N...b_1b_0)_2 = ... + b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0 =$$

$$... + (b_7 2^3 + ... + b_4 2^0)16^1 + (b_3 2^3 + ... + b_0 2^0)16^0 =$$

$$... + e_1 16^1 + e_0 16^0 = (e_M...e_1e_0)_{16}$$

il che dimostra come le quattro cifre più a destra (meno significative) in rappresentazione binaria corrispondono alla cifra esadecimale meno significativa, e così via.

### 1.2.3 Rappresentazione dei numeri negativi

Il modo più semplice per rappresentare i numeri negativi utilizzando segnali digitali potrebbe essere quello di utilizzare $N$ bits per il modulo, e un bit per il segno. Il problema di questa tecnica è che richiede procedure diverse per eseguire le operazioni aritmetiche con i numeri positivi e con quelli negativi; ciò, in termini elettronici significa maggiore complessità circuitale e quindi maggiori costi.
Una seconda possibilità, che è quella utilizzata di solito nei computers, è data dalla cosiddetta rappresentazione in complemento a due. Questa tecnica consiste nell' utilizzare i numeri positivi $2^N - n$ per rappresentare i numeri negativi $-n$ ($N$ indica il numero di bits utilizzati, o, come si dice la lunghezza della parola). Ovviamente bisogna fare attenzione a non confondere numeri positivi con i numeri negativi associati; si usa questa convenzione: i numeri con il bit più significativo (l' $N$-mo) a 1 sono negativi, gli altri sono positivi. La sequenza dei numeri rappresentabili su $N$ bits sarà quindi:

$$
\begin{array}{lcccc}
0111...111 & = & +(2^{N-1}-1) & & \\
0111...110 & = & +(2^{N-1}-2) & & \\
0111...101 & = & +(2^{N-1}-3) & & \\
0111...100 & = & +(2^{N-1}-4) & & \\
 & ... & & & \\
0000...010 & = & +2 & & \\
0000...001 & = & +1 & & \\
0000...000 & = & 0 & & \\
1111...111 & = & -1 & \equiv & 2^N-1 \\
1111...110 & = & -2 & \equiv & 2^N-2 \\
1111...101 & = & -3 & \equiv & 2^N-3 \\
 & ... & & & \\
1000...010 & = & -2^{N-1}+2 & \equiv & 2^N-2^{N-1}+2 \\
1000...001 & = & -2^{N-1}+1 & \equiv & 2^N-2^{N-1}+1 \\
1000...000 & = & -2^{N-1} & \equiv & 2^N-2^{N-1}
\end{array}
$$

Usando la rappresentazione in complemento a due su $N$ bits potremo quindi rappresentare i numeri da $-2^{N-1}$ a $+2^{N-1}-1$.

### 1.2.4 Operazioni aritmetiche in complemento a due

Una delle buone proprietà della rappresentazione in complemento a due consiste nel fatto che le operazioni di somma e sottrazione si eseguono con le stesse regole su numeri positivi e negativi. Consideriamo infatti due numeri $p$ e $q$ tali che $|p| > |q|$. Se $p$ e $q$ sono entrambi positivi definiremo come regola per l' addizione quella che, data la rappresentazione binaria dei due numeri fornisce la rappresentazione di $p+q$.
Se ora consideriamo $p < 0$ (rappresentato da $2^N - |p|$) e $q > 0$ otterremo, utilizzando la regola per i numeri positivi $2^N - |p| + q = 2^N - (|p| - |q|)$ che è appunto la rappresentazione in complemento a due del numero negativo $p + q = -(|p| - |q|)$
Se invece $p > 0$ e $q < 0$ l' utilizzo della medesima regola fornisce come risultato $p + 2^N - |q| = 2^N + (p - |q|)$. Questo numero binario è diverso dalla rappresentazione del numero positivo $p+q$, per il quale ci aspetteremmo una rappresentazione del tipo $p - |q|$. Osserviamo però che se si utilizzano $N$ bits $2^N + x$ risulta indistinguibile da $x$, in quanto $2^N$ è rappresentato in base due da un "1" seguito da $N$ "0", di modo che l' "1" risulta invisibile (è come dire che in base 10, se si utilizzano solo tre cifre $1000 + x$ ha la stessa rappresentazione di $x$).

Identicamente, nel caso $p < 0$ e $q < 0$ si ottiene come risultato $2^N + 2^N - (|p| + |q|)$ che è indistinguibile, su $N$ cifre, dalla rappresentazione corretta di $p + q$ che in questo caso è $2^N - (|p| + |q|)$.
Ovviamente quando si eseguono operazioni aritmetiche avendo a disposizione solo un numero limitato di cifre si deve fare attenzione a che il risultato sia ancora correttamente scrivibile nello stesso numero di cifre. Un dispositivo sommatore deve quindi essere in grado di fornire, oltre al risultato, anche due segnali digitali che indichino se si è verificato un "overflow", ovvero se la somma di due numeri positivi fornisce un risultato troppo grande, o un "underflow", ovvero la somma di due numeri negativi è troppo piccola (non c' è mai problema con la somma di un numero positivo e uno negativo)(vedi Fig. 1.2).

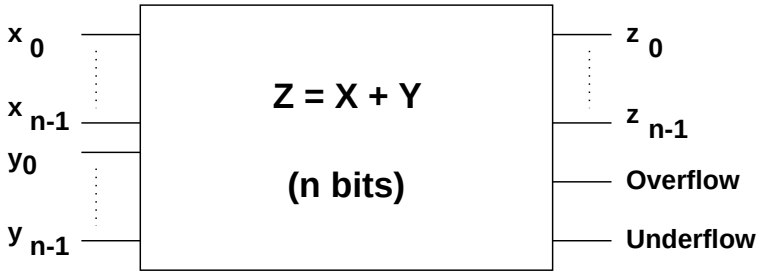


Figura 1.2: Schema di un sommatore binario.

Il segnale di overflow e di underflow possono essere ottenuti facilmente, se si osserva che un overflow si ha quando la somma di due positivi (con il bit più significativo a zero) fornisce un risultato con il bit più significativo a uno, mente l' underflow si ha quando la somma di due negativi (con il bit più significativo a uno) fornisce un risultato con il bit più significativo a zero.
Un' altra osservazione utile nella pratica riguarda la trasformazione di $x$ in $-x$. Si dimostra facilmente (fatelo per esercizio) che per ottenere la rappresentazione in complemento a due di $-x$ è sufficiente eseguire un NOT su tutti i bits della rappresentazione di $x$ ed aggiungere $+1$ al risultato.

# Capitolo 2

# Sistemi a microprocessore

## 2.1 Struttura di un sistema a microprocessore

Dopo aver compreso come i segnali digitali possano essere utilizzati per rappresentare i numeri, possiamo passare ad una descrizione strutturale (cioè non dettagliata, ma solo basata sui blocchi funzionali) dei sistemi a microprocessore, che sono sistemi di notevole complessità dedicati alla elaborazione numerica.

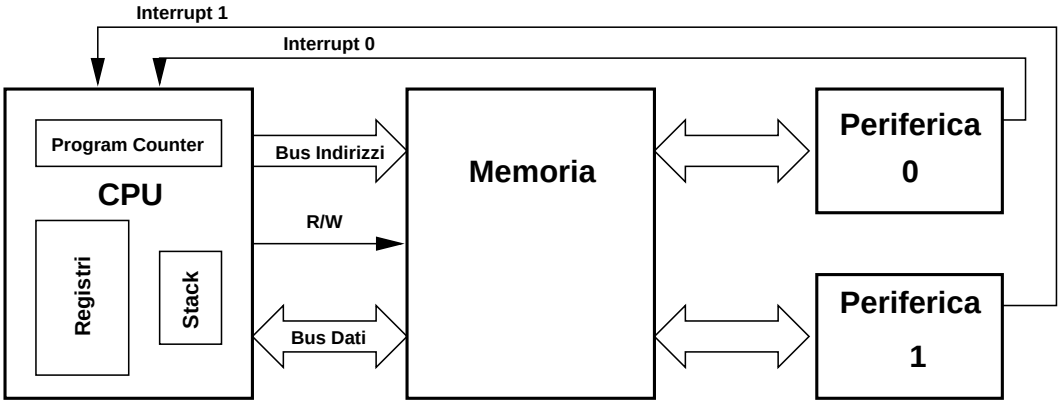


Figura 2.1: Schema a blocchi di un sistema a microprocessore.

Con il termine "sistema a microprocessore" si intende un sistema programmabile basato su una unità di calcolo, detta CPU, e su una memoria (Fig. 2.1). I sistemi a microprocessore sono da considerare come il "cuore" di ogni computer, anche se va detto che un qualunque PC, come pure una qualunque console per videogiochi, posseggono una grande quantità di dispositivi che coadiuvano il sistema CPU-Memoria, dei quali non ci occuperemo minimamente.

### 2.1.1 La memoria

La memoria può essere pensata come un insieme di elementi ciascuno capace di memorizzare un bit; tali elementi sono organizzati in $M$ parole da $N$ bits cadauna. Ogni parola possiede un proprio indirizzo, dato dalla sua posizione, da 0 a $M-1$, tramite il quale è possibile effettuare una operazione di lettura o scrittura. Nelle RAM (Random Access Memories), ogni singola parola può essere letta o scritta individualmente, e non esistono restrizioni sull' ordine delle

operazioni effettuabili. Le RAM sono le memorie usate tipicamente nei computers; si utilizzano parole da 32 o 64 bits, e le capacità complessive sono dell' ordine di 100-1000 milioni di parole. Per potere accedere alla memoria è necessario innanzitutto specificare su quale delle $M$ parole si vuole lavorare: per fare questo si deve scrivere il numero binario corrispondente all' indirizzo su un insieme di $m$ linee digitali ($M = 2^m$) che prendono il nome di "bus di indirizzi". Bisognerà pure specificare se si vuole eseguire una operazione di scrittura della parola o di lettura del contenuto attuale, per cui sarà necessaria una ulteriore linea digitale, detta "Read/Write" o "R/W". A questo punto, se si è richiesta una lettura si otterranno in uscita dalla memoria gli $N$ bits della parola prescelta su $N$ apposite linee digitali; se invece si è richiesta una scrittura si dovranno fornire, sulle medesime $N$ linee, i valori degli $N$ bits che si vogliono scrivere nella parola richiesta. L' insieme delle $N$ linee si dice "bus di Input/Output" o "bus dati", ed ha la caratteristica peculiare di essere un bus bidirezionale; questo significa che all' interno della memoria sono collegati a queste linee gli ingressi di $N$ amplificatori e le uscite di altri $N$ amplificatori: a seconda che sia selezionata l' operazione di scrittura o quella di lettura un solo set di $N$ amplificatori risulterà attivo.
Ricordiamo infine che la memoria è un dispositivo sincrono, e di conseguenza i segnali di ingresso e uscita sono sincronizzati con i fronti di salita di un clock; ad esempio, se all' istante $t_n$ si presentano in ingresso l' indirizzo della locazione da leggere ed il segnale di "Read" si ottengono in uscita sul bus dati, al tempo $t_{n+1}$ i valori memorizzati nella parola prescelta.

### 2.1.2 La CPU

La CPU (Central Processing Unit) è un dispositivo capace di eseguire operazioni sulle informazioni contenute nella memoria. Oltre alla possibilità di leggere e scrivere parole delle memoria, la CPU può spostare dati dalla memoria ai registri interni, ovvero ad un insieme limitato (di solito compreso tra 10 e 1000) di parole di memoria interne alla CPU che possono quindi essere lette o scritte molto rapidamente; inoltre la CPU è capace di eseguire operazioni aritmetiche (di solito addizione, sottrazione, moltiplicazione divisione e confronto) tra numeri interi (in rappresentazione naturale o in complemento a due) o reali (in rappresentazioni a virgola mobile, descritte al paragrafo 2.1.7).

### 2.1.3 Il "linguaggio macchina"

Ma come fa la CPU a sapere quali operazioni deve eseguire, e su quali dati contenuti in memoria? La CPU deve eseguire un programma: a ognuna delle istruzioni eseguibili dalla CPU è associato a un codice numerico, che quindi può essere scritto nella memoria. All' interno della CPU esiste un particolare registro, detto "Program Counter", che contiene, ad ogni istante, l' indirizzo in memoria dell' istruzione da eseguire. La CPU legge dalla memoria il codice numerico relativo all' istruzione da eseguire e la esegue; l' esecuzione può comportare uno o più accessi alla memoria per la lettura e/o la scrittura dei dati relativi all' operazione. A operazione concluda il Program Counter viene incrementato e la CPU comincia l' esecuzione della successiva istruzione.
Il programma consiste quindi in una serie di codici numerici scritti in locazioni consecutive della memoria; ovviamente oltre ai codici vengono scritti in memoria i parametri relativi alle istruzioni (ad esempio, per un' istruzione del tipo "copia una locazione di memoria in un registro" saranno necessari dei parametri: l' indirizzo di memoria da copiare e il numero del registro in cui copiarlo).
Sebbene la cosa non sia strettamente necessaria, la memoria di un computer viene convenzionalmente divisa in due parti, una riservata ai programmi ed una riservata ai dati.

### 2.1.4 Salti, subroutines

Oltre alle istruzioni che comportano l' esecuzione di una operazione ci sono operazioni che implementano salti; quando una di queste istruzioni viene eseguita, nel Program Counter viene scritto l' indirizzo dell' istruzione che rappresenta la destinazione del salto (vedi Fig. 2.2 (a)), di modo che l' esecuzione del programma riprenderà da quel punto. I salti possono essere incondizionati o condizionati al verificarsi di particolari condizioni (ad es. solo se il contenuto di due registri è uguale); in questo modo le istruzioni di salto possono essere utilizzate per implementare le strutture di tipo IF-THEN-ELSE e i cicli DO-WHILE.

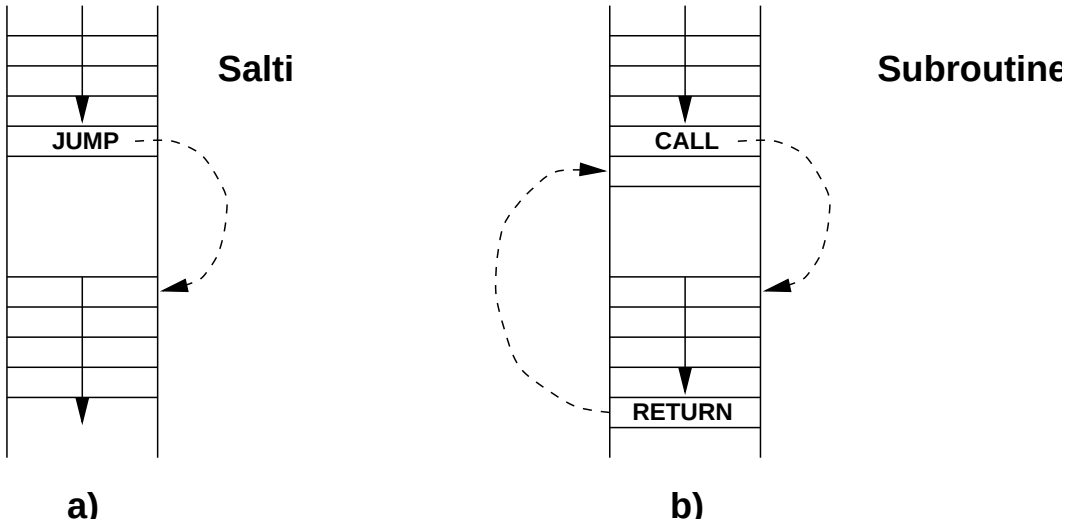


Figura 2.2: Modo di funzionamento dei salti (a) e delle subroutines (b).

Le subroutines invece sono trasferimenti temporanei del flusso di esecuzione da un punto ad un altro del programma, seguito poi dal ritorno alla posizione di partenza (Fig. 2.2 (b)). Quando la CPU incontra una istruzione di salto a subroutine salva in una memoria interna alla CPU, detta "stack" il valore del Program Counter; quindi esegue un normale salto alla destinazione specificata come parametro. Quando incontra una speciale istruzione, usualmente chiamata "Return" la CPU rilegge dallo "stack" il valore del program counter iniziale e riprende l' esecuzione da quel punto.
Lo stack è una memoria particolare, detta "LIFO" (Last In First Out); a differenza della RAM, una LIFO non possiede un bus di indirizzi, ma è, come dice il nome, una pila: si può solo scrivere una parola un cima alla pila, e quando si legge, si legge la parola che è posta in cima alla pila. Facendo in questo modo è possibile concatenare diverse chiamate a subroutine; ad ogni chiamata la CPU metterà l' indirizzo di ritorno in cima allo stack, spostando in basso di una posizione gli altri indirizzi eventualmente presenti; ad ogni istruzione di return la CPU estrarrà dalla cima dello stack l' ultimo indirizzo scritto, e quindi riprenderà l' esecuzione dall' istruzione successiva all' ultima chiamata a subroutine eseguita.

### 2.1.5 Periferiche e interrupts

Resta da chiarire in quale modo la CPU possa entrare in comunicazione con il mondo esterno, ad esempio per caricare i programmi da eseguire o i dati da elaborare, o per comunicare i risultati delle operazioni eseguite. In generale bisogna capire come la CPU possa interagire con i

dispositivi che vengono definiti "periferici" (ad esempio il monitor, la tastiera, i dischi magnetici, le interfacce con le reti di trasmissione dati etc.).
La tecnica utilizzata per la comunicazione con i dispositivi periferici si chiama mappatura in memoria: ad ogni dispositivo viene riservata una porzione (di solito relativamente piccola) della memoria, che funge da area di scambio di informazioni tra la CPU e quel particolare dispositivo. In particolare la CPU indicherà, attivando gli opportuni bits nell' area di interfaccia quale operazione il dispositivo deve eseguire; quando il dispositivo avrà terminato l' operazione, o comunque quando avrà informazioni da trasferire alla CPU, attiverà una "linea di interrupt". Si tratta di un segnale digitale che, quando attivato, produce lo stesso effetto di una chiamata a subroutine ad un indirizzo particolare; a tale indirizzo verrà posto un programma (specifico per ogni periferica) che gestisce il dialogo con la CPU e, al termine, restituisce il controllo al programma che era in esecuzione al momento in cui l' interrupt è stato generato. Tale programma si chiama routine di interrupt o "driver" del dispositivo.

### 2.1.6 Il sistema operativo

Come è facile intuire, la gestione dei dispositivi periferici con la tecnica delineata al paragrafo precedente è piuttosto complessa, e, soprattutto, ha il notevole svantaggio di richiedere soluzioni specifiche per ogni tipo di dispositivo e per ogni possibile configurazione (ad esempio se un computer possiede due dischi dovranno essere riservate due diverse porzioni di memoria per i due dischi, e il driver relativo dovrà sapere su quale porzione agire in corrispondenza di ogni operazione).
Di conseguenza i computer sono dotati di un programma speciale, chiamato "sistema operativo", che ha lo scopo di gestire il dialogo con le periferiche e di fornire all' utente un' interfaccia standard (cioè indipendente dai dettagli funzionali e di configurazione) con i dispositivi esterni. In questo senso il sistema operativo svincola chi scrive codice eseguibile dalla conoscenza dei dettagli dei dispositivi installati in un particolare computer e consente quindi di realizzare programmi che possono essere più facilmente trasportati da una macchina ad un' altra.
Un' altra importante funzione del sistema operativo è quella di consentire a più programmi contemporaneamente di utilizzare la CPU. Una parte del sistema operativo, detto "scheduler", mantiene una tabella contenente le informazioni su tutti i programmi indipendenti che devono essere eseguiti dalla CPU (vengono detti "processi"); dalla tabella lo scheduler estrae il processo che deve essere eseguito per primo (perché ha maggiore priorità o perché non viene eseguito da molto tempo), regola un timer esterno alla CPU in modo che generi un interrupt dopo un intervallo di tempo prefissato e fa partire l' esecuzione del processo scelto; quando il timer genererà l' interrupt lo scheduler riprenderà il controllo e selezionerà un nuovo processo da eseguire. In questa maniera molti programmi diversi possono, utilizzando a turno la CPU, essere portati a termine contemporaneamente.

### 2.1.7 Codifica dei dati in memoria

Nel corso della descrizione dei sistemi a microprocessore abbiamo avuto modo di osservare come nelle parole della memoria possano essere scritte informazioni di tipo diverso utilizzando differenti codifiche. Accanto alla codifica binaria naturale, adatta ai numeri interi positivi, abbiamo visto che si può utilizzare la codifica in complemento a due per i numeri interi con segno.
I numeri reali non possono ovviamente essere rappresentati in quanto tali su un sistema con precisione finita come è, per definizione un insieme di $n$ bits, con $n$ comunque grande. Quello che si potrebbe fare sarebbe trasformare un reale in un intero moltiplicandolo per $10^a$ e troncando le cifre decimali rimanenti. Si avrebbero così $a$ cifre di precisione. Però in questo modo si

possono rappresentare intervalli piuttosto limitati. Si preferisce quindi utilizzare una notazione a precisione variabile, o "a virgola mobile": si scrive un numero reale $x$ come $y \cdot 10^a$, dove $a$ è un intero con segno e $y$ un reale compreso tra -1 e 1. $y$ (la mantissa) può essere rappresentato (con precisione finita) da un intero; l' esponente è di per sé intero, e quindi facile da rappresentare. Con questa tecnica si possono rappresentare numeri reali su intervalli piuttosto ampi (ad esempio, su 32 bits, si possono rappresentare numeri con esponente compreso tra -38 e +38 e otto cifre significative sulla mantissa).
Un' altra rappresentazione che abbiamo visto è la codifica numerica delle istruzioni eseguibili dalla CPU. In realtà esistono altre codifiche standard; ad esempio ad ogni lettera o simbolo di interpunzione viene attribuito un codice numerico di 8 bits, detto codice ASCII, che consente di scrivere in memoria testi alfanumerici. Altre codifiche possono essere inventate caso per caso per consentire una efficiente memorizzazione di dati specifici.
Si deve porre sempre molta attenzione, quando si devono scambiare dati tra parti diverse di un programma o tra computers diversi, ad assicurarsi che chi legge e chi scrive siano d' accordo sulla codifica utilizzata per rappresentare in memoria i dati da elaborare.

### 2.1.8 Vantaggi e svantaggi dei linguaggi macchina

I codici numerici associati alle istruzioni eseguibili dalla CPU ed i parametri ad esse associate vengono detti "linguaggio macchina". Come il nome suggerisce, il linguaggio macchina è specifico di ogni tipo di CPU: CPU diverse hanno linguaggi macchina diversi.
Per di più i linguaggi macchina hanno lo svantaggio di potere utilizzare soltanto gli indirizzi numerici delle locazioni di memoria per accedere ai dati o alle periferiche o alle subroutines; in altri termini, se si vuole memorizzare un dato in memoria non è possibile assegnargli un nome, ma solo un numero corrispondente alla locazione da esso occupata.
Per questi motivi si cerca di evitare di programmare in linguaggio macchina, e si cerca di demandare il compito della generazione del codice eseguibile dalla CPU a programmi dedicati.

# Capitolo 3

# I linguaggi evoluti

## 3.1 introduzione

I linguaggi evoluti sono insiemi di regole sintattiche che consentono di specificare operazioni eseguibili da una generica CPU. Rispetto ai linguaggi macchina, i linguaggi evoluti hanno il vantaggio di rendere possibile l' utilizzo di nomi simbolici per le variabili e per le subroutines; inoltre mettono a disposizione del programmatore un insieme di istruzioni elementari molto più ampio di quello disponibile in un normale linguaggio macchina, ed hanno l' ulteriore vantaggio di essere totalmente indipendenti nella sintassi e nella funzionalità dal tipo di CPU utilizzata. Quindi, grazie all' utilizzo di linguaggi evoluti e di sistemi operativi diventa possibile scrivere programmi in modo del tutto indipendente dal tipo di computer utilizzato.
Naturalmente, siccome alla fine ogni particolare CPU non puó che eseguire programmi scritti nel suo linguaggio macchina, dovrà essere svolta una operazione di traduzione che porti dal programma scritto in linguaggio evoluto al codice eseguibile scritto in linguaggio macchina.

## 3.2 Compilazione e Link

L' operazione di traduzione di un programma in linguaggio evoluto in uno in linguaggio macchina si chiama compilazione, e viene eseguita per mezzo di un programma specifico detto "compilatore". I compilatori sono specifici, oltre che del linguaggio evoluto utilizzato, anche del tipo di CPU utilizzata, della quale devono conoscere il linguaggio macchina; sono invece di solito indipendenti dai dettagli di configurazione del computer utilizzato.
Il codice in linguaggio macchina generato da un compilatore non è tuttavia eseguibile dalla CPU, e prende il nome di "codice oggetto". Il codice oggetto è in effetti solo la traduzione in linguaggio macchina del programma effettivamente scritto dal programmatore in linguaggio evoluto: mancano tutte quelle routines che permettono di realizzare in linguaggio macchina le operazioni elementari in linguaggio evoluto, come pure mancano tutte le routines specifiche del sistema operativo che rendono possibile il dialogo con le periferiche. Ad esempio, in qualunque linguaggio evoluto la stampa sul monitor del valore di una variabile reale chiamata `A` si può eseguire con una semplice istruzione del tipo `PRINT A`. Dal punto di vista della CPU questo significa leggere la memoria alla locazione corrispondente alla variabile `A`, decodificarla tenendo conto della codifica utilizzata, ricavare una rappresentazione corrispondente in base 10, scriverla su una stringa e passare questa stringa al driver del monitor perché la visualizzi. Queste operazioni vengono eseguite da routines specifice del linguaggio e del sistema operativo che devono essere prelevate da opportune librerie. L' operazione di completamento del codice oggetto con le

opportune routines prelevate dalle librerie di linguaggio o di sistema operativo si chiama "link". Durante tale operazione possono essere aggiunte al codice oggetto anche routines prelevate da librerie utente.
Osserviamo infine che nel sistema operativo UNIX le operazioni di compilazione e link vengono di solito eseguite dallo stesso comando (**g++** per il C++); è però possibile, usando opportune opzioni dei comandi (vedere Par.5.11.2), eseguirle separatamente.

## 3.3 Elementi di un linguaggio evoluto

In un linguaggio evoluto si possono distinguere diverse componenti:

- Sintassi: indica l' insieme delle regole generali che governano la scrittura del programma.
- Struttura: indica il modo in cui è possibile suddividere un programma complesso in sottoprogrammi e fornisce le regole per far comunicare tra loro i sottoprogrammi.
- Variabili: sono le regole per la definizione e l' utilizzo delle aree di memoria che contengono i dati da elaborare.
- Controllo di Flusso: sono le regole per eseguire test, clicli etc.
- Funzioni: sono i comandi per eseguire operazioni complesse, sia di tipo aritmetico che di input/output.

Nel seguito ci occuperemo degli aspetti generali (cioè sostanzialmente indipendenti dal linguaggio) di variabili e struttura. Sintassi, Controllo di Flusso e Funzioni sono invece specifici dei diversi linguaggi; per il C++ saranno trattati al Capitolo 5

## 3.4 Le variabili

Una variabile è una porzione di memoria a cui il compilatore assegna un nome. È importante ricordare che il nome ha un senso solo nell' ambito del programma in linguaggio evoluto; dopo la compilazione resteranno solo gli indirizzi numerici di memoria, ed il programma eseguibile non saprà più nulla dei nomi. Soltanto nel caso la compilazione venga eseguita con l' opzione **-g** al codice eseguibile verrà aggiunta una tabella di corrispondenza nome-indirizzo che consentrà al "debugger simbolico" (Par 4.9) di accedere, nel corso dell' esecuzione del programma, alle variabili utilizzando il loro nome.
Nel seguito esamineremo le caratteristiche principali delle variabili.

### 3.4.1 Tipo e dimensione di una variabile

Si dice "tipo" di una variabile il metodo di codifica utilizzato per scrivere i dati in memoria. Come abbiamo visto esistono codifiche adatte ai numeri interi, ai numeri reali, alle variabili alfanumeriche e così via; i diversi linguaggi utilizzanno diverse parole chiave per indicare lo stesso tipo di codifica; si veda Par. 5.4.1 per i nomi utilizzati dal C++.
Il tipo di codifica utilizzato determina anche la lunghezza in bits di un elemeto di quel tipo: ad esempio un reale in singola precisione è scritto su 32 bits, ovvero su 4 bytes, mentre un reale in doppia precisione occupa 64 bits, ovvero 8 bytes. Oltre alla lunghezza di un singolo elemento i linguaggi evoluti consentono di creare vettori e matrici di singoli elementi di un dato tipo. La dimensione di una variabile sarà quindi data dal numero complessivo di elementi del vettore

o della matrice; vale la pena di ricordare che la lunghezza totale occupata in memoria da una variabile sarà data dal prodotto della dimensione per la lunghezza in bytes del singolo elemento.

### 3.4.2 Posizione in memoria degli elementi di una matrice

Gli elementi di un vettore vengono sempre posti in memoria in locazioni consecutive. Per quel che concerne le matrici e i vettori a più indici invece la disposizione degli elementi in memoria è meno ovvia; in particolare in C++ la disposizione degli elementi avviene nell' ordine $i[0][0]$, $i[0][1]$, ..., $i[0][n-1]$, $i[1][0]$, .... L' ordine esatto della disposizione delle matrici in memoria è importante, in quanto molto spesso i computers accedono molto più velocemente a blocchi contigui di memoria che non a elementi sparsi; quindi quando si realiza un loop che legge o scrive gli elementi di un vettore è bene cercare di assecondare l' ordine naturale della posizione dei singoli elementi: basterà ricordare che in C++ devono scorrere più velocemente gli indici più a destra.

### 3.4.3 Tipi composti

I tipi composti sono agglomerati di variabili standard (sia scalari che vettori o matrici). Una volta definiti (vedere Par. 5.12.1), i tipi composti possono essere utilizzati, come quelli predefiniti, per definire variabili scalari, vettori o matrici. Pure una variabile appartenente a un tipo composto può essere passata come argomento ad una subroutine specificandone semplicemente il nome: per questo motivo i tipi composti consentono di semplificare molto la scrittura dei programmi ed il trasferimento di dati tra diverse routines e tra i programmi ed i files di dati. Nei linguaggi "object oriented" (come il C++), è inoltre possibile ridefinire il comportamento degli operatori sui tipi composti. Oltre a questo è ovviamente possibile utilizzare operatori tradizionali sui singoli campi della struttura.

## 3.5 Allocazione statica ed automatica della memoria

Con allocazione della memoria si indica l' operazione che il compilatore esegue quando riserva lo spazio di memoria necessario per ospitare una variabile.
L' allocazione statica viene eseguita al momento della compilazione, e di conseguenza non può essere alterata nel corso dell' esecuzione. Questo metodo di allocazione viene usato normalmente dai compilatori per le variabili globali di un programma: queste variabili vengono allocate una volta per tutte e rimangono nella medesima posizione di memoria durante tutta l' esecuzione del programma. In questo modo possono essere utilizzate con sicurezza da tutte le componenti del programma.
L' allocazione statica è svantaggiosa per variabili che non sono necessarie durante tutto lo svolgimento del programma, ma solo durante fasi particolari, in quanto mantiene lo spazio relativo occupato inutilmente. In questi casi viene usata l' allocazione automatica, che consiste nel riservare lo spazio in memoria al momento in cui la variabile deve essere utilizzata (durante l' esecuzione del programma quindi) e nel rilasciarlo quando la variabile non serve più. Questa tecnica è utilizzata normalmente per le variabili locali delle subroutine, che vangono allocate al momento della chiamata della routine e buttate via quando la routine termina l' esecuzione. Di conseguenza non si deve mai fare affidamento sul fatto che una variabile locale conservi il suo valore tra una chiamata e la successiva, a meno che non si utilizzino dichiarazioni particolari (Par. 5.4.3) che forzano l' allocazione statica anche per le variabili locali.

## 3.6 I puntatori

I metodi di allocazione che abbiamo visto fino ad ora sono eseguiti e comandati dal compilatore, vuoi in fase si compilazione (allocazione statica) vuoi in fase di esecuzione (allocazione automatica). In alcuni casi tuttavia è necessario poter intervenire direttamente sul meccanismo di allocazione delle variabili: il caso tipico è quello che si presenta quando si devono disporre $n$ dati in un vettore, ma $n$ non è noto al momento della compilazione ma viene deciso volta per volta al momento dell' esecuzione del programma. In questo caso si vorrebbe decidere la lunghezza del vettore, e quindi eseguire l' allocazione, al momento dell' esecuzione.
Lo strumento che consente di intervenire sul processo di allocazione è il "puntatore". Si tratta di una normale variabile, di solito allocata staticamente, che contiene, invece di un dato, l' indirizzo in memoria di un' altra variabile.

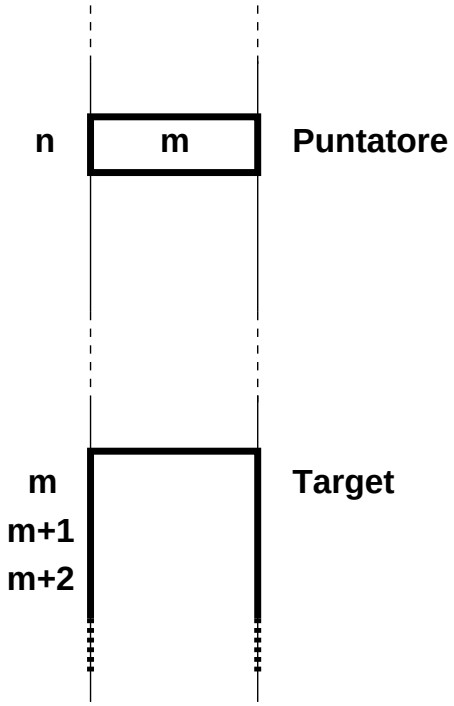


Figura 3.1:

Quamdo si accede ad una variabile per mezzo di un puntatore il sistema prima legge il contenuto del puntatore, quindi lo utilizza per trovare la posizione in memoria sulla quale effettuare l' operazione richiesta. La variabile "puntata" dal puntatore viene detta "target".

### 3.6.1 Allocazione dinamica della memoria

Grazie all' utilizzo dei puntatori e di speciali routines del sistema operativo che consentono, nel corso dell' esecuzione di un programma, di cercare aree di memoria libere e di riservarle ad un particolare uso, è possibile realizzare la allocazione dinamica delle variabili in memoria.
La sola operazione necessaria in fase di compilazione consiste nel definire un puntatore. Durante l' esecuzione del programma, una volta deciso quanto spazio è necessario per la variabile che vogliamo allocare, dovremo solo chiedere al sistema operativo di trovare una zone di memoria sufficientemente grande e dovremo scrivere nel puntatore l' indirizzo iniziale di questa zona. Ciò

fatto potremo accedere alla nostra variabile tramite il puntatore come se si trattasse di una variabile allocata in modo statico; la variabile sarà locale o globale a seconda che il puntatore, e quindi l' indirizzo in esso contenuto, siano leggibili o no da diverse routines. Quando la nostra variabile non ci servirà più potremo rilasciare lo spazio ed azzerare il puntatore, o eventualmente potremo riutilizzare il puntatore per puntare a un target di dimensione differente: in questo modo sarà possibile, ad esempio, incrementare progressivamente, secondo le esigenze che si presentano nel corso dell' esecuzione, la dimensione di un vettore allocato dinamicamente.
Malgrado gli enormi vantaggi di versatilità e flessibilità che presenta, l' allocazione dinamica scarica completamente sul programmatore la responsabilità di controllare la corretta gestione della memoria. Per questo motivo va utilizzata solo quando realmente necessario e facendo la dovuta attenzione.
Al Paragrafo 5.9 sono descritte le tecniche usate in C++ per realizzare praticamente l' allocazione dinamica. Riassumiamo qui solo i concetti fondamentali. Il C++ utilizza simboli specifici per il puntatore ed il target (se `p` è un puntatore `*p` è il suo target. I puntatori si comportano all' incirca come variabili intere, per cui le operazioni sui puntatori si possono eseguire (quasi) come su normali interi positivi. Ogni puntatore è specifico del tipo del target (i puntatori a interi sono diversi dai puntatori a float ad esempio) e può essere usato indifferentemente per puntare a scalari o a vettori; nel secondo caso la notazione per l' accesso al vettore è identica a quella che si usa per i vettori allocati staticamente.

## 3.7 Programmazione strutturata

Con il termine "programmazione strutturata" si intende la suddivisione di un programma complicato in sottoprogrammi aventi ciascuno una funzionalità ben definita. Questo metodo di programmazione consente di semplificare notevolmente le fasi di messa a punto di un programma, dal momento che i vari pezzi possono essere provati e messi a punto indipendentemente, anche da persone differenti. Inoltre sottoprogrammi scritti in un particolare contesto possono poi essere riutilizzati, sostanzialmente senza modifiche in situazioni diverse.
Un altro vantaggio della programmazione strutturata è la possibilità di essere applicato in una progettazione del software di tipo top-down, nella quale cioè il progetto complessivo viene suddiviso in parti più semplici, per ciascuna delle quali è definita una "interfaccia" con le altre, e la procedura è ripetuta iterativamente per ciascun sottoprogetto fino ad arrivare a blocchi elementari. Il vantaggio della progettazione top-down è quello di fornire descrizioni schematiche a diversi livelli di progetti complessi, consentendo a chi lavora in un particolare settore e quindi conosce i dettagli di quella parte, di avere delle descrizioni funzionali di massima di tutte le rimanenti componenti senza per altro dover venire a conoscenza di alcun dettaglio realizzativo.
Gli elementi dei linguaggi di programmazione che consentono la realizzazione dei blocchi elementari di un programma strutturato sono le subroutines.

## 3.8 Subroutines

Una subroutine è un insieme di istruzioni che vengono eseguite in sequenza; quando la sequenza termina, o quando viene incontrata un' istruzione specifica detta "return" la subroutine termina e il programma riprende dall' istruzione successiva a quella che ne ha causato l' attivazione (istruzione che appartiene per definizione ad un' altra subroutine). Una subroutine particolare è il "main program". Il main program non viene chiamato da nessuna subroutine, ma viene

attivato al momento in cui il programma parte; quando la sequenza di istruzioni del main program termina anche il programma termina.
Ogni subroutine può definire al suo interno variabili locali. Tali variabili sono accessibili solo alla subroutine che le dichiara e, in assenza di differenti direttive, sono allocate in modo automatico, per cui vanno perdute non appena l' esecuzione della subroutine termina.
Nella logica della programmazione strutturata, le routines devono comunicare tra loro, scambiandosi dati e risultati delle elaborazioni, e questo non può essere fatto usando variabili locali. Ogni subroutine possiede quindi un insieme di parametri, che vengono usualmente specificati tra parentesi dopo il nome; si tratta variabili fittizie o "dummy", per le quali cioè non esiste una allocazione di memoria. Ciò nonostante possono essere utilizzate nell' ambito della subroutine come normali variabili: infatti, al momento della chiamata, il programma chiamante sepcificherà quali delle sue variabili locali devono essere passate come parametri, e quando la subroutine utilizzerà i nomi delle variabili dummy lavorerà in realtà su queste variabili.
In questo modo si può fare sì che due subroutines comunichino tra loro; in più si può ottenere che una subroutine esegua le operazioni per cui è stata progettata su variabili diverse semplicemente passando parametri differenti in successive chiamate: la subroutine non dovrà sapere nulla dei veri nomi delle variabili su cu lavora, in quanto utilzzerà sempre i nomi delle variabili dummy.
Bisogna sempre fare molta attenzione a che il programma chiamante passi come parametri delle variabili identiche, per tipo e dimensione, a quelle specificate nella definizione della subroutine come variabili fittizie; in caso contrario i dati scritti in memoria dal programma chiamante saranno decodificati in modo errato dalla subroutine ed i risultati saranno pure errati.
La sintassi per la definizione e la chiamata delle subroutines è descritta al Par. 5.10.

### 3.8.1 Passaggio dei parametri alle subroutines

In C++ il passaggio dei parametri alle subroutines avviene "by value": per ogni parametro passato alla subroutine viene eseguita una copia, e questa copia è resa accessibile come variabile dummy; il sottoprogramma può quindi leggere i parametri, ma eventuali modifiche verranno eseguite sulla copia, e non saranno quindi visibili al programma chiamante quando la subroutine sarà terminata. Un modo per ovviare a questa difficoltà consiste nel passare come parametro non una variabile ma un puntatore ad essa (passaggio "by reference"): in questo modo verrà eseguita una copia del puntatore, ma la subroutine, leggendo l' indirizzo della variabile, potrà comunque leggerla e modificarne il contenuto. Questa tecnica viene implicitamente usata per i vettori, in quanto in C++ il nome di un vettore non seguito da parentesi quadre indica il puntatore al vettore.

### 3.8.2 Functions

Una variante delle subroutine sono le functions, che si usano di solito per implementare funzioni matematiche o comunque operazioni assimilabili a funzioni matematiche. Le functions posseggono un parametro dummy aggiuntivo, che è associato al nome stesso della function e che viene sempre e solo passato (by value) dalla function al programma chiamante. In questa maniera il nome di una function seguito dalla lista dei parametri posta tra parentesi può essere utilizzato dal programma chiamante all' interno di espressioni aritmetiche o di istruzioni di assegnazione o di test, esattamente come se si trattasse di una normale variabile; tuttavia mentre una normale variabile viene semplicemente letta dalla memoria e usata nell' espressione, una function viene chiamata come una subroutine, le vengono passati i parametri e viene eseguito il codice eseguibile in essa contenuto che termina con la assegnazione della variabile dummy di ritorno; questo valore viene poi sostituito nell' espressione. Di conseguenza l' utilizzo di functions in

espressioni, specie all' interno di cicli ripetuti molte volte, può risultare ben più "CPU time consuming" dell' uso di una normale variabile.
Va notato che in C++ di fatto esistono solo le functions. Una subroutine è vista come una function che ritorna una variabile di tipo `void`.

### 3.8.3 Ricursività

La ricursività è la capacità da parte di una function di richiamare se stessa. Da quello che abbiamo detto sull' allocazione automatica delle variabili locali si intuische che linguaggi come il C++ ammettono la ricursività. In effetti ad ogni chiamata (o auto–chiamata) viene creato un nuovo insieme di variabili locali, e di conseguenza si cancellano le possibili interferenze tra due istanze della stessa function contemporaneamente presenti nello stack delle subroutines. Va però osservato che le variabili globali o le variabili locali statiche sono potenzialmente fonti di interferenza tra chiamate ricursive, e vanno usate con molta attenzione nel caso si implementino algoritmi ricursivi.

### 3.8.4 Moduli e variabili globali

L' utilizzo dei parametri come metodo di comunicazione tra le routines può diventare scomodo se si ha un gruppo di routines che lavorano su un numero elevato di variabili; in questo caso si preferisce ricorrere alle variabili globali. Si tratta di variabili, allocate staticamente, che sono visibili non ad una sola routine ma a gruppi di routines.
Il meccanismo più sicuro per l' implementazione delle variabili globali consiste nella realizzazione di un modulo, ovvero di un gruppo di routines: ogni routine definirà al suo interno le proprie variabili locali, ma tutte le variabili definite entro il modulo ma fuori da ogni routine saranno automaticamente considerate comuni a tutto il modulo. Le regole per la realizzazione dei moduli si trovano al Par. 5.11.1.
Vale la pena di osservare come l' approccio delle variabili globali cancelli l' indipendenza delle subroutines dai nomi delle variabili su cui esse agiscono: i nomi delle variabili globali sono fissi, e quindi una subroutine che agisce su una variabile globale agisce sempre e solo su quella (a meno che la variabile globale non sia un puntatore che viene ridefinito dinamicamente nel corso dell' esecuzione...)
Se le routines singole sono quindi adatte ad implementare blocchi funzionali semplici, che, nell' ambito di un programma possono essere applicati a diversi insiemi di dati, i moduli vanno utilizzati per per implementare blocchi complessi, che agiscono su un' unica base di dati.

### 3.8.5 La programmazione object–oriented

I moduli sono collezioni di procedure che agiscono su una base di dati fissata ed univoca. In alcune circostanze questa limitazione (che pure può essere aggirata con un uso sapiente di puntatori ed allocazione dinamica) risulta fastidiosa. È il caso delle interfacce grafiche a finestre: ogni finestra corrsiponde a dati diversi, ma i programmi di manipolazione sono similari; si desidererebbe quindi un accoppiamento naturale tra dati e subroutines, del tipo che tutte le volte che viene creata la struttura dati corrispondente ad una finestra venisse pure creata una copia delle routines di manipolazione specifiche per quella particolare finestra.
Questa è la logica dei linguaggi di programmazione object-oriented (come ad esempio il C++). In questi linguaggi è possibile definire tipi composti (detti classi) che contengono, oltre che variabili, subroutines e functions. Ogni volta che si crea una variabile del tipo composto (nel linguaggio OO, ogni volta che si istanzia una classe) vengno anche create copie dedicate delle

funzioni. Questo incapsulamento del codice risulta molto utile in tutte quelle circostanze in cui sia necessario, all' interno di un singolo programma, realizzare molte ripetizioni di algoritmi identici o molto simili agenti su basi di dati differenti.

### 3.8.6 Interfacce con librerie di routines

Le routines che appartengono ad uno stesso modulo si conoscono tra loro, e di conseguenza si possono chiamare senza bisogno di particolari definizioni; inoltre il compilatore sarà in grado di controllare se le variabili dummy ed i parametri effettivamente passati sono definiti in modo coerente. Si dice in questo caso che l' interfaccia tra le varie routines di un modulo è definita implicitamente.
In alcuni casi tuttavia si devono chiamare routines per le quali non è disponibile un 'interfaccia implicita; questo accade, ad esempio, quando si utilizzano funzioni che appartengono ad una libreria esterna, e per le quali non si scrive il codice. In tal caso si deve creare una interfaccia esplicita che informi il compilatore del nome e del tipo della funzione come pure del numero e del tipo dei parametri (Par. 5.11.4). In questo modo il compilatore, pur non vedendo il codice della funzione, può controllare se le chiamate sono corrette in termini di numero e tipo dei parametri specificati.

# Capitolo 4

# Il sistema operativo

## 4.1 Livelli di funzionalità di un calcolatore

La figura 4.1 rappresenta la gerarchia dei termini che formano i livelli di funzionalità del PC. Se con hardware indichiamo tutte le parti "fisiche" dell'eleboratore notiamo che questo livello contiene una parte detta firmware che individua la parte, normalmente registrata in una memoria ROM, dove sono presenti microistruzioni necessarie per la fase di avvio (bootstrap) e per il caricamento del sistema operativo.
Il livello più basso è il **linguaggio macchina** (sequenza di numeri binari) che è l'unico linguaggio che la CPU riesce a comprendere.
Non è molto comodo scrivere un programma utilizzando questo linguaggio; il divario tra un linguaggio macchina e un linguaggio più evoluto è colmato dal sistema operativo (OS Operating System).
Un sistema operativo si preoccupa di:

- gestire l'interfaccia utente, cioè di interpretare ed eseguire una serie di comandi comunicati dall'utente tramite tastiera o mouse. Ciò permette di manipolare insiemi di dati (files) e di accedere alle periferiche senza avere una conoscenza approfondita dell'hardware del PC.
- permettere l'utilizzo contemporaneo della CPU da parte di più programmi.
- gestire l'utilizzo della memoria centrale.

Probabilmente molti di voi conoscono i sistemi operativi commerciali più diffusi (DOS, WindowsXX, MAC OS, ...), durante il corso utilizzeremo un sistema operativo completemente gratuito chiamato Linux, che deriva da un sistema operativo creato per workstation (Unix) e adattatato all'architettura dei PC.

## 4.2 Linux

### 4.2.1 Un po' di storia

Linux è una versione per PC del sistema operativo Unix studiato per la gestione delle workstation utilizzate come server (ovvero macchine in grado di gestire un centro calcolo con molti utenti e reti locali).
Il sistema UNIX sviluppato dalla AT&T Bell Laboratories nacque essenzialmente come reazione ai sistemi operativi di grandi dimensioni, complessi e poco flessibili, disponibili alla fine degli

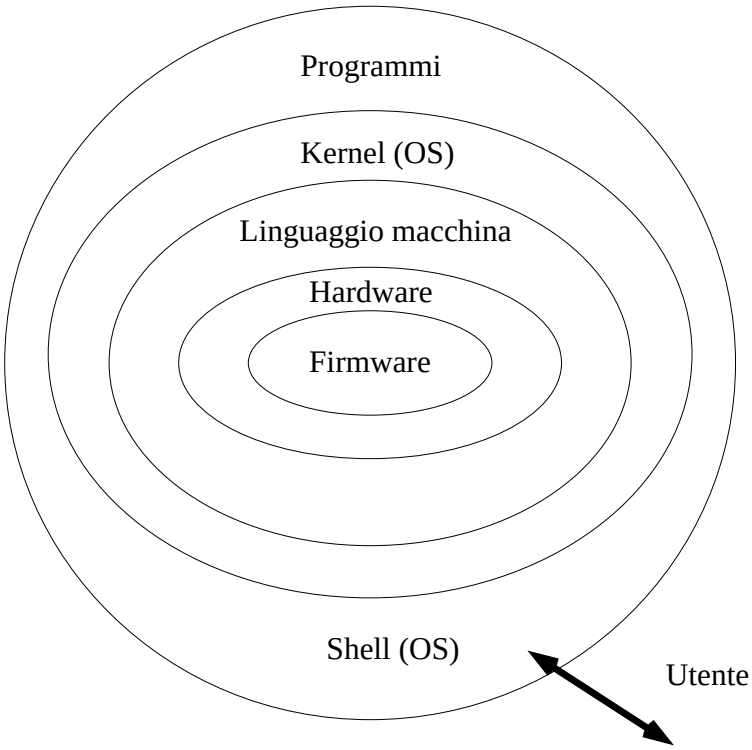


Figura 4.1: Il livelli di funzionalità di un calcolatore

anni sessanta. Il principale obiettivo di progettazione fu quello di ottenere un sistema operativo d'elevata potenzialità.
Originariamente (anni '70) il sistema operativo fu sviluppato da un gruppo di esperti di computer come uno strumento di sviluppo per uso personale, ignorando così le esigenze dei principianti a vantaggio della velocità e della precisione. A causa delle leggi antitrust il software non poteva essere venduto e prese la strada delle Università Americane che a loro volta iniziarono a sviluppare varie versioni di Unix facendolo maturare velocemente ma creando anche un po' di disordine e confusione. In questo modo iniziarono a formarsi una casta di persone che una volta laureate e pronte per il mondo del lavoro iniziarono a divulgare e far conoscere Unix nelle aziende. Pian piano Unix potè diventare un prodotto commerciale e si iniziarono a vedere sul mercato varie versioni proprietarie e quindi a pagamento. Linux è nato come estensione del Minix (sistema operativo Unix per processori i86 sviluppato ad uso didattico dal professor Andrew S. Tanenbaum) e come clone di Unix, dallo studente finlandese dell' Università di Helsinki Linus Beneditc Torvalds, per un progetto universitario sullo studio delle funzionalità multitasking dei microprocessori i386.
Linux viene mantenuto da un gruppo di programmatori sparsi per tutto il mondo e la sua flessibilità e la sua potenza sono accompagnati da un vantaggio che non deve essere trascurato. Linux è completamente gratuito cioè non "si è obbligati" a pagare per poter utilizzare questo sistema operativo. In pratica esiste una licenza GNU ( sigla ricorsiva GNU's not Unix) che assicura che Linux rimanga gratuito ma anche standard.
Uno dei vantaggi principali di Linux è che in esso sono forniti tutti gli strumenti per il calcolo scientifico, come per esempio: editori di testo molto complessi, tutti i linguaggi di programmazione, librerie grafiche etc..

Non è da trascurare il fatto che è un sistema multiutente e multitasking, ovvero più utenti possono utilizzare lo stesso computer (da terminali differenti) e richiedere al sistema di eseguire più operazioni (task) contemporaneamente.
Recentemente le interfacce grafiche per l'utilizzo del sistema operativo sono si fatte sempre più elaborate e "simili" a quelle dei sistemi commerciali.
Per la cronaca, dopo sette anni dalla nascita di Linux, Linus Torvalds si è laureato e dopo aver collaborato per diversi anni come ricercatore all'interno dell'Università di Helsinki si è trasferito in America dove continua, anche se non come attività primaria, a dirigere lo sviluppo del sistema operativo e a lavorare su una versione real-time di Linux.

### 4.2.2 La struttura di Linux: il kernel e la shell

Il sistema operativo è costituito da un sistema di base e da un certo numero di estensioni. Il sistema di base comprende il kernel, le shell e le utility di base. Le estensioni sono ad esempio le funzionalità grafiche, le librerie , i compilatori. Il kernel di Linux, che consiste nell'interfaccia tra l'hardware e i processi, gestisce l'accesso alla CPU e la memoria, e si occupa, fra le altre cose, del controllo e della gestione di ciascuna periferica (device). Rappresenta, quindi, il nocciolo dell'intero sistema operativo e idealmente può assumersi come una sorta di astrazione della macchina fisica essendo posto al livello software più basso.
Per quanto riguarda le versioni del kernel c'e' da dire che esse utilizzano uno specifico schema per la loro numerazione : le versioni V.x.y con x pari rappresentano le versioni stabili, mentre al contrario le versioni con x dispari sono le beta-release (ancora in prova) utilizzate normalmente dagli sviluppatori e possono essere instabili o generare effetti collaterali non conosciuti: mano a mano che si risolvono i bug (bachi!) il numero y viene incrementato.
Bisogna stare attenti a non confondere la versione del kernel con la distribuzione che contiene anch'essa un numero. Esistono infatti diversi distributori di Linux (cambia praticamente solo l'aspetto grafico e alcune utilities) RedHat, Mandrake, Slackware, Suse, Debian, quindi posso avere per esempio la Mandrake 8.2 che fornisce il kernel 2.4.3-20mdk (V=2,x=4,y=3).
La shell è il collegamento tra l'utente e il kernel. Mentre il kernel è definito, per una data installazione, una volta per tutte, esistono diverse possibili realizzazioni di SHELL per lo stesso computer, che possono essere presenti simultaneamente nello stesso sistema ed eventualmente essere richiamate o abbandonate dall'utente nel corso di una sessione interattiva.
Per ogni utente viene definita una cosidetta SHELL di login, ossia che è attiva all'atto del collegamento; nel nostro caso la SHELL di login è la "**tcsh**".
È importante osservare che le SHELL Linux distinguono le lettere maiuscole da quelle minuscole; in questo senso si dice che Linux è "case sensitive".
I comandi Linux sono terminati dal tasto **Enter** che serve a passare il comando stesso al sistema operativo per la sua esecuzione. Essi possono contenere delle opzioni e/o dei parametri, per i quali occorre rispettare la stessa distinzione tra lettere minuscole e maiuscole.
Per richiamare comandi eseguiti in precedenza si possono utilizzare i tasti **Up** e **Down** (↑ e ↓).
Il comando `history n` mostra su terminale gli ultimi n comandi eseguiti.

## 4.3 Struttura dei files e delle directories

Dal punto di vista logico un file è un contenitore di informazioni memorizzate in modo permanente (cioè che non vengono perse quando il computer viene spento). Dal punto di vista fisico un file è una porzione di un disco magnetico sulla quale si può scrivere, leggere e cancellare.

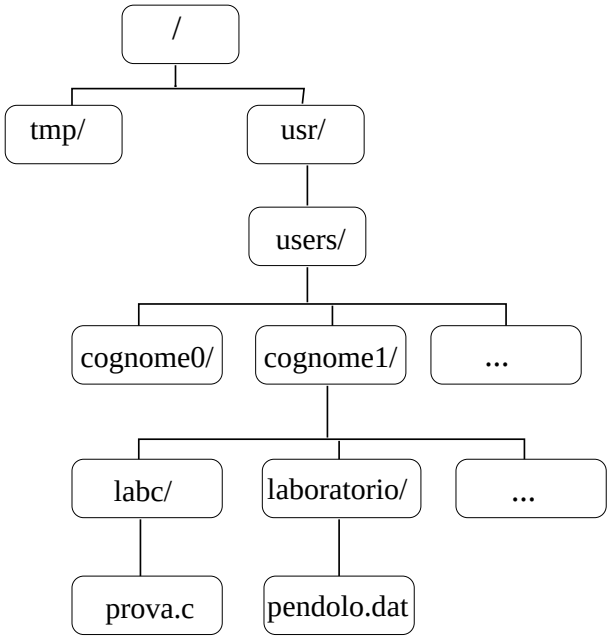


Figura 4.2: Esempio di struttura ad albero di directories

Ogni file possiede un nome, che ne consente l' identificazione. Inoltre, per consentire una migliore gestione dello spazio disponibile, i files sono, dal punto di vista logico, racchiusi in contenitori che si chiamano "directories". Ogni directory può contenere files o altre directories, di modo che l' intero sistema (detto "filesystem") risulta organizzato ad albero (Fig. 4.2). Il nome completo di un file è dato dalla serie completa dei nomi delle directories in cui è contenuto a partire dalla radice dell' albero oltre che dal suo nome proprio. La radice si chiama **/**, e lo stesso carattere **/** divide i nomi delle directories ed il nome dell' ultima directory dal nome proprio del file. Ad esempio il nome del file **prova.c** nella directory **labc** sarà

**/usr/users/cognomeI/labc/prova.c**

dove **I** è l'ultima cifra significativa dell' anno di iscrizione. Ovviamente è piuttosto scomodo utilizzare i nomi completi dei files, visto che di solito sono lunghi; di conseguenza Linux fornisce alcuni strumenti per abbreviare i nomi. Il più importante si chiama "working directory" (WD): si tratta di una particolare directory che viene usata come punto di partenza per espandere i nomi di files che non cominciano per **/**. Ad esempio, se la WD è **/usr/users/cognomeI** il nome del file precedente può essere abbreviato in

**labc/prova.c**

Per sapere quale è la WD si usa il comando **pwd**; per cambiare la WD si usa il comando **cd**, che accetta come parametro il nome di una directory (con o senza **/** iniziale a seconda che si voglia partire dalla radice o dalla WD).
Il nome della WD può essere abbreviato con il simbolo **.** (punto), mentre per la directory genitrice della WD si può usare il simbolo **..** (due punti). Ad esempio, se siamo nella directory **labc**, potremmo accedere al file **pendolo.dat** nella directory **laboratorio**,utilizzando il nome

**../laboratorio/pendolo.dat**

in quanto, se la WD è **labc**, **..** è un sinonimo di

**/usr/users/cognomeI/**

All'inizio di una sessione la WD è una directory specifica per ciascun utente (il cui nome coincide con lo username); questa directory viene detta "home directory" (HD) e il suo nome può essere abbreviato con il simbolo ~.
Ogni utente può creare nuove directories all' interno della HD utilizzando il comando **mkdir**.
La quantità di dati che ogni utente può scrivere nella propria HD e nelle directories che essa contiene è limitato. Il comando **quota** consente di verificare la situazione del proprio account in termini di spazio libero (i dati sono espressi in kbytes). Nel caso lo spazio disponibile nella HD non fosse sufficiente, è possibile utilizzare un' area temporanea per salvare files di grandi dimensioni; l' area temporanea si chiama **/tmp**. È evidente che devono essere messi in tale area solo files temporanei o facilmente ricreabili; ad esempio i files eseguibili ottenuti dalla compilazione di un sorgente C++ possono essere scritti in **/tmp**.

## 4.4 Controllo dei processi

Come già detto Linux è un sistema multi-task ed ogni SHELL può eseguire diversi processi contemporaneamente. Il comando `ps` mostra tutti i processi che sono in esecuzione nella SHELL. Questo comando è utile per trovare l' identificatore (PID) di un processo che non risponde e che si vuole fermare. L' arresto di processo si esegue con il comando `kill PID`.
Linux permette di sospendere l' esecuzione di un programma e riattivarla in seguito.
Quando un programma è "lanciato" dalla linea di comando digitando il suo nome, si dice che tale processo è eseguito in "foreground" (il terminale resta bloccato fino al termine del programma). Un programma in "foreground" che non risponde può essere fermato bruscamente con Ctrl-C, o sospeso con Ctrl-Z.
Se un stesso programma è "lanciato" aggiungendo dopo il nome il carattere &, si dice che il programma è avviato in "background" (il terminale è libero di ricevere comandi).
Il comando `jobs` mostra la lista dei programmi in "background". Ogni programma è preceduto da un numero (`n`). Per fermare un programma in "background" si esegue il comando `kill %n`. Il comando `fg %n` (`bg %n`) manda in "foreground" ("background") il programma corrispondente. Se `n` non è specificato `fg` e `bg` agiscono sul programma corrente (quello con il + nella lista stampata da `jobs`).

## 4.5 Nomi dei files e delle directories

Come abbiamo già detto il SO Linux è "Case sensitive" quindi un file o una directory con il nome **Pippo** è distinta da quello/a con il nome **pippo**.
I nomi dei files e delle directories non possono contenere nessuno dei seguenti caratteri: spazio, ?, * ,! ,apici, accenti e virgolette varie, <, >, |, il carattere separatore '/', '%', '$', ', parentesi varie. Questi sono caratteri che hanno speciali significati per la shell. Anche far cominciare il nome di un file con il carattere "-" non è una buona idea, in quanto usato per indicare le opzioni di un comando.

## 4.6 Comandi per la gestione di files e directories

La maggior parte dei comandi del SO agisce su files: di conseguenza è molto importante imparare bene a scrivere i nomi dei files ed a manipolare le directories.
La Tab. 4.1 contiene è una lista non esaustiva dei comandi per la manipolazione di files in Linux, le opzioni riportate sono solo quelli più utilizzate: *nome* indica un generico nome di file o di directory, mentre *fil* e *dir* si riferiscono in modo specifico a un file e a una directory. Le opzioni si possono combinare; se, per esempio, si vuole eseguire il comando `ls` con l' opzione `-a` e `-l` si scriverà `ls -la`.
Per una descrizione più completa si rimanda al manuale online (vedere Par. 4.12).

| comm | opt | | |
|---|---|---|---|
| `pwd` | | | Stampa su schermo il nome della WD (Working Directory) |
| `echo` | | *textline* | Mostra la linea di testo *textline* |
| `cd` | | *dir* | Cambia la WD in dir |
| `ls` | | *nome* | Lista i files contenuti nella dir specificata |
| | `-l` | | stampa infomazioni aggiuntive sui files (data, protezioni) |
| | `-a` | | lista tutti i files (compresi quelli del tipo .nome) |
| | `-t` | | ordina i files per data dell' ultima modifica |
| `mkdir` | | *dir* | Crea *dir* |
| `rmdir` | | *dir* | Cancella *dir* |
| `mv` | | *n1 n2* | Ridenomina *n1* in *n2* |
| `cp` | | *n1 n2* | Copia *n1* in *n2* |
| | `-r` | | copia il contenuto della dir *n1* nella dir *n2* |
| `rm` | | *fil* | Cancella *fil* |
| | `-i` | | chiede conferma prima di cancellare ciascun file |
| `cat` | | *fil* | Stampa su schermo il contenuto di *fil* (accetta più files) |
| `less` | | *fil* | Impagina il contenuto di *fil* su schermate |
| | | | (spazio → schermata successiva, b → schermata precedente) |
| `tail` | `-n` $N$ | *fil* | Stampa su schermo le ultime $N$ linee di *fil* |
| `head` | `-n` $N$ | *fil* | Stampa su schermo le prime $N$ linee di *fil* |
| `grep` | | *string fil* | Cerca la stringa *string* nel file *fil* |
| | `-i` | | non fa differenza tra maiuscole e minuscole (in *string*) |
| `find` | `-name` | *fil* | Cerca un file in WD e nelle sue sotto-directories |
| `diff` | | *fil1 fil2* | Stampa su schermo le linee di testo diverso tra due files |
| `ln` | `-s` | *fil link* | Crea un equivalenza simbolica tra *link* e *fil* |
| | | | (tutte le operazioni eseguite su *link* sono applicate a $fil$) |
| | `-f` | | se *link* esista già lo cancella |
| `lpr` | `-P` *printer* | *fil* | Manda in stampa sulla stampante *printer* il file *fil* |

Tabella 4.1: Breve elenco di alcuni comandi Linux utili per la gestione di files

### 4.6.1 Utilizzo di wildcard

Può essere utile, in alcuni casi, lasciare le lettere non definite nei nomi dei files (wildcards) (ad esempio se si vogliono eseguire comandi su più files aventi una stringa di caratteri comune nel

nome).
Le wildcards consentite sono:

| | |
|---|---|
| * | sostituisce qualsiasi stringa di caratteri (di qualsiasi lunghezza) |
| ? | sostituisce un singolo carattere |
| [03] | sostituisce il carattere 0 o 3 |
| [0-3] | sostituisce i caratteri corrispondenti ai numeri da 0 a 3 (per i caratteri vale l' ordine alfabetico) |

Es.:
Se in una directory sono contenuti i files: test00.f test1.f test2.f test3.f, il comando `ls test*.f` li listerà tutti, `ls test?.f` listerà solo gli ultimi tre e il comando `ls test[12].f` listerà il secondo e il terzo.

### 4.6.2 Pipe e redirezione

In Linux i comandi possono essere eseguiti in sequenza. Per passare l' output di un comando come input al seguente si utilizza il carattere | (chiamato pipe). Vediamo un esempio:

```
ls | grep ps | less
```

questo comando lista il contenuto dei files nelle directory (`ls`), cerca quelli che contengono la stringa ps nel nome (`grep`) e li passa a `less` che li visualizza su schermo.
I simboli `<` e `>` permettono di ridirigere lo standard input (tastiera) e lo standard output (terminale).
Se scriviamo:

```
ls > listato
```

il listato della directory invece di comparire sul terminale viene scritto sul file *listato* (se non esiste il file viene creato, se esiste sovrascritto).
Il comando

```
ls >> listato
```

è simile al precedente salvo che, ora, il risultato di `ls` è scritto in coda al file *listato* (se non esiste il file viene creato).

### 4.6.3 Protezioni

Le protezioni definiscono come i diversi utenti possono accedere a files e directories.
Le protezione per un file/directory possono essere controllate con il comando `ls -l` che fornisce (per una directory) un risultato di questo tipo:

```
drwxr-xr-x   2 smith                1024 Oct  7 20:01 Work
```

La legenda della prima serie di caratteri (che contiene le protezioni) è data dalla seguente tabella

```
Type  User Group  Other
 *    ***    ***    ***
 d    rwx    rwx    rwx
 l    ---    ---    ---
 -
```

Il primo carattere definisce il tipo di file (d=directory,l=link simbolico,−= file), gli altri regolano i privilegi di accesso ai files (r=lettura,w=scrittura,x=esecuzione) da parte di diversi gruppi di persone.
Lo User è l' utente che si è collegato, Group il suo gruppo di appartenenza, Other tutte le altre persone che possono avere accesso a quella macchina.
Per modificare le protezioni si utilizza il comando `chmod`.
Es.:
`chmod g+x` *fil*: aggiunge a Group il diritto di eseguire *fil*
`chmod o-x` *fil*: toglie a Group il diritto di leggere *fil*

## 4.7 Il text editor

Il text editor è uno strumento che serve a creare files; ogni tipo di file "leggibile" (un testo, il sorgente di un programma, un file di dati) può essere creato con un text editor. Esistono molti tipi di editor, e la scelta dipende essenzialmente dai gusti personali. Quello che vi consigliamo si chiama **emacs**. Per attivarlo è sifficiente dare il comando

**emacs** nome-fil &

o semplicemente

**emacs** &

Se la variabile DISPLAY (Par. 4.13) è correttamente definita, verrà creata sul vostro X-terminal una nuova finestra per l' editor; **emacs** possiede un numero enorme di comandi che consentono funzioni di editing molto sofisticate, ma la descrizione di queste possibilità esula dagli scopi di queste note. Il menu "Help" contiene tutte le informazioni che servono per usare al meglio **emacs**. Basterà qui osservare che **emacs** è un editor full-screen, e che quindi è possibile muoversi all' interno del file usando i tasti con le frecce; inoltre la maggior parte dei comandi possono essere attivati usando il mouse e la "menu-bar" che appare nella parte alta della finestra. È possibile aprire più files contemporaneamente, utilizzando l' opzione "Open File" nel menu "Files" e il menu "Buffers" per scegliere tra i diversi files. Per salvare un file modificato si usa il comando "Save Buffer" nel menu "Files".
Siccome disponete, per le esercitazioni, di terminali che consentono di lavorare su più finestre, è conveniente evitare di aprire e chiudre l' editor in continuazione. All' inizio della seduta potrete aprire **emacs** utilizzando la "&" a fine riga in modo da creare una finestra indipendente per l' editor e lasciare il vostro terminale libero di eseguire i comandi di compilazione ed esecuzione. Dopo avere eseguito una modifica al vostro file, salvatelo senza uscire dall' editor con "Save Buffer" nel menu "Files", compilate ed eseguite utilizzando il terminale e tornate a modificare il file nella finestra dell' editor, che dovrà essere chiusa solo alla fine del lavoro.

## 4.8 Compilazione ed esecuzione di un programma

In questo paragrafo daremo una breve descrizione dei comandi necessari per compilare un programma allo scopo di realizzare il primo semplice programma in C++. Un' introduzione più completa sarà fornita in Par. 5.11.2.
Il comando di compilazione permette di tradurre un programma scritto in linguaggio di "alto" livello (in linguaggio C++ nel nostro caso) in un codice eseguibile dalla macchina. La compilazione prevede in genere due fasi: la compilazione vera e propria che trasforma il file sorgente

(.cpp) in file oggetto (.o) e la fase di "link" in cui si produce il file eseguibile includendo, se necessario, files di libreria o altri files oggetto.
Il comando:

```
g++ -o file file.cpp
```

compila il sorgente `file.cpp` e crea l' eseguibile `file`.
Questo comando crea in un solo passo l' eseguibile ed è equivalente a

```
g++ -c -o file.o file.cpp     (compilazione)
g++    -o file   file.o       (link)
```

Il file eseguibile può essere eseguito semplicemente digitando il suo nome completo:

```
./file
```

## 4.9 Il debugger simbolico

Il debugger simbolico è uno strumento software che consente di eseguire un programma una istruzione alla volta e che permette di esaminare il contenuto delle variabili (sia locali che globali) in una qualunque fase dell' esecuzione. È quindi uno strumento utilissimo per capire le cause di malfunzionamento di un programma. Esistono sul mercato vari tipi di debugger: quello che vi consigliamo si chiama **gdb**.

Esistono due modi per utilizzare un debugger; il primo modo è "post–mortem", nel senso che utilizza il file chiamato `core` che ogni programma genera in caso di errore. In questo caso i comandi da usare sono:

```
gdb <nome programma> core
where
```

Il comando `where` del debugger vi informarà di quale linea del programma ha causato l' errore. Per uscire dal debugger utilizzate il comando `quit`.

Il secondo utilizzo del debugger è quello di eseguire un programma passo–passo. In questo caso si comincia con

```
gdb <nome programma>
```

Si deve quindi predisporre almeno un "breakpoint", ovvero una posizione del programma alla quale l' esecuzione deve sospendersi; un breakpoint può essere specificato in due modi:

```
break <routine>   ! Sospende l' esecuzione all' ingresso nella routine
```

oppure

```
break <numero linea>  ! Sospende l' esecuzione alla linea specificata
```

Il numero di linea da specificare è quello del programma sorgente.

Dichiarati i breakpoints si può cominciare l' esecuzione con il comando `run`. Quando l' esecuzione si interrompe si può:

- riprendere fino al successivo break con il comando `cont`;
- eseguire solo la linea successiva senza entrare in eventuali routines chiamate con il comando `next`;
- eseguire solo la linea successiva entrando in eventuali routines chiamate con il comando `step`;
- proseguire fino all' uscita dalla routine corrente con il comando `return`;
- leggere il file sorgente nel punto corrispondente alla linea successiva a quella appena eseguita con il comando `list`;
- esaminare il contenuto di una variabile visibile nella routine corrente con il comando `print <nome_variabile>`;
- modificare il contenuto di una variabile con il comando
  `set <nome_variabile> = <valore>`;
- creare nuovi breakpoints.

Le componenti dei vettori possono essere esaminate individualmente, usando le parentesi quadre (per i programmi scritti in C++).
L' utilizzo di questi comandi, e dei molti altri disponibili per i quali potete consultare il manuale (vedi Par. 4.12), consente di trovare le cause di malfunzionamenti e può aiutare a comprendere meglio il modo di operare di alcune istruzioni dei linguaggi di programmazione.

Il debugger **gdb** possiede una versione grafica cui potete accedere con il comando **ddd**. In questa versione il file sorgente è sempre visibile ed è possibile definire un breakpoint semplicemente "cliccando" a fianco di ogni linea.

## 4.10 Procedure di comandi

Accade sovente di dover usare ripetutamente sequenze di comandi complesse. Per consentire un risparmio di tempo ed un incremento di flessibilità gli interpretatori dei comandi (shell) sono dotati di un certo grado di programmabilità: è cioè possibile scrivere delle sequenze di comandi in un file (che viene detto "script") ed eseguirle scrivendo un solo comando. Inoltre all' interno di uno script si possono utilizzare variabili e strutture complesse tipo if-then-else, do-while e così via. Un qualsiasi manuale di Linux riporta in dettaglio le possibilità di programmazione incluse nei diversi tipi di shell (quella che noi utilizziamo si chiama `csh`,"C shell"[1]). Nel seguito analizzeremo solo alcuni elementi sintattici di base che possono servire per scrivere scripts molto semplici.

### 4.10.1 Uso delle variabili

Nella C shell possono essere definiti variabili e vettori di tipo stringa. Esistono variabili locali e variabili globali: le prime esistono solo all' interno della procedura che le crea, le seconde restano attive nell' arco di tutta la sessione. Le variabili locali vengono dichiarate con il comando **set**, mentre quelle globali con il comando **setenv**; ad esempio:

[1] la shell di login `tcsh` è semplicemente una versione "potenziata" della `csh`

| | |
|---|---|
| `set a1 = val_1` | Crea la variabile locale a1 e le assegna il valore val_1 |
| `setenv g1 val_2` | Crea la variabile globale g1 e le assegna il valore val_2 |
| `set vect = (v1 v2 v3)` | Crea il vettore a tre componenti vect ed assegna i valori v1, v2 e v3 alle tre componenti |

Il contenuto della variabile `a1` è dato dal simbolo `$a1`, il valore dell' n-esima componente del vettore `vect` dal simbolo `$vect[n]`, mentre il numero di componenti del vettore `vect` è dato da `$#vect`.
È possibile assegnare ad una variabile l' output di un comando, includendo il comando medesimo tra apici rovesciati (‘); ad esempio

```
set files = (‘ls‘)
```

crea un vettore di nome `files` che contiene i nomi dei file presenti nella WD, dati come output dal comando **ls**.
Alcune variabili globali sono automaticamente definite dal sistema nella fase di login. Per vedere quali sono basta eseguire il comando `setenv` senza argomenti.
Una importante variabile globale è *PATH* che contiene la lista delle directories in cui il sistema cerca un file eseguibile se questo non è specificato con il nome completo.

### 4.10.2 Realizzazione ed esecuzione di una procedura di comandi

Una procedura di comandi può essere scritta con un text editor. La prima riga deve contenere la specificazione della shell che eseguirà i comandi. Nel caso della C shell la prima riga sarà

```
#!/usr/bin/csh
```

Le linee successive conterranno la sequenza di comandi che si desidera eseguire. Le righe che cominciano com `#` sono considerate commenti e non vengono eseguite. Una volta preparato il file è necessario renderlo eseguibile con il comando

**chmod** `u+x nome-fil`

Per eseguirlo basta quindi scriverne il nome completo, o solo il nome proprio nel caso lo script sia in una directory contenuta nel `PATH`. Il nome dello script può essere seguito da uno o più parametri. Questi verranno passati allo script come un vettore locale di nome `argv`. Di seguito trovate il listato di uno script che esegue compilazione e link del sorgente specificato come primo argomento ed esegue il programma così ottenuto. Se il secondo parametro è `-c` viene eseguita la sola compilazione, se è `-r` il programma viene solo eseguito.

```
#!/usr/bin/csh
if ($#argv == 1) then
  g++ -o /tmp/$argv[1] argv[1] fun1.cpp ‘crootlib‘
  chmod u+x /tmp/$argv[1]
  /tmp/$argv[1]
else
  switch ($argv[2])
    case -c:
    g++ -o /tmp/$argv[1] argv[1] fun1.cpp ‘crootlib‘
    chmod u+x /tmp/$argv[1]
    breaksw
```

```
    case -e:
    /tmp/$argv[1]
    breaksw

    default:
    echo "Unknown option $argv[2]"
endif
```

Sulla base di questo esempio vi sarà possibile creare procedure di comandi che semplifichino le operazioni che eseguite frequentemente.

## 4.11 Archiviazione di files

È in generale buona norma archiviare e salvare il lavoro fatto su un apposito supporto per evitare che vada perso per erronee operazioni dell' utente o problemi di sistema. Nel corso non vi dovrete preoccupare di questo problema perché l' archiviazione di files verrà eseguita automaticamente; tuttavia nel seguito descriveremo le operazioni di base necessarie.

### 4.11.1 Compressione di files

Per limitare l' occupazione dello spazio si disco può essere utile comprimere, con un processo riproducibile, i files in modo che occupino meno spazio. Per esemplificare il metodo di compressione consideriamo un file di testo: invece di scrivere tutti i caratteri si può scrivere il singolo carattere e quante volte questo è ripetuto (basti pensare a quanti spazi bianchi consecutivi sono contenuti in un file di testo per capire quanto si può guadagnare). Ovviamente gli attuali algoritmi di compressione utilizzano metodi più sofisticati di quello descritto.
Il comando per comprimere[2] è
`gzip` *fil*
che crea il file compresso *fil*.gz, che può essere decompresso con il comando:
`gunzip` *fil*.gz
Per archiviare un' intera struttura di directory in unico file si usa il comando:
`tar cvf` *tarfile.tar dir1 dir2 dir3*
che crea (`c`=create) il file archivio *tarfile*.
Per riottenere la struttura di directories dal file *tarfile.tar*:
`tar xvf` *tarfile.tar dir1 dir2 dir3*
(`x`=extract).
Il file di archivio può essere automaticamente compresso se si aggiunge l' opzione `z`.

### 4.11.2 Accesso a dischi rimuovibili

Uno dei supporti più semplici e più economici su cui salvare files sono i floppy disk.
Prima di utilizzare un floppy disk occore creare su di esso un "file-sistem" (formattazione) con il comando:
`mkfs -t` *os device blocks*
dove *os* è il tipo di "file-system" (*ext2* è il default in Linux, *msdos* è invece utile se volete rileggere il dischetto anche da Windows), *device* è l' identificatore del device dell' unità floppy (in genere

[2] i comandi riportati sono modifiche elaborate dalla fondazione GNU ai comandi standard `zip` e `unzip`

e nel nostro caso /dev/fd0) e *blocks* è il numero di blocchi (in unità di 1Kb) da allocare sul disco (per un floppy *blocks* = 1440). Questa operazione non è ovviamente necessaria se il dischetto è già formattato.
Il dischetto è ora pronto per essere utilizzato in scrittura ed in lettura ma non fa ancora parte del sistema su cui state lavorando; per includerlo nel sistema occorre montarlo con il comando:

```
> mount /mnt/floppy
```

A questo punto potete fare sul dischetto tutte le operazioni che usualmente fate sul disco rigido (quello che contiene la vostra HD).
Finite le operazioni dovete smontare il dischetto con il comando `umount` (analogo a `mount`).

## 4.12 Come ottenere ulteriori informazioni

Ulteriori informazioni sono reperibili, oltre che sui manuali cartacei, direttamente sul computer. La sintassi di ogni comando e funzione di libreria di Linux è documentata per mezzo di un help "on–line" cui potete accedere tramite il comando **man** (o **xman** se state utilizzando un X-terminal; la sintassi è

```
man <comando>
```

o semplicemente

```
xman &
```

Se non si sa il nome del comando che interessa, ma si conosce qualche parola chiave relativa all'argomento si può dare il comando

```
man -k parola_chiave
```

## 4.13 Il sistema grafico X11

Per utilizzare un elaboratore è necessario avere un terminale. Il terminale in senso stretto è oggetto "stupido", nel senso che si limita a mostrare sullo schermo i caratteri che riceve dal computer cui è collegato e a trasmettere al computer i caratteri digitati sulla tastiera.
Più spesso, però, quello che si indica con terminale è un piccolo computer capace di elaborazioni grafiche piuttosto complesse: in generale possiede un "sistema a finestre" dove ogni finestra può emulare un terminale tradizionale che può essere connesso ad uno o piu' elaboratori. Inoltre può ricevere comandi grafici da altri computers ed eseguirli mostrando il risultato su una finestra.
L' architettura "client-server" gestisce, nei sistemi LINUX, il funzionamento dei terminali.
Il "server" è un computer che invia dei comandi grafici, il "client" è un "oggetto" che riceve i comandi grafici e li esegue mostrando il risultato su terminale grafico. La comunicazione tra client e server avviene tramite una rete di trasmissione dati, nella quale ad ogni macchina è attribuito un "indirizzo" che ne consente l' identificazione (vedere Fig. 4.3).
Il protocollo di comunicazione si chiama X11.
Per le esercitazioni utilizzerete dei PC con il sistema operativo LINUX.
Il loro indirizzo è **stationN.aula1.fisica** con $N = 1..18$. In questo caso non si tratta di terminali ma di veri e propri computers che possono essere usati, oltre che per collegarsi ad un sistema remoto, anche per eseguire programmi localmente. In questo caso ciascuna macchina è un sistema "client-server".

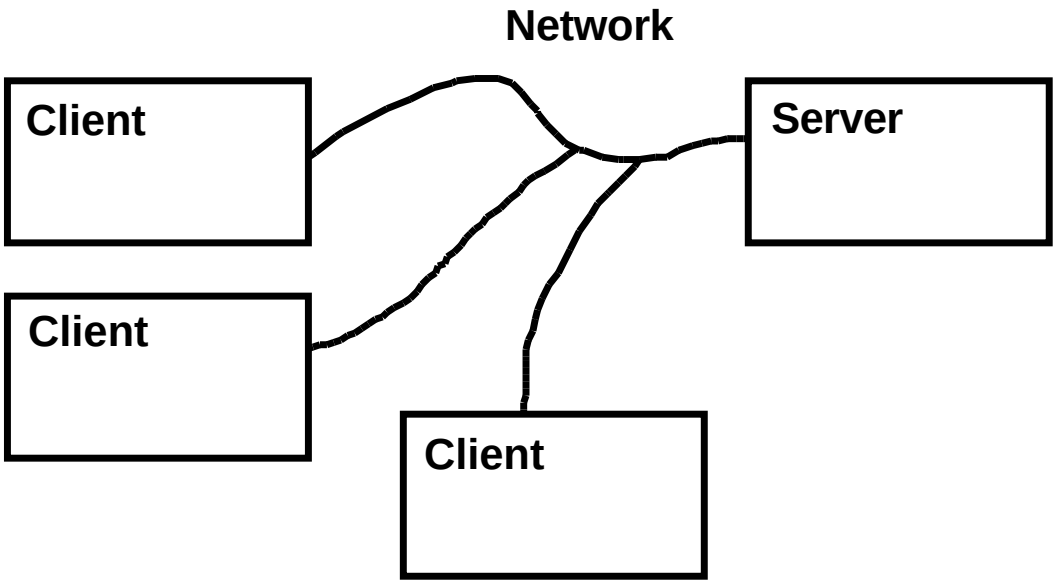


Figura 4.3: Esempio di sistema client-server

Per utilizzare questi PC come semplici "client" nel collegamento remoto ad un altro PC (ad es. station05.aula2.fisica) occorre seguire i seguenti passi:

- su una delle finestre inviare il comando

  ```
  ssh station05.aula2.fisica
  ```

- Seguire la normale procedura di login (vedere Par 4.2.2).

- Informare **station05** dell' indirizzo dell' X-terminal che state usando, in modo che sia in grado di aprire le varie finestre grafiche su quel terminale. In generale questa operazione viene eseguita automaticamente durante il login; in alcuni casi però è necessario eseguirla manualmente. Se avete dubbi potete controllare quale X-terminal **station05** pensa voi stiate usando con il comando **echo $DISPLAY**, e fornire l' indirizzo con il comando **setenv DISPLAY xxxx.yyyy.fisica:0** (Per maggiori dettagli vedere Par. 4.10)

Tutte le esercitazioni del corso utilizzeranno come i PC macchine locali, in questo caso la variabile DISPLAY viene automaticamente attribuita.
Altre macchine a vostra disposizione sono i PC dell'aula di Laboratorio 1 stationN.esp1.fisica $N = 01..10$ e i PC dell' aula terminali al quarto piano stationN.aula2.fisica $N = 1..8$. Tutti questi PC hanno il sistema operativo Linux e condividono la stessa directory di login (vedi Par. 4.3).

# Capitolo 5

# Sintassi del linguaggio C++

## 5.1 Introduzione

L' operazione di apprendimento di un particolare linguaggio di programmazione deve in qualche misura essere smitizzata. La maggior parte dei linguaggi evoluti, al di là degli aspetti estetici o semantici, sono tra loro molto simili nella struttura logica: gli strumenti che forniscono al programmatore sono spesso del tutto analoghi. La realizzazione di un programma può essere divisa in due parti: la prima consiste nel passaggio da una descrizione formale di un problema ad una descrizione di tipo algoritmico della sua soluzione; tale descrizione algoritmica, dovendo essere eseguita da una macchina "stupida", deve prevedere tutte le possibili variazioni del problema e tutti i possibili errori che si possono incontrare nel giungere alla soluzione, e trattarli adeguatamente. La descrizione algoritmica utilizzerà dei "building blocks" fondamentali (ad esempio il ciclo DO-WHILE) che sono propri del linguaggio di programmazione prescelto. Ora, proprio la similitudine nella struttura logica dei linguaggi permette di affrontare questa prima fase della programmazione in modo del tutto indipendente da quella che sarà la scelta finale del linguaggio. La seconda fase della programmazione, ovvero la realizzazione del codice sorgente scritto in un determinato linguaggio, sarà in qualche modo una pura opera di traduzione dal linguaggio astratto dei "building blocks" al linguaggio reale che si è scelto. Servirà quindi una sorta di vocabolario che, data una operazione fondamentale, ne illustri l' utilizzo e spieghi come realizzarla in un dato linguaggio: questo è lo scopo di queste pagine.

Ciò premesso sembrerebbe legittimo affermare che, noto il linguaggio astratto e dato un vocabolario, si possa programmare in qualunque linguaggio. In realtà non tutti i linguaggi di programmazione sono tra loro strutturalmente uguali: alcuni (ad esempio il BASIC) hanno funzionalità inferiori rispetto al C++, altri (ad esempio il C e tutti i linguaggi non "Object Oriented") sono meno flessibili. Vi è tuttavia una famiglia piuttosto ampia di linguaggi che sono del tutto similari alla parte non "Object Oriented" del C++ (tra questi i più diffusi sono probabilmente il C stesso, il FORTRAN e il PASCAL), e che potrete legittimamente affermare di "conoscere" non appena abbiate compreso a fondo la struttura logica comune.

Da queste poche considerazioni si comprende comunque come tutte le disquisizioni sulla superiorità di questo o quel linguaggio siano, nella sostanza, del tutto inutili. La vera scelta sta nel modo di realizzare l' algoritmo che risolve un dato problema, e questa è la vera essenza (e la vera difficoltà) della programmazione. La scelta finale del linguaggio dipenderà da considerazioni di tipo estetico, affettivo, economico o pratico; comunque sarà piuttosto irrilevante. Saper programmare significa essere capaci a risolvere problemi in modo algoritmico (ovvero in modo razionale, controllato e riproducibile, esaminando tutte le possibilità e prevedendo tutte le ec-

cezioni). È un esercizio di logica più che di informatica, e può essere molto utile anche quando non ci si trova davanti ad una tastiera.

Prima di proseguire con l' illustrazione del linguaggio è bene osservare che le note che seguono non pretendono di introdurre tutti gli aspetti e le potenzialità del C++, ma solo un sottoinsieme sufficiente ad affrontare le problematiche tipiche delle esercitazioni di laboratorio o della maggior parte delle tesi di laurea. Molti strumenti avanzati non verranno neppure citati, ed in alcuni casi si riporteranno solo sintassi semplificate di alcune istruzioni.

## 5.2 Formato di un programma C++

Il sorgente di un programma è un file che contiene un insieme di istruzioni in un dato linguaggio (C++ in questo caso). Le estensioni valide per un sorgente C++ sono `.cpp`, `.cxx` e `.cc`.
Per essere eseguito il file sorgente dovrà passare attraverso le operazioni di compilazione e link descritte che saranno descritte nel paragrafo 5.11.
Le regole con cui il file sorgente deve essere scritto dipendono dal linguaggio.
Per il C++ le regole sono piuttosto blande: i comandi possono essere scritti in qualunque posizione, uno o più per riga; si deve porre un punto e virgola (;) alla fine di ciascun comando (un comando terminato da ; viene anche detto istruzione). I comandi sono usualmente composti da più elementi, ed uno o più di questi elementi possono essere a loro volta comandi completi; è il caso del comando `if`, che è fatto così :

`if (`*espressione*`)` *comando*`;`

dove *comando* è un qualunque comando C++. Gruppi di comandi possono essere realizzati utilizzando coppie di parentesi graffe ({}); sintatticamente un gruppo di comandi equivale ad un comando singolo; nel caso dell' *if* scriveremo:

`if (`*espressione*`) {`
*comando_1*;
*comando_2*;
...
}

Da notare che se un blocco è posto alla fine di un comando si può omettere il punto e virgola finale.
Non esistono regole di ordinamento delle istruzioni, a parte il fatto che ogni simbolo deve essere definito prima di essere utilizzato. Le lettere maiuscole e quelle minuscole sono considerate diverse: questo è rilevante sia per le istruzioni, sia per i nomi delle variabili. È possibile commentare un linea o un parte di una linea con un doppio slash (//), tutto ciò che segue (in linea) tale simbolo non viene considerato nella compilazione.

```
...  // Questo e' un commento
```

È possibile inserire commenti che si estendono su più linee racchiudendoli tra coppie `/* */`. Ad esempio:

```
/* Questo e' un commento che si
   estende su due linee      */
```

## 5.3 Programma principale

Ogni programma è formato da un insieme di funzioni; le funzioni possono a loro volta essere raggruppate in moduli. Un modulo contiene usualmente funzioni tra loro affini (per esempio che lavorano sugli stessi dati o che svolgono funzioni tra loro correlate).
In C++ le funzioni contenute in uno stesso file sorgente (ovvero in uno stesso file .cpp) formano un unico modulo.
Ogni programma completo deve contenere una ed una sola funzione, detta "main program", dalla quale partirà l' esecuzione. In C++ il main program è una funzione che si chiama `main`.

```
main() {
  ...
}
```

La struttura dei programmi e l'utilizzo delle funzioni saranno descritti nel paragrafo 5.10.

## 5.4 Variabili

Una variabile identifica una locazione di memoria: il suo nome è l' identificatore di tale locazione. Una variabile è caratterizzata dal tipo, dalla dimensione, dal modo di allocazione e dalla visibilità o scope. Il tipo specifica la codifica con cui i dati vengono scritti in memoria; la dimensione indica di quanti elementi singoli di un dato tipo si compone una variabile: si possono fabbricare scalari, vettori o matrici a due o più dimensioni; il modo di allocazione indica come il compilatore riserva la memoria relativa alla variabile; lo scope indica quali parti di un programma possono accedere in lettura e/o scrittura alla variabile.

### 5.4.1 Tipi

Il C++ ha un insieme di tipi fondamentali che si adattano alle più comuni unità di memoria e ai loro usi più frequenti nella gestione dei dati:

- Booleano (variabile logica `bool`)
- Carattere (`char`)
- Tipi Interi (as es. `int`)
- Tipi Floating-point (as es. `double`)
- Void (utilizzato per indicare assenza di informazione)

Per la maggior parte delle applicazioni possono bastare: `bool` per le variabili logiche, `char` per i caratteri, `int` per gli interi e `double` per i numeri floating-point. I restanti tipi sono variazioni utili per usi specifici e possono essere considerate solo al momento in cui se ne presenti il bisogno.

#### Tipo Booleano

Una variabile di tipo booleano può avere solo due valori *true* (vero) o *false* (falso).
Una variabile booleana può essere utilizzata per esprimere il risultato di un operazione logica:

```
bool test = a == b; // = assegnazione, == uguaglianza
```

Se `a` e `b` hanno lo stesso valore `test` diventa *true* altrimenti `test` diventa *false*.
Per definizione *true* ha valore 1 mentre *false* ha valore 0. Allo stesso modo gli interi possono essere implicitamente convertiti a booleani: interi diversi da zero si convertono in *true* e 0 si converte in *false*.

### Tipi Carattere

Una variabile di tipo carattere contiene un carattere alfanumerico e può essere utilizzata per rappresentare testi. In generale, piuttosto che una singola variabile di tipo carattere si utilizzano insiemi di caratteri, che vengono detti "stringhe". Per maggiori informazioni sull'implementazione di caratteri e stringhe in C++ si veda il Par. 5.14.

### Tipi interi

I tipi interi sono tre, con diversa capacità, `short int`, `int` e `long int`. Per tutte queste variabili esiste, come per `char` le varianti `signed` o `unsigned`. I tipi interi senza segno `unsigned` sono in genere utilizzati per rappresentare variabili intere come sequenze di bit.
Le costanti intere (literal int) possono essere rappresentate in quattro modi: decimale, octale, esadecimale o costanti carattere. Una costante che inizia per *0x* è un esadecimale, per *0* è un ottale.

| | | | |
|---|---|---|---|
| decimal: | 0 | 2 | 63 |
| ottale: | 00 | 02 | 077 |
| esadecimale: | 0x0 | 0x2 | 0x3f |

### Tipi floating-point

I tipi floating-point rappresentano numeri reali. Come gli interi sono disponibili in tre versioni, corrispondenti a diverse precisioni: `float`, `double` e `long double`.
Una costante floating point (literal floating) si può indicare con

1.0 1. .23 1.e10 3e-20

La scrittura $xey$ significa $x \cdot 10^y$.

### Spazio occupato in memoria dai diversi tipi

L'unità di spazio in memoria è il byte (8 bits). Il numero di bytes occupati da una variabile di un certo tipo non è definito in generale ma dipende dall' implementazione. Il simbolo `sizeof(tipo)` (dove `tipo` è un qualsiasi tipo definito) ritorna il numero di byte occupati dal tipo `tipo`.
Di seguito è riportato lo spazio occupato dai diversi tipi e i loro intervalli di validità nell'architettura che useremo per le esercitazioni di laboratorio

| Tipo | Bytes | Intervallo (con segno) | Intervallo (senza segno) | Precisione |
|---|---|---|---|---|
| bool | 1 | - | - | - |
| char | 1 | [-127,127] | [0,255] | - |
| short int | 2 | [-32768,32767] | [0,65535] | - |
| int | 4 | [-2147483648,2147483647] | [0, 4294967295] | - |
| long int | 4 | [-2147483648,2147483647] | [0, 4294967295] | - |
| float | 4 | ∼[-1.e+38,1.e+38] | - | 6-7 digits |
| double | 8 | ∼[-1.e+308,1.e+308] | - | 14-15 digits |
| long double | 12 | | | |

### 5.4.2 Definizione

#### Scalari

Ogni variabile deve essere definita, prima di essere utilizzata, con l' istruzione

`tipo` *nome*`;`

La definizione di una variabile non implica alcuna inizializzazione (il valore iniziale non è determinato); questa, se necessario, deve essere eseguita esplicitamente.
Definizione di variabili:

```
int    i,j;
char   car;
double a,b;
bool   test;
```

definizione ed inizializzazione

```
int    i=10,j=9;
char   car='a';
double a=3.0,b=2.0;
bool   test=false;
```

#### Array

Gli array sono serie di elementi (variabili) dello stesso tipo disposti in memoria consecutivamente in maniera da poter essere accessibili aggiungendo un indice ad un unico nome.
Come ogni altra variabile l' array deve essere definito prima di essere utilizzato:

`tipo` *nome*`[`*n*`];`

*n* è una costante intera che indica il numero di elementi. L'indice dell' array, che permette di identificare i singoli elementi, va da *0* a *n-1*. Il primo elemento dell' array è quindi identificato con *nome[0]*.
L'inizializzazione di un array si esegue indicando, dopo l'operatore =, i singoli elementi tra parentesi graffe separate da virgole.

`tipo` *nome*`[`*n*`]={` *elemento_0*`,....,`*elemento_n-1*`};`

Ad esempio (array di interi):

```
int vect[5]={3,4,6,7,8}; // inizializzazione
vect[3]=4;               // modifica del valore del quarto elemento
```

Per eseguire operazioni su array è necessario ricondurle ad operazioni sui singoli elementi.
Ad esempio (acquisizione dei valori di un array da terminale);

```
int i;
double x[10];
...
for(i=0;i<10;i++){
  cin >> x[i];
}
```

**Matrici**

Gli array possono avere due componenti (matrici) o più e vengono definiti come:
`tipo` *nome*`[`*m*`][`*n*`];`
Ad esempio una matrice di double può essere definita come:

```
double mat[2][3];
```

Una matrice mat[n][m] è rappresentata in memoria con un array ad una sola componente composto da, nell'ordine,

```
x[0][0] x[0][1]...x[0][m] x[1][0]...x[1][m]...x[n][0]...x[n][m]
```

il primo indice è l'indice di riga, il secondo quello di colonna.
L'inizializzazione dei valore deve quindi tenere conto di questa convenzione:

```
double mat[2][3]={0.0, 2.0, 3.0, 2.0, 3.0, 1.0};
```

la prima riga della matrice è composta da 0.0 2.0 3.0 la seconda riga da 2.0 3.0 1.0 .
Tipicamente l'accesso agli elementi di matrice si realizza con due cicli annidati:

```
for(int i=0; i<2; ++i) {// i identifica la riga
  for(int j=0; j<3; ++j){// per ogni riga, j identifica
                         // l'elemento della riga (la colonna)
    cout << mat[i][j] << endl;
    // mat[i][j]: l'elemento sulla riga i e sulla colonna j
  }
}
```

### 5.4.3 Visibilità

Per regione di visibilità di una variabile si intende la regione del codice in cui la variabile è vista e può quindi essere utilizzata.
Una variabile definita al di fuori del corpo di ogni funzione viene detta variabile **globale** ed è visibile a tutte le funzioni nel file sorgente. Una variabile definita nel corpo di una funzione viena detta variabile **locale**. La regione di visibilità di una variabile coincide con il più piccolo blocco in cui questa è definita (un blocco è un insieme di istruzioni racchiuse fra { }: può identificare una funzione o il blocco di un' istruzione di controllo).

```
for (int i=0; i<n; i++){
  if (i>2){            // i e' nella sua regione di visibilita'
    double x = i;
  }
  x+=2;                // errore: la regione di visibilita' di x
                       // si limita al blocco dell' if
}
```

### 5.4.4 Allocazione

Per ogni variabile deve essere definita, all' atto della definizione, il modo in cui viene creato lo spazio memoria a cui farà riferimento:

- *automatico*: è il modo di allocazione implicito per tutte le variabili: la variabile viene creata (viene riservato lo spazio di memoria necessario) all' atto della definizione, sarà distrutta (lo spazio di memoria viene liberato) all' uscita della sua regione di visibilità.

- *statico*: la variabile viene creata all' atto della definizione ma non viene distrutta all' uscita della sua regione di visibilità. Resta disponibile (e non viene ricreata ogni volta) per tutto il programma.
  Per definire *statica* una variabile bisogna preporre la chiave `static` alla sua definizione.

Esempio: il codice

```
for (int i=0; i<3 ;i++){
  int k=0;
  cout << k << endl;
  k++;
}
```

dà come output

```
0
0
0
```

mentre il codice

```
for (int i=0; i<3 ;i++){
  static int k=0;
  cout << k << endl;
  k++;
}
```

dà come output

```
0
1
2
```

## 5.5 Costanti

È possibile definire delle costanti (identificatori corrispondenti a valori numerici costanti) tramite l'istruzione `const`

`const tipo` *nome*;

Esempi:

```
const int i=10;
const double a=3.14;
const int vect[3]={3,4,5};
```

Poichè una costante non può essere modificata deve essere inizializzata quando viene definita. L'utilizzo di costanti può essere utile per rendere più chiaro il codice e per evitare errori quando è necessario utilizzare più volte la stessa costante.

## 5.6 Operatori

Gli operatori sono simboli che eseguono un operazione sulla o sulle variabili cui vengono applicati. Gli operatori che agiscono su una sola variabile si chiamano unari (u). Gli operatori che agiscono su due variabili, come ad esempio l' operatore di somma, sono detti binari (b).
Gli operatori si possono poi dividere in relazionali, logici, numerici e logici "bit a bit" o "bitwise".

### 5.6.1 Operatori numerici

| | |
|---|---|
| = | assegnazione |
| + | addizione |
| - | sottrazione |
| | moltiplicazione |
| / | divisione |
| % | Resto intero (resto della divisione per un intero) |

Esempio:

```
x = 10\%3; // x vale 1.
```

### 5.6.2 Operatori "bit a bit"

Gli operatori "bit a bit" &, —, ^, ~, >> e << si applicano a tipi interi. L' utilizzo tipico degli operatori "bit a bit" consiste nell'eseguire operazioni tra numeri interi (senza segno) che rappresentano una sequenza di bits.

| | |
|---|---|
| ~bitseq | complemento |
| bitseq1 & bitseq2 | AND |
| bitseq1 \| bitseq2 | OR |
| bitseq1 ^ bitseq2 | OR esclusivo |
| bitseq << n | shift di n bit a sinistra della sequenza di bit |
| bitseq >> n | shift di n bit a destra della sequenza di bit |

dove bitseq, bitseq1, bitseq2 e n sono generiche variabili intere.
Per illustrare meglio il funzionamento di tali operatori negli esempi che seguono l'inizializzazione degli interi sarà eseguita utilizzando la notazione esadecimale

```
unsigned int bitseq1=0x00001011,
             bitseq2=0x00001000,
             bitseq3;
             //bitseq1 vale 3, bitseq2 vale 8
bitseq3 = bitseq1 << 1;      // bitseq3 vale 0x00002022;
bitseq3 = bitseq1 >> 1;      // bitseq3 vale 0x00000808;
bitseq3 = ~bitseq1;          // bitseq3 vale 0xffffefee;
bitseq3 = bitseq1 & bitseq2  // bitseq3 vale 0x00001000;
bitseq3 = bitseq1 | bitseq2  // bitseq3 vale 0x00001011;
bitseq3 = bitseq1 ^ bitseq2  // bitseq3 vale 0x00000011;
```

### 5.6.3 Composizione di operatori

È possibile combinare l'operazione di assegnazione con qualsiasi operatore numerico o "bit a bit". Ad esempio:

```
x += y;  // equivale a x = x + y;
i <<=1;  // equivale a i = i<<1;
y /= 2;  // equivale a y = y/2;
```

### 5.6.4 Operatori di incremento e decremento

++ è equivalente a $+= 1$
−− è equivalente a $-= 1$

Se l'operatore è prefisso (postfisso) ad una variabile, ++x (x++), il valore è modificato prima (dopo) che l'espressione a cui la variabile appartiene sia valutata.
Esempio:

```
b = 2;                          b = 2;
a = ++b;                        a = b++;
// a vale 3, b vale 3           // a vale 3, b vale 3
```

### 5.6.5 Operatori relazionali e logici

#### Operatori relazionali

`==` uguale
`!=` diverso
`>` maggiore
`<` minore
`>=` maggiore o uguale
`<=` minore o uguale

#### Operatori logici

`!` NOT logico
`&&` AND logico
`||` OR logico

Gli operatori relazionali e logici sono in genere utilizzati per esprimere condizioni.

```
if (x >= 5 || y < 9) ...
```

Traduce la condizione *se x è maggiore od uguale a 5 e y è minore di 9.*
È bene osservare che la seguente espressione non funziona come ci potremmo aspettare

```
if ( 0 <= x <= 10) ...
```

l' istruzione è legale ma viene interpreta come `(0<=x)<=10`: il risultato del primo confronto è *true* o *false*; tale valore booleano viene convertito a 1 o 0 (rispettivamente) che viene quindi confrontato con 10 dando *true*. Quindi la condizione è verificata per ogni `x>=0`.
Per testare che x sia nell'intervallo [0,10] si usa:

```
if ( 0 <= x && x <= 10) ...
```

Un errore comune è quello di utilizzare = (assegnazione) al posto di == (uguale) nelle condizioni:

```
if (a=10) ...
```

In questo caso alla variabile a viene assegnato il valore 10 e quindi viene c testata come condizione (*true* se diverso da 0, *false* altrimenti).

### 5.6.6 Operatore di conversione di tipo

Una variabile di un tipo può essere usata come variabile di tipo diverso utilizzando l'operatore di conversione di tipo.
Per eseguire questa operazione occorre preporre alla variabile da convertire il tipo in cui deve essere convertita tra parentesi tonde

```
... (tipo) variable .. ;
```

Esempio: divisione decimale tra due interi.

```
int x=9,y=10;
double res;
res = x/y; //res vale 0 la divisione viene fatta tra interi

int x=9,y=10;
double res;
res = ((double)x)/y; //res vale 0.9
```

### 5.6.7 Operatore ternario

L'operatore ternario valuta un'espressione ed esegue una diversa istruzione a seconda che l'espressione sia vera (istruzione_1) o falsa (istruzione_2):

```
condizione ? istruzione_1 : istruzione_2;
```

Esempio:

```
a > b ? max=a : max=b; // assegna a max il maggiore tra a e b.
```

### 5.6.8 Precedenza degli operatori

La precedenza degli operatori indica l'ordine in cui l'operatore è utilizzato. Ad esempio:

```
int res;
res = 1+2*3;  //res vale 7
```

non è equivalente a

```
int res;
res = (1+2)*3; //res vale 9
```

perché l'operatore * ha precedenza sull'operatore + (quando si hanno dubbi le parentesi assicurano la corretta esecuzione dell'operazione).
In Tavole 5.1 sono riportati i diversi operatori in ordine di precedenza (gli operatori nello stesso gruppo hanno stessa precedenza e precedenza maggiore rispetto a quella degli operatori nel gruppo sottostante); *espr* indica una qualsiasi espressione, *var* una generica variabile e *tipo* un tipo generico.

## 5.7 Istruzioni di controllo

Le istruzioni di controllo specificano l' ordine in cui devono essere eseguite le le istruzioni contenute in un programma C++. Nel seguito con il termine `istruzione` si farà riferimento sia ad una singola istruzione sia a gruppi di istruzioni racchiuse tra parentesi graffe { }.

| | |
|---|---|
| post incremento | var++ |
| post decremento | var-- |
| conversione di tipo | (tipo) espr |
| dimensione di tipo | sizeof(tipo) |
| pre incremento | ++var |
| pre decremento | --var |
| complemento | ~espr |
| negazione | !espr |
| moltiplicazione | espr * espr |
| divisione | espr / espr |
| resto intero | espr % espr |
| addizione | espr + espr |
| sottrazione | espr - espr |
| shift sinistro | espr << espr |
| shift destro | espr >> espr |
| minore | espr < espr |
| minore o uguale | espr <= espr |
| maggiore | espr > espr |
| maggiore o uguale | espr >= espr |
| uguale | espr == espr |
| diverso | espr != espr |
| AND bit a bit | espr & espr |
| OR esclusivo bit a bit | espr ^ espr |
| OR inclusivo bit a bit | espr \| espr |
| AND logico | espr && espr |
| OR inclusivo logico | espr \|\| espr |
| assegnazione | var = espr |
| moltiplicazione ed assegnazione | var *= espr |
| divisione ed assegnazione | var /= espr |
| resto intero ed assegnazione | var %= espr |
| addizione ed assegnazione | var += espr |
| sottrazione ed assegnazione | var -= espr |
| shift sinitro ed assegnazione | var <<= espr |
| shift destro ed assegnazione | var >>= espr |
| AND ed assegnazione | var &= espr |
| OR esclusivo ed assegnazione | var ^= espr |
| OR inclusivo ed assegnazione | var \|= espr |

Tabella 5.1: Tabella riassuntiva sugli operatori. Operatori nello stesso gruppo hanno eguale precdenze e precedenza maggiore rispetto agli operatori dei gruppi sottostanti

### 5.7.1 if - else

L'istruzione di controllo `if - else` è utilizzata per esprimere decisioni. La sintassi è:

```
if (espressione)
   istruzione_1
else
```

*istruzione_2*

L' *espressione* è valutata: se è vera viene eseguita *istruzione_1*, se è falsa viene eseguita *istruzione_2*. La parte contenente `else` è opzionale: si può omettere, ad esempio, quando non si voglia compiere alcuna azione se *espressione* è falsa.

### 5.7.2 if - else if - else

`if (`*espressione*`)`
  *istruzione*
`else if (`*espressione*`)`
  *istruzione*
`else if (`*espressione*`)`
  *istruzione*
...
`else`
  *istruzione*

Una sequenza di `if-else if-else` è la maniera più generale per esprimere decisioni multiple. Le *espressioni* sono valutate una dopo l'altra; se un' *espressione* è vera, l' *istruzione* ad essa associata viene eseguita e la catena di `if` si interrompe. `else` traduce la condizione *"nessuna delle condizioni precedenti"*.

### 5.7.3 switch

L'istruzione di controllo `switch` rappresenta una decisione multipla che confronta il valore di un' espressione con valori costanti interi ed esegue le istruzioni ad essi collegati.

switch (*espressione*){
  `case` *costante*`:`
    *istruzione*
  `case` *costante*`:`
    *istruzione*
  ...
  `default:`
    *istruzione*
`}`

Ogni `case` è identificato da uno o più numeri interi (o espressioni intere costanti). Se un `case` corrisponde al valore di `espressione`, l'esecuzione inizia da quel case. Il case `default` viene eseguito se nessuno degli altri case corrisponde al valore dell'espressione (`default` è opzionale: può essere omesso se non è necessario).
Siccome i case rappresentano etichette che indicano da dove deve partire l' esecuzione, se si desidera che l' esecuzione non prosegua ai case seguenti occorre utilizzare `break` o `case` (vedi Par. 5.7.7). Ad esempio il codice

```
switch (val){
  case 0:
    cout << "case 0 " << endl;
  case 1:
    cout << "case 1 " << endl;
  default:
    cout << "case non trovato" << endl;
```

```
  }
```

per val=0 produce l'output

```
case 0
case 1
case non trovato
```

mentre il codice

```
  switch (val){
    case 0:
      cout << "case 0 " << endl;
      break;
    case 1:
      cout << "case 1 " << endl;
      break;
    default:
      cout << "case non trovato" << endl;
      break;
  }
```

produce, sempre per val=0, l'output

```
case 0
```

### 5.7.4 while

L' istruzione di controllo `while`
`while (`*espressione*`)`
*istruzione*
ripete *istruzione* finché *espressione* resta vera

### 5.7.5 do while

`do`
*istruzione*
`while (`*espressione*`)`
`do while` ha la stessa funzionalità di `while` eccetto che *espressione* viene valutata dopo l'esecuzione di *istruzione*. Ciò garantisce che l' istruzione (o il blocco di istruzioni) siano eseguite almeno una volta.

### 5.7.6 for

`for (`*espr_1*`;` *espr_2*`;` *espr_3*`)`
*istruzione*
La funzionalità di `for` è quella di ripetere *istruzione* finché la *espr_2* resta vera (in modo analogo a `while`). In aggiunta `for` fornisce un' istruzione di inizializzazione ed una di incremento. `for` è quindi adatto all' iterazione di istruzioni dipendenti da un contatore.
Il suo funzionamento può essere cosìschematizzato:

1. inizializzazione (*espr_1*), generalmente costituita dall'inizializzazione di un contatore, viene eseguita una volta sola;

2. l' espressione *espr_2* è valutata, se vera il ciclo continua, altrimenti il ciclo termina;

3. esecuzione di *istruzione*;

4. esecuzione di *espr_3* (molto spesso un'incremento o un decremento);

5. il ciclo ritorna al passo 2.

Il ciclo `for` è equivalente al seguente ciclo `while`

*espr_1*;
`while (`*espr_2*`){`
  *istruzione*
  *espr_3*;
}

Esempio di $n$ iterazioni di un blocco di istruzioni utilizzando come contatore il numero intero `i`

```
for (int i=0; i<n; i++){
  ...
}
```

Le espressioni di inizializzazioni, condizione ed incremento possono essere contenere più variabili:

```
for (int i=0, int j=10; i < j; i++, j--){
  ...
}
```

### 5.7.7 break e continue

Sono istruzioni di controllo utili bloccare l' esecuzione di un ciclo senza modificare la condizione che ne regola la ripetizione.
`break` interrompe immediatamente il ciclo (`while` o `for`) che lo contiene o esce immediamente da uno `switch`.
`continue` (che si agisce solo sui cicli `while` e `for`) causa il passaggio alla prossima iterazione.
Esempio: il codice

```
for (int i=0; i<n; i++){
  if (x[i]<0) break;
  // blocco di istruzioni
}
```

si ferma se un elemento dell' array x è negativo, mentre il codice

```
for (int i=0; i<n; i++){
  if (x[i]<0) continue;
  // blocco di istruzioni
}
```

esegue il blocco di istruzioni solo sugli elementi positivi o nulli dell'array x

## 5.8 Puntatori

Un puntatore è una variabile il cui contenuto è l'indirizzo in memoria di un' altra variabile. Dato un tipo `T`, `T*` è di tipo "puntatore a `T`". Una variabile di tipo `T*` può contenere l' indirizzo di una variabile di tipo `T`.

```
double* pa; // pa e' un puntatore a double
int* pj;    // pj e' un puntatore a int
```

La dichiarazione di un puntatore non implica alcuna inizializzazione: per evitare di compiere operazioni su indirizzi di memoria non determinati dall' utente è utile inizializzare il puntatore ad un valore invalido (NULL indica lo 0 per i puntatori).

```
int *p=NULL;
```

Le operazioni fondamentali sui puntatori sono:

- la dereferenziazzione: operazione che ritorna il valore contenuto nella locazione di memoria il cui indirizzo è dato dal puntatore, l'operatore di deferenzazzione è l' operatore unario (prefisso) $*$;

- la referenziazzione: operazione che ritorna l' indirizzo di una variabile, l'operatore di referenzazzione è l' operatore unario (prefisso) &;

```
int i=10;
int *ip;
ip = &i;  // l'indirizzo di i viene assegnato al puntatore a intero ip
cout << *ip << endl;  // stampa del valore contenuto all'indirizzo ip
```

### 5.8.1 Array e puntatori

In C++ i puntatori e gli array sono intimamente collegati. Il nome di un array può essere utilizzato come puntatore alla suo primo elemento

```
double x[3]={7.,4.,3.};
double* p1 = x;      // puntatore al primo   elemento
double* p2 = &x[0];  // puntatore al primo   elemento
double* p3 = &x[1];  // puntatore al secondo elemento
```

In modo simmetrico un puntatore a cui sia assegnato l'indirizzo del primo elemento di un array (o in generale l' indirizzo dell' inizio di una zona di memoria contenente una sequenza di variabili dello stesso tipo) può essere utilizzato come un array

```
double x[3]={7.,4.,3.};
double* p = x;     // puntatore al primo   elemento
p[2] = 4.0;        // accesso al terzo elemento e sua riassegnazione
```

### 5.8.2 Algebra dei puntatori

L' accesso agli elementi di un array può essere realizzato tramite indice o tramite puntatore. Ad esempio i codici

```
double v[4]={3.,4.,5.,1.}
for (int i=0; i<4; i++)
  cout << v[i] << endl;
```

e

```
double v[4]={3.,4.,5.,1.}
for (double* p=v; p<=&v[3]; p++)
  cout << *p << endl;
```

sono equivalenti.
Il risultato di un operatore aritmetico `+`, `-`, `++` o `--` ad un puntatore dipende dal tipo del puntatore. Dato un puntatore `p` di tipo `T`*, assegnato a `p` un indirizzo di un elemento di un array di tipo `T`, `p+1` punterà all' elemento successivo, `p-1` punterà all' elemento precedente. Questo implica che il valore intero di `p+1` sarà maggiore del valore intero di `p` di `sizeof(T)` unità (corrispondente allo spazio occupato in memoria dal singolo elemento dell' array).
Complicate operazioni aritmetiche sui puntatori sono in genere inutili e vanno evitate.

## 5.9 Allocazione dinamica della memoria

L' allocazione dinamica permette di gestire l' allocazione in memoria delle variabili ottimizzandone l' uso, in particolare consente di creare array e matrici di dimensioni che sono note solo durante dell'esecuzione del programma.

### 5.9.1 Operatore new

L' operatore unario `new` alloca uno spazio in memoria e ne restituisce l' indirizzo.
*puntatore* `= new tipo[`*dimensione*`];`

- `tipo` è il tipo della variabile che si vuole allocare (*puntatore* deve essere un puntatore allo stesso tipo);
- *dimensione* è il numero di elementi consecutivi (dello stesso tipo) che si vogliono allocare in memoria (se la dimensione non è specificata viene allocato un singolo elemento)

In caso di errore (memoria non disponibile) `new` restituisce `NULL`.

### 5.9.2 Operatore delete

L'operatore unario `delete` *dealloca* (libera) la memoria puntata dal puntatore su cui agisce.
`delete` *puntatore*`;`
`delete` non restituisce alcun valore e quindi costituisce da solo un' istruzione.
Contrariamente all' apparenza l' operatore `delete` non cancella la variabile puntatore e neppure altera il suo contenuto: l' unico effetto è di liberare la memoria puntata rendendola disponibile per ulteriori allocazioni.
Se il puntatore punta ad un' area di memoria in cui sono stati allocati più elementi, per liberare tutta l' area occorre aggiungere una coppia di parentesi quadre dopo l'operatore:
`delete[]` *puntatore*`;`

### 5.9.3 Allocazione dinamica di array

```
int n=4;
double* p = new double[n]; // alloca lo spazio di memoria necessario
                           // per 4 double e restituisce il puntatore
                           // all'inizio di tale spazio che viene
                           // assegnato a p
// ... utilizzo di p

delete[] p;
```

Nota: l' istruzione `delete p` dealloca solo la memoria occupata dal primo elemento dell' array.

### 5.9.4 Allocazione dinamica di matrici

Un puntatore può contenere l' indirizzo di un' altro puntatore (e così di seguito). Un' importante applicazione di questa regola generale consiste nell' allocazione dinamica di matrici.

```
int nrow, ncol;
double    **matrix;

//Allocazione
matrix = new double*[nrow];
for(int i=0; i<nrow; ++i)
    matrix[i] = new double[ncol];

//Utilizzo della matrice matrix (elemento generico matrix[i][j])

//Deallocazione
for(int i=0; i<nrow; ++i)
    delete[] matrix[i];
delete[] matrix
```

Analizziamo il codice sopra riportato:

- Definizione: una matrice è una struttura a righe e colonne; può quindi essere pensata come un array di puntatori alle righe che la compongono, ogni riga è poi, a sua volta, un array di elementi. Per questo una matrice di tipo `T` è dichiarata come "puntatore a puntatore a T" (`T**`).

- Allocazione: innanzitutto si alloca l' array di puntatori (tipo `T*`) con lunghezza il numero di righe. Deferenziando il puntatore a tale array (`matrix[i]`) si ottengono i puntatori a ciascun array di riga (tipo `T`) che vengono allocati al numero di colonne (lunghezza di riga).

- Utilizzo: l' elemento ij di una matrice si ottiene dereferenziando il puntatore `matrix` che fornisce il puntatore a ciascuna riga `matrix[i]` e quindi dereferenziando ancora tale puntatore per ottenere l' elemento j all'interno della riga i `matrix[i][j]`.

- Deallocazione: si disallocano prima i puntatori agli array che compongono le righe e poi l' array di puntatori.

È importante notare che, al contrario dell'array, il nome di una matrice allocata staticamente non è equivalente alla struttura "puntatore a puntatore" di una matrice dinamica:

```
double mat[2][3];
double **x;
x = mat;          // Non consentito
```

ciò dipende dal fatto che le due strutture sono allocate in maniera diversa in memoria: una matrice statica di fatto allocata come un array, con tutti gli elementi contigui mentre in una matrice dinamica sono contigui solo gli elementi della stessa riga (non le diverse righe tra loro). In ultimo si noti che la matrice dinamica ha una struttura più generale di una matrice: ciascuna riga può infatti può essere allocata con lunghezza diversa.

## 5.10 Funzioni

In un programma spesso accade di dover eseguire piu' volte una stessa sequenza di operazioni. Questo significa dover riscrivere le stesse istruzioni piu' volte.
In C++, come in altri linguaggi di programmazione, è possibile racchiudere un insieme di istruzioni in un blocco esterno al programma principale, assegnandogli un nome, in modo da poterlo eseguire semplicemente facendo riferimento al suo nome.
Il blocco di istruzioni così definito si chiama *funzione*.
Un esempio frequente nella programmazione scientifica è il calcolo di operazioni matematiche irrazionali, come la radice quadrata; questo tipo di operazioni, che vengono effettuate tramite approssimazioni numeriche, spesso sono eseguite più volte all'interno di un programma; risulta molto efficiente scrivere una sola volta il codice che le esegue, dandogli un nome (per la radice quadrata: **sqrt()**), e poi richiamare questa funzionalità ogni volta che serve, tramite la sola istruzione:

```
double radice = sqrt(10.3);
```

L'utilizzo delle funzioni permette di scrivere le istruzioni relative ad una certa operazione solo una volta, ed elimina la necessità di riscrivere lo stesso codice in piu' parti del programma, riducendo in questo modo la dimensione del sorgente e confinando in un unico blocco eventuali errori di codifica; questo rende più veloce la scrittura del codice, e molto più semplici le fasi di correzione o modifica del programma.
In generale, anche se non c'è necessità di riutilizzare il codice, è vantaggioso separare le operazioni di un programma isolandole in funzioni, in modo da migliorare la leggibilità del codice. È quindi opportuno effettuare una analisi del problema da risolvere cercando di suddividere in funzioni le operazioni da effettuare.
Questa tecnica di programmazione si chiama *programmazione strutturata*.

### 5.10.1 Esecuzione di una funzione

In C++ le funzioni vengono richiamate con una istruzione che ne indica il nome seguito da parentesi tonde. Le parentesi indicano al compilatore che quello che desideriamo fare è eseguire il codice della funzione che ha quel nome, che deve essere stata definita o dichiarata in precedenza.
Le funzioni possono richiedere alcuni parametri per effettuare le operazioni desiderate: nell'esempio di **sqrt()**, la funzione richiede come parametro il valore numerico di cui calcolare la radice. Questi parametri, che chiameremo **argomenti della funzione**, devono essere specificati tra le parentesi tonde e separati da virgole. Gli argomenti possono essere delle costanti, delle variabili del tipo opportuno, o delle operazioni tra costanti e variabili.
Le funzioni possono eventualmente ritornare un risultato, sotto forma di un valore di un qualche tipo, che può essere memorizzato in una variabile del tipo opportuno tramite una istruzione di assegnazione. Nell'esempio appena visto, la funzione **sqrt()** ritorna il risultato sotto forma di **double**, e questo può essere assegnato ad una variabile di tipo double ponendo la chiamata a funzione come membro a destra di una istruzione di assegnazione. Il tipo di valore ritornato può essere un qualsiasi tipo del C++ (**int**, **double**, etc.); in C++ anche le funzioni che non ritornano alcun valore hanno un tipo, che si chiama **void**.

### 5.10.2 Definizione di una funzione

La **definizione** di una funzione consiste nella scrittura del codice che la caratterizza.
Una funzione viene definita specificando il tipo di valore ritornato, il nome seguito da parentesi

tonde e le istruzioni che vengono eseguite ogni qualvolta la funzione è chiamata, racchiuse in un blocco (cioè tra {}). Nel blocco possono essere inserite tutte le istruzioni del C++, quali assegnazioni, definizioni di variabili, istruzioni di controllo di flusso, chiamate ad altre funzioni, etc.

```
void Ciao() {
   cout << "Ciao" << endl;
}
```

L'esempio definisce una funzione di tipo **void**, cioè che non ritorna nessun tipo di valore, di nome **Ciao**. La funzione scrive su terminale un messaggio.
Si può usare questa funzione con una istruzione che specifica il suo nome, seguito da parentesi tonde:

```
main() {
   Ciao();
}
```

Va osservato che la chiamata a funzione, essendo una istruzione, va terminata con il ;.

### 5.10.3 Argomenti passati ad una funzione

Una funzione può ricevere degli argomenti che possono essere utilizzati nel blocco della funzione stessa. Gli argomenti devono essere indicati tra le parentesi tonde che seguono il nome della funzione, specificandone il tipo ed il nome che verrà usato per riferisri ad essi nel corpo della funzione stessa, e separandoli con virgole.

```
void Ciao(string nome) {
   cout << "Ciao" << nome << endl;
}

main() {
   string s = "Mario";
   Ciao(s);
}
```

In questo esempio, la funzione richiede che le sia passato un parametro di tipo **string**. La definizione dell'argomento vale come definizione di variabile all'interno della funzione, quindi non deve essere ridefinita al suo interno.
Va osservato come non ci sia relazione tra il nome dell'argomento utilizzato nella funzione ed il nome della variabile che viene utilizzata quando la funzione viene chiamata. Nell'esempio, il **main()** chiama la funzione **Ciao()** passandogli come argomento la variabiler **s**, che ha per valore "Mario"; questo fa si che nelle istruzioni della funzione in corrispondenza a questa chiamata, la variabile **nome** avrà per valore "Mario".
È comunque lecito passare come argomento ad una funzione una variabile che ha lo stesso nome utilizzato nella definizione della funzione stessa.

### Argomenti passati per valore

Gli argomenti vengono passati ad una funzione per valore (*by value*); questo significa che, in corrispondenza della chiamata alla funzione, vengono create nuove locazioni di memoria per poter contenere gli argomenti della funzione, e vengono copiati in queste zone i valori delle variabili nella chiamata alla funzione.
Poiché le zone di memoria che contengono la variabile del chiamante e la variabile della funzione sono diverse, eventuali modifiche fatte al valore della variabile nel corpo della funzione non vengono trasferite alla variabile del chiamante.

```
void Quadrato(double d) {
   d = d*d;
   cout << "Nella funzione d = " << d << endl;
}

main() {
   double d = 5.0;
   Quadrato(d);
   cout << "Nel main: d = " << d << endl;
}
```

L'esecuzione di questo programma mostra come il valore della variabile **d** all'interno della funzione sia 25, mentre nel main(), anche dopo la chiamata a Quadrato(), il valore di **d** resta 5.

### Argomenti passati per puntatore

Per poter eseguire all'interno di una funzione operazioni che comportino modifiche dei valori degli argomenti in modo da trasferire le modifiche anche alle variabili del chiamante, è necessario utilizzare come argomento un puntatore alla variabile:

```
void Quadrato(double *d) {
   *d = (*d)*(*d);
   cout << "Nella funzione *d = " << *d << endl;
}

main() {
   double a = 5.0, *p = &a;
   Quadrato(p);
   cout << "Nel main: a = " << a << endl;
}
```

In questo caso, la funzione richiede come argomento un puntatore. La funzione riceve quindi l'indirizzo di una variabile, e poi modifica non il valore dell'argomento, ma il contenuto della zona di memoria puntata dall'argomento (cioè **a**). In questo modo, dopo l'esecuzione della funzione, il valore di **a** risulterà modificato.

### Argomenti passati per referenza

Consideriamo il seguente esempio:

```
void Quadrato(double &d) {
   d = d*d;
   cout << "Nella funzione d = " << d << endl;
}

main() {
   double a = 5.0;
   Quadrato(a);
   cout << "Nel main: a = " << a << endl;
}
```

In questo caso la funzione accetta come argomento una **double &**, cioè una referenza ad una double, il che significa che il nome **d** nella funzione non indica una double nuova a cui viene assegnato il valore del parametro specificato, ma è un nome che indica **la stessa variabile specificata come argomento nella chiamata alla funzione**. In sostanza quello che accade è che alla funzione viene passato l'indirizzo dell'argomento - come avviene con i puntatori - ma nella funzione al nome **d** corrisponde una variabile di tipo **double**, non un puntatore, ed allo stesso modo il chiamante dovrà specificare come argomento una variabile di tipo **double** e non un puntatore.
Naturalmente, poiché la variabile della funzione fa riferimento alla stessa zona di memoria della variabile usata come argomento nella chiamata, una modifica al suo valore nella funzione comporterà una modifica anche alla variabile del chiamante.

C'e' una certa analogia tra il passare argomenti per puntatore o per referenza: in entrambi i casi si rende possibile operare modifiche sulle variabili del chiamante, ed entrambi - in occasione di tipi di variabile caratterizzate da grande occupazione di memoria - risultano molto efficienti in quanto si copia un indirizzo (4 bytes) e non il valore dell'argomento, che può essere piu' grosso. La differenza tra i due metodi è che nel primo caso si deve utilizzare un puntatore, che va dereferenziato ogni volta che ci si vuole riferire al valore della variabile, nel secondo caso non si usano puntatori.
Va osservato che molti programmatori utilizzano la tecnica di passare argomenti per referenza solo per motivi di efficienza, quando non è previsto che vengano apportate modifiche al valore della variabile, in caso contrario utilizzano la tecnica di argomenti passati per puntatori; tuttavia, questa è solo una pratica comune: le regole del C++ ci lasciano liberi di agire a nostro piacere.

### Conversione implicita di tipo degli argomenti

Come già visto, il compilatore C++ è in grado di effettuare implicitamente conversioni di tipo. Quando una funzione viene chiamata con un argomento di tipo diverso da quello specificato nella definizione della funzione stessa, se il compilatore può effettuare la conversione, questa viene effettuata implicitamente. Ad esempio:

```
void Quadrato(double d) {
   d = d*d;
   cout << "d = " << d << endl;
}

main() {
   double d = 5.0;
```

```
    int i = 3;
    Quadrato(d);
    Quadrato(i);
}
```

In questo caso, al momento di eseguire l'istruzione **Quadrato(i)**, viene operata una conversione implicita di **i** da **int** a **double**, e quindi viene eseguita la funzione.
Va sottolineato che tali conversioni non sono quasi mai possibili tra puntatori a diversi tipi: se si tenta di fare ciò il compilatore darà un errore.

### Argomenti array

Come già visto, il nome di una variabile array viene all'occasione implicitamente convertito in un puntatore al primo elemento dell'array. In questo modo, quando una funzione richiede come argomento un array, viene passato il puntatore, e le operazioni sull'array effettuate all'interno della funzione andranno a modificare gli elementi dell'array del chiamante; passare argomenti di tipo array è quindi a tutti gli effetti come passare argomenti puntatore.
L'argomento array va specificato utilizzando le parentesi [] senza indicare la dimensione:

```
double SommaArray1(double v[], int dim) {
    double sum = 0;
    for(int i=0; i<dim; ++i) sum += v[i];
    return sum;
}
```

Capita molto spesso che la funzione sia definita specificando come parametro non un array ma un puntatore. Il comportamento è equivalente:

```
double SommaArray2(double *v, int dim) {
    double sum = 0;
    for(int i=0; i<dim; ++i) sum += v[i];
    return sum;
}
```

### Numero di argomenti non specificato

In alcuni casi non è possibile stabilire a priori il numero ed il tipo di argomenti specificato. In questo caso è possibile definire la funzione utilizzando come argomento tre punti consecutivi:

```
int execl(char * ...) {
```

Questa definizione dice che la funzione deve essere chiamata con un parametro di tipo char *, seguito opzionalmente da altri parametri di tipo non specificato. La funzione potrà quindi essere chiamata anche nel corso dello stesso programma con una serie di argomenti differente volta per volta, fatto salvo che il primo parametro, nell'esempio, deve essere un **char ***.
È naturalmente compito del programmatore scrivere il codice della funzione in modo da trattare tutti i casi possibili in modo corretto.

### 5.10.4 Valore di ritorno di una funzione ed istruzione return

Come detto, le funzioni possono ritornare un valore, il cui tipo è specificato nella definizione. In questo modo il chiamante può assegnare il valore di ritorno ad una sua variabile. L'istruzione per ritornare un valore è **return <valore>**. Qualora la funzione sia di tipo **void**, l'istruzione **return** non deve specificare alcun valore.

```
double Quadrato(double d) {
   return d*d;
}

main() {
   double d = 5.0, d2;
   d2 = Quadrato(d);
}
```

Nell'esempio, l'esecuzione della funzione ritorna una double, il cui valore è il quadrato del valore dell'argomento, e questo valore viene assegnato alla variabile **d2** nel **main()**.
Anche in questo caso valgono le regole della conversione implicita, per cui è possibile assegnare il valore di ritorno, ad esempio, di una funzione di tipo **int** ad una variabile di tipo **double**. Si deve naturalmente fare attenzione a ciò che si fa: la conversione inversa, ad esempio, comporta il troncamento della parte decimale del valore ritornato.
L'istruzione **return** può comparire in qualunque punto della funzione; in corrispondenza di tale istruzione l'esecuzione abbandona il codice della funzione, e prosegue con le istruzioni successive alla chiamata della funzione.

```
double MiaPow(double base, double exp) {
   if(base == 0 && exp == 0)
      return 1.0;
   return pow(base, exp);
}
```

Nell'esempio, la funzione esegue un controllo sugli argomenti, e ritorna il valore 1.0 in caso di condizione indeterminata. Altrimenti esegue il calcolo ricorrendo a sua volta alla funzione di libreria **pow()** e ritorna il valore calcolato.

### 5.10.5 Funzioni recursive

Il C++ permette che una funzione possa chiamare se stessa generando una sequenza di chiamate della stessa funzione. Un esempio classico è il calcolo del fattoriale di un intero:

```
int fattoriale(int n) {
   if(n == 1) return 1;
   return n*fattoriale(n-1);
```

Nell'esempio la funzione **fattoriale()** chiamam se stessa con argomento il cui valore decresce di volta in volta, fino ad arrivare alla chiamata con argomento **1** che interrompe la catena di chiamate recursive.
Si osservi che la variabile **n** è diversa per ogni chiamata della funzione: infatti alla chiamata di una funzione viene creata in memoria una zona in cui la funzione chiamata opera, e questa

zona di memoria è diversa volta per volta (dipende non dalla funzione in se, ma dalla specifica chiamata alla funzione).
Quando si scrivono funzioni a chiamata recursiva, si deve prestare molta attenzione a fare in modo che la catena di chiamate si interrompa, altrimenti si incorrerà in un errore a *run-time*.

### 5.10.6 Dichiarazione delle funzioni: il prototipo della funzione

Spesso capita di dover utilizzare una funzione che non è scritta nel nostro file sorgente (ad esempio una funzione di libreria, come **sqrt()**); in altre circostanze potremmo dover utilizzare una funzione prima che questa sia definita nel nostro sorgente.
Tuttavia in C++, come per tutti gli altri nomi utilizzati in un programma, anche i nomi delle funzioni devono essere dichiarati prima di poter essere utilizzati.
In tale circostanza è necessaria una dichiarazione che specifichi quali sono le caratteristiche che definiscono quel nome, vale a dire che rappresenta una funzione, il tipo e gli argomenti. Questa si chiama *dichiarazione del prototipo* della funzione:

```
double sqrt(double d);
main() {
   double a = sqrt(12.3);
   ...
```

Il prototipo della funzione corrisponde di fatto alla prima riga della definizione della funzione stessa; va osservato come, a differenza della definizione della funzione, che è seguita dal blocco delle istruzioni, la dichiarazione vada terminata dal simbolo ";".
Nell'esempio, viene **dichiarato** il prototipo della funzione **sqrt()**, che quindi può essere poi utilizzata nel **main()**. Se non ci fosse la dichiarazione, il compilatore lamenterebbe il fatto che viene utilizzata una funzione sconosciuta. Sarà poi un altro pezzo del g++ (il linker) che si occuperà di andare a vedere se e dove viene **definita** la funzione **sqrt()**.
Si può capire ora quale sia il motivo delle istruzioni **#include** che abbiamo visto fin'ora: quando ad esempio si utilizzano funzioni matematiche quali **pow()**, **sqrt()**, **sin()**, etc., l'istruzione **#include <cmath>** include un file in cui sono dichiarati tutti i prototipi delle funzioni definite nella libreria matematica del C++; allo stesso modo, l'include file **<iostream>** contiene tutte le dichiarazioni relative agli oggetti utilizzati nell'I/O da terminale, quali **cout**, **cin**, **endl**, etc..

Nota: poiché la dichiarazione del prototipo di una funzione non comporta l'utilizzo degli argomenti (a differenza della definizione, dove gli argomenti vengono utilizzati nel corpo della funzione stessa) ma solo la specificazione del loro tipo, la definizione di un nome per gli argomenti è opzionale, e può essere omessa. Ad esempio, le due dichiarazioni

```
double pow(double x, double y);
double pow(double, double);
```

sono del tutto equivalenti.
Spesso negli include file relativi a funzioni di libreria le dichiarazioni vengono fatte nella seconda forma.

#### Dichiarazione con argomenti a valore predefinito

Spesso una funzione viene definita con più argomenti di quanti non siano necessari nella maggior parte dei casi. Ad esempio, si pensi di scrivere una funzione per stampare interi in una base scelta dal chiamante:

```
void Print(int numero, int base);
```

La maggior parte delle volte la funzione verrà chiamata specificando **10** come base, costringendo così gli utenti della funzione a specificare sempre un secondo argomento usualmente non necessario.
Il C++ ci permette di indicare, al momento della **dichiarazione**, un valore di default da assegnare ad un argomento di una funzione:

```
void Print(int numero, int base = 10);
```

In questo modo saranno possibili chiamate del tipo:

```
Print(23, 16);
Print(23); // equivalente a Print(23, 10)
```

Nel primo caso avremo una stampa del numero **23** in esadecimale, nel secondo caso la funzione **Print()** verrà chiamata assegnando al secondo argomento il valore predefinito **10**, ed avremo una stampa in base 10.

Nota: non c'e' relazione tra la assegnazione di un valore predefinito all'atto della definizione di una funzione, o all'atto della sua dichiarazione: il valore predefinito può comparire in una, nell'altra o in entrambe; quando si utilizza una funzione, questa è caratterizzata dalla sua definizione (se compare prima nel nostro sorgente) o dalla dichiarazione del prototipo (in caso contrario). L'utilizzo delle variabili predefinite può essere fatto solo in base alla caratterizzazione della funzione nel momento in cui questa viene chiamata. Se la dichiarazione non contiene la definizione di una variabile predefinita, non si può omettere la variabile in questione nella chiamata anche se nella definizione della funzione (che si trova altrove) la variabile ha un valore predefinito. Il codice seguente:

```
void Print(int numero, int base);
main() {
   Print(10);  // errore: il prototipo di Print() chiede due interi
}
void Print(int numero, int base = 10) {
   ...  // corpo della funzione
}
```

risuterà in un errore di compilazione.
L'assegnazione di valori predefiniti agli argomenti di una funzione può riguardare più di un argomento:

```
double Fun(double, int, int = 10, string = "ciao", char * = NULL);
```

tuttavia non è possibile dichiarare un prototipo con un parametro a valore predefinito che preceda in posizione un parametro senza valore predefinito. Ad esempio:

```
double Fun(double a, double b = 0.5, double c, double d = 3.0);
```

risulterà in un errore di compilazione.
È quindi possibile dichiarare valori predefniti solo agli ultimi argomenti delle funzioni, ed eventualmente a tutti.

### 5.10.7 Overload delle funzioni

In C++ è possibile definire due funzioni che hanno lo stesso nome, ma prototipo diverso per numero di argomenti o per il loro tipo. In sede di compilazione, le due funzioni vengono considerate come funzioni differenti, e quando viene chiamata una delle due funzioni, viene scelta quella giusta in base al tipo degli argomenti con cui la funzione viene chiamata.

```
void Print(int);
void Print(char *);
int main() {
   Print(10); // viene eseguita la prima Print()
   Print("Ciao, mondo"); // viene eseguita la seconda Print()
```

Questa tecnica viene chiamata *overload delle funzioni*, ed è spesso utilizzata per poter usare la stessa forma sintattica in casi in cui vanno effettuate operazioni concettualmente uguali, ma che necessitano di istruzioni differenti.
L'utilizzo di overload di funzioni va pienificato ed attuato con molta attenzione; vi sono infatti situazioni in cui è facile incorrere in errori. Si pensi ad esempio alle funzioni:

```
void Fun(int, double);
void Fun(double, int);
```

Il compilatore formalmente accetta la dichiarazione, in quanto le funzioni hanno prototipo differente. Tuttavia se si inserisce nel codice l'istruzione:

```
double d = Fun(10,5);
```

la funzione viene chiamata con argomenti **(int, int)**. Tuttavia il compilatore non è in grado di capire se fare una conversione implicita da **int** a **double** del primo argomento e chiamare la seconda **Fun()**, o fare la conversione implicita da **int** a **double** del secondo argomento e chiamare la prima **Fun()**; si avrà un errore che ci costringerà a rivedere la struttura del nostro codice.
Va infine osservato che il valore di ritorno non è preso in considerazione nella ricerca della funzione corretta da richiamare: il processo di overload e di identificazione della funzione corretta riguarda solamente gli argomenti. Nondimeno, diverse funzioni con lo stesso nome possono, all'occorrenza, ritornare tipi diversi.

### 5.10.8 Puntatori a funzione

In C++ è possibile definire un puntatore di tipo *puntatore a funzione* di un certo prototipo. Questa variabile è destinata a contenere come valore l'indirizzo di una funzione che ha uno specifico prototipo. Ad esempio:

```
void Ciao(string s) {
   cout << "Ciao " << s << endl;
}

int main() {
   void (*p) (string);
   p = &Ciao; // assegnazione a p dell'indirizzo di Ciao()
   (*p)("Mario"); // esecuzione della funzione Ciao()
}
```

Nell'esempio abbiamo definito **p** come puntatore a funzione di tipo **void** che accetta come argomento una **string**. La sintassi, apparentemente complessa, non è altro che la dichiarazione del prototipo di una funzione, in cui al posto del nome si usa un puntatore, racchiudendolo tra parentesi tonde per evitare ambiguità.
Una volta definito il puntatore, è possibile assegnargli l'indirizzo di una funzione, che si esprime utilizzando l'operatore **&** prefisso al nome della funzione in questione.
Va osservato che l'assegnazione deve essere fatta omettendo le parentesi tonde dopo il nome della funzione, altrimenti il significato della istruzione sarebbe: "esegui la funzione Ciao(), ed assegna l'indirizzo del valore di ritorno alla variabile p", cosa che comporterebbe un errore in compilazione o in esecuzione.
A questo punto si può richiamare l'esecuzione della funzione dereferenziando il puntatore, tramite l'operatore prefisso *****, seguito dalle parentesi tonde che racchiudono gli eventuali parametri. In sostanza, in ogni circostanza in cui utilizzeremmo il nome **Ciao** possiamo utilizzare ***p**.
Un possibile utilizzo dei puntatori a funzione è nella possibilità di passare una funzione come parametro ad un'altra funzione: si pensi ad una funzione che, per eseguire le sue operazioni, debba chiamare una ulteriore funzione, che però non è la stessa a seconda dei casi. Tale funzione può esserle passata di volta in volta come argomento.
Ad esempio una funzione che ordina un array, e che a seconda del tipo degli elementi dell'array dovrà utilizzare una ulteriore funzione che le dica se un elemento è *maggiore* di un altro. La funzione che ordina gli elementi non deve necessariamente sapere come ordinare elementi di un certo tipo: è sufficiente che le venga passato come argomento il puntatore alla funzione che, a seconda del tipo, si occupi di valutare quale tra due elementi sia il maggiore.

Nota: il nome di una funzione, al pari del nome di un array, può venire implicitamente convertito a puntatore ad una funzione di prototipo uguale al suo. Quindi l'assegnazione vista nell'esempio precedente può essere eseguita omettendo l'operatore prefisso **&**:

```
p = Ciao;
```

Allo stesso modo, il puntatore a funzione può essere utilizzato in luogo della funzione senza essere dereferenziato, omettendo quindi l'operatore prefisso *****:

```
p("Mario"); // esecuzione della funzione Ciao()
```

Essendo queste sintassi più concise, senza complicare la leggibilità del codice, usualmente sono preferite.

### 5.10.9 La funzione main()

Dal punto di vista sintattico, **main()** è una funzione come tutte le altre, quindi deve avere un tipo, e può avere degli argomenti. Tuttavia main() è una funzione particolare, e può essere di tipo **int** o **void**; nel primo caso, l'istruzione **return** deve specificare un valore, nel secondo caso l'istruzione **return** non deve avere valore specificato.
L'istruzione **return** che precede la fine del main() è opzionale. Se l'istruzione viene omessa ed il main() è definito come **int**, viene ritornato il valore 0.
Se il main() viene dichiarato senza specificarne il tipo, viene considerato di tipo **int**. In questo caso una eventuale istruzione **return** senza l'indicazione di un valore di ritorno viene considerata dal compilatore un errore.

### Esecuzione di un programma ed argomenti passati alla funzione main()

Grossolanamente parlando, quando si manda in esecuzione un programma (ad esempio se diamo da shell un comando, come **ls**), il sistema operativo legge il comando dato, cerca l'eseguibile che abbiamo chiamato, cerca nell'eseguibile la funzione **main()** e crea un processo che comincia ad eseguire le istruzioni del main().
È possibile inviare al main() degli argomenti, specificandoli nella riga di comando (ad esempio, dando da shell il comando **ls -l**, passiamo al main() di ls un argomento, di valore **-l**). Il sistema operativo chiamerà in questo caso la funzione main() passandogli due argomenti: un intero (argc) che indica il numero di "parole" con cui è stato richiamato l'eseguibile (compreso il nome stesso dell'eseguibile: nel caso di **ls -l** argc varrà 2), ed un char ** (argv) che rappresenta un array di char*, di lunghezza argc.
Ogni elemento dell'array contiene un puntatore ad una C-string rappresentante nell'ordine le "parole" con cui è stato eseguito il comando da shell (nell'esempio, argv[0] sarà la stringa "ls", argv[1] sarà la stringa "-l").
Per poter utilizzare gli argomenti passati in questo modo, il main() dovrà essere definito nel seguente modo:

```
main(int argc, char ** argv)
```

### Completamento di un programma e valore di ritorno del main()

In molti casi, è desiderabile comunicare a chi esegue un programma se qualcosa è andato storto. Per fare questo è possibile definire il main() come tipo **int**, e ritornare un valore opportuno in caso di errore. Il sistema operativo si accorgerà di questo fatto e comunicherà al processo che ha eseguito il programma il valore di ritorno del main (assegnando il valore di ritorno ad una opportuna variabile di environment della shell).

```
int main() {
   ...
   if(error) return 1;
   ...
   return 0;
}
```

Nell'esempio, se **error** è vero, il programma si interrompe, e ritorna alla shell il valore 1, altrimenti il programma arriva in fondo e ritorna 0.
In generale, in Unix, tutti i comandi di sistema (ls, rmdir, cp, etc.) sono programmi che ritornano un valore diverso da 0 in caso di errore, e 0 in caso di successo.

### Completamento di un programma in funzioni diverse dal main()

Talvolta è necessario interrompere l'esecuzione di un programma quando si stanno eseguendo le istruzioni di una funzione. Non sempre è possibile ritornare al **main()** - ad esempio se si verifica un errore in una funzione non direttamente chiamata dal **main()** - ma le condizioni richiedono una interruzione immediata.
Questo è possibile attraverso una funzione il cui prototipo è

```
void exit(int status);
```

dichiarata nell'include file **<stdlib.h>**. La chiamata alla **exit()** può essere effettuata in qualunque punto del programma, e provoca la terminazione del programma in modo analogo alla istruzione **return** nel main(). Se viene passato un parametro intero, questo viene ritornato al processo che ha eseguito il programma.

## 5.11 Programmazione modulare e librerie

### 5.11.1 Programmazione modulare

Generalmente un programma non banale é costituito da svariate migliaia di linee di codice, ma non è raro doverne scrivere decine o continaia di migliaia. In queste condizioni non è possibile realizzare un unico file sorgente, per svariati motivi: un file del genere comporterebbe una notevole difficoltà solo per essere editato; inoltre sarebbe di fatto impossibile per un singolo programmatore gestire un programma di grosse dimensioni, ed utilizzando un unico file sorgente diversi programmatori non potrebbero effettuare contemporaneamente modifiche al codice; inoltre dividere il codice in più file facilita le operazioni di correzione e modifica del programma.
Usualmente un programma è costituito da diversi file sorgente, ciascuno dei quali contiene le definizioni di una parte delle funzioni utilizzate nel programma; generalmente le funzioni vengono raggruppate secondo un qualche criterio logico, ad esempio separando le funzioni di carattere matematico da quelle che gestiscono l'I/O, e così via.
Questa tecnica di scrittura del codice si chiama **programmazione modulare**, e più che una opzione è una scelta obbligata al crescere delle dimensioni del programma.
La programmazione modulare è resa possibile dal fatto che la creazione di un eseguibile a partire dai sorgenti è una operazione che avviene in due fasi distinte: la **compilazione** ed il **link**, che verranno descritte nei prossimi paragrafi.

### 5.11.2 La compilazione

La compilazione viene eseguita da un programma chiamato compilatore, ed è specifico del linguaggio di programmazione utilizzato per scrivere il codice.
In questa fase un sorgente viene analizzato sintatticamente, secondo le regole del linguaggio di programmazione utilizzato, e viene prodotto un **object file** che rappresenta la traduzione del sorgente in linguaggio macchina, cioè in istruzioni che il processore è in grado di eseguire. Generalmente vengono eseguite operazioni di ottimizzazione finalizzate a fornire migliori prestazioni in esecuzione, e, se richieste, vengono memorizzate informazioni aggiuntive relative ai nomi di variabili e funzioni utilizzati nel sorgente per future azioni di *debugging*.
Durante la compilazione non vengono effettuati controlli relativi alla esistenza delle definizioni di eventuali funzioni dichiarate e utilizzate nel sorgente compilato, ma che potrebbero essere definite altrove.
Questa è la caratteristica che ci permette di spezzare il sorgente in diversi file, ciascuno dei quali contiene la definizione solo di una parte delle funzioni utilizzate.
È importante sottolineare che l'object file, anche se scritto in linguaggio macchina, **non è eseguibile**, in quanto manca ancora di informazioni essenziali che verranno aggiunte dal **link**.

Dato un file sorgente, ad esempio **programma.cpp**, l'istruzione per compilare il sorgente **senza eseguire il link** è la seguente:

```
g++ -c programma.cpp
```

L'opzione **-c** dice al compilatore di limitarsi a generare l'object file, che si chiamerà con lo stesso nome del sorgente, ma con l'estensione cambiata da **.cpp** a **.o** (nell'esempio otterremo un file di nome **programma.o**).
Il compilatore **g++** accetta molte altre opzioni: alcune le vedremo in seguito, per le altre si rimanda al manuale relativo.

Come detto, il file sorgente non deve necessariamente contenere la definizione di tutte le funzioni utilizzate, a patto che quelle non definite - o utilizzate prima di essere definite - siano state dichiarate tramite la **dichiarazione del prototipo**.

In generale un compilatore è un programma specifico del linguaggio di programmazione e della architettura del calcolatore, in quanto deve conoscere le caratteristiche del linguaggio di programmazione così come quelle del linguaggio macchina utilizzato dal processore, e delle istruzioni di controllo specifiche del sistema operativo.
Esisteranno quindi diversi compilatori per i diversi linguaggi, e per le diverse architetture. Esistono anche diversi compilatori relativi allo stesso linguaggio ed alla stessa architettura che svolgeranno il loro lavoro in modo differente, anche se tutti si devono attenere ad uno standard definito che specifica le caratteristiche del linguaggio di programmazione. Non è affatto ovvio che un programma compilato da due compilatori diversi si comporti sempre esattamente allo stesso modo.
Il compilatore utilizzato fin'ora e nel seguito, si chiama g++, ed è il compilatore più diffuso in ambiente Linux, realizzato e distribuito gratuitamente dalla GNU.

### 5.11.3 Il link

Il link viene eseguito da un programma chiamato linker (in Unix il comando per richiamare il linker è **ld**), che non dipende dal linguaggio di programmazione utilizzato.
In questa fase vengono messi insieme tutti gli **object files** prodotti dalla compilazione e necessari a creare l'eseguibile; viene verificato che tutte le funzioni chiamate siano state definite, che esista una funzione di nome **main**, e che nessuna funzione sia definita più di una volta.
Vengono infine aggiunte le informazioni necessarie al sistema operativo per l'esecuzione del programma, ed il tutto viene messo in un unico file, che è l'eseguibile.
Per proseguire nell'esempio precedente, il comando per realizzare l'eseguibile del nostro programma è:

```
ld -o programma programma.o
```

Il comando dice al linker di analizzare il file (o i files) in coda al comando (in genere object files che hanno estensione **.o** e di generare in output un file eseguibile. L'opzione **-o** serve a specificare il nome da dare al file eseguibile, e deve essere seguita dal nome desiderato (nell'esempio **programma**).
Il linker accetta svariate altre opzioni, alcune delle quali saranno viste in seguito; per le altre si rimanda al manuale relativo.

Il linker è un programma che deve solo raccogliere un insieme di object files e generare un eseguibile in base alla architettura. Per questo non ha legami con il linguaggio di programmazione; in alcuni casi è anche possibile mettere insieme object file prodotti da compilazioni di sorgenti scritti in diversi linguaggi di programmazione.

#### Il g++ ed il linker

Il g++, dopo aver eseguito la compilazione, chiama automaticamente il linker, a meno che non gli venga esplicitamente detto di non farlo tramite l'opzione **-c**.
È quindi possibile produrre l'eseguibile a partire dal sorgente con un unico comando:

```
g++ -o programma programma.cpp
```

In questo caso il g++ compila il sorgente, genera un object file temporaneo, e poi chiama il linker passando ad esso l'object file e le opzioni specifiche del linker (nel caso: **-o programma**). Più in generale, è possibile eseguire il seguente comando:

```
g++ -o programma file1.cpp file2.cpp file3.o file4.o
```

In questo caso il g++ è capace di distinguere quali siano i file da compilare (**file1.cpp** e **file2.cpp**) e li compila, quindi chiama il linker passandogli gli object file compilati e gli object file specificati in linea di comando (**file3.o** e **file4.o**). Alla fine si otterrà il file eseguibile.
Nel caso estremo, in assenza di files sorgenti, il g++ chiama direttamente il linker.
Per motivi che si vedranno in seguito, il comando **ld** non viene quasi mai utilizzato: si chiama il linker tramite il comando **g++** anche se non ci sono file da compilare:

```
g++ -o programma programma.o
```

### 5.11.4 Librerie

Spesso è utile codificare alcune funzioni di carattere generale per poi renderle disponibili a tutti i programmi che ne possano avere bisogno. Vorremo quindi compilare le nostre funzioni una volta per tutte, per poi racchiuderle in un file, che diverrà una sorta di "raccolta" di object files, in modo che qualsiasi programma che utilizzi una parte di queste funzioni possa accedere a questo file e prelevare da esso il codice compilato relativo alle sole funzioni che il programma utilizzerà. Questo file si chiama **libreria**.

#### Realizzazione di una libreria: un esempio

Supponiamo di voler scrivere il codice per poter calcolare numericamente il valore di alcune funzioni matematiche irrazionali. Potremo allora scrivere un file, **trigonometriche.cpp** in cui definiremo alcune funzioni del tipo:

```
double coseno(double x) {
   ... // corpo della funzione coseno
}

double cotangente(double x) {
   ... // corpo della funzione cotangente
}

...
```

per il calcolo del valore delle funzioni trigonometriche per un dato **x**; quindi potremo creare un altro file, che chiamiamo **logaritmiche.cpp** in cui definiremo le funzioni:

```
double logaritmo(double base, double x) {
   ... // corpo della funzione logaritmo
}

double esponenziale(double x) {
```

```
    ... // corpo della funzione esponenziale
}

...
```

per il calcolo delle funzioni logaritmiche, e così via.
Creati i sorgenti, dovremo compilarli per creare gli object file:

```
g++ -c trigonometriche.cpp
g++ -c logaritmiche.cpp
...
```

A questo punto disponiamo degli object file ***.o**. Per creare una libreria si fa utilizzo di un programma che raccoglie gli object file, genera un indice del contenuto, e mette tutto in un file (che deve avere per nome **lib***nome_della_libreria***.a**). Questo programma si chiama *archiver*, ed in Linux si richiama con il comando **ar**:

```
ar r libmatematica.lib trigonometriche.o logaritmiche.o ...
```

Il comando chiama l'archiver e gli dice di raccogliere in un file chiamato **libmatematica.a** gli object file specificati in coda al comando. La libreria è pronta per essere utilizzata.
Va sottolineato che quanto fatto non è trasformabile in un programma, in quanto manca la funzione **main()**.

### Utilizzo di una libreria

Le funzioni di una libreria, come detto, possono essere utilizzate una volta dichiarate nel nostro sorgente. Tuttavia, al momento del **link**, dovremo dire al linker dove cercare la definizione di queste funzioni: infatti nessuna di queste comparirà nel nostro (o nei nostri) object file. Per dare al linker questa informazione, si devono utilizzare le opzioni **-L** e **-l**: la prima, seguita dal nome di una directory, dice al linker dove cercare il file di libreria, la seconda, seguita dal **nome della libreria privato del prefisso lib e del suffisso .a** specifica il nome del file di libreria.

Nota: non si devono inserire spazi tra l'opzione **-l** ed il nome della libreria! Lo spazio è invece opzionale tra l'opzione **-L** ed il nome della directory.

Nota: se si omette l'opzione **-L**, il linker cercherà la libreria in alcune directory predefinite, come **/usr/lib** e **/usr/local/lib**.

Nota: l'opzione **-l** deve essere collocata in coda al comando di compilazione.

Proseguendo nell'esempio del paragrafo precedente, noi potremo scrivere un programma di nome **pitagora.cpp**:

```
#include <iostream>
double coseno(double); // dichiarazione della funzione coseno
double seno(double); // dichiarazione della funzione seno
int main() {
```

```
    double d = coseno(0.5)*coseno(0.5)+seno(0.5)*seno(0.5);
    if (d != 1) cout << "Errore!" << endl;
    return 0;
}
```

Come visto, supponendo che il file **libmatematica.a** si trovi in /home, il comando che dovremo dare per ottenere l'eseguibile sarà:

```
g++ -o pitagora pitagora.cpp -L/home -lmatematica
```

Il compilatore genera l'object file del nostro sorgente, quindi il linker guarda in questo object file quali sono le funzioni usate ma non ancora definite, le cerca nel file di libreria specificato dalle opzioni **-L** e **-l**, ed estrae da queste il codice compilato relativo alle funzioni usate per metterlo assieme al codice compilato del nostro sorgente, quindi crea l'esguibile di nome **pitagora**.

### Ordine di ricerca delle funzioni nelle librerie

Il linker legge gli object file e le librerie sequenzialmente, nell'ordine specificato dalla riga di comando.
Durante la lettura degli object file, tutte le funzioni **definite** vengono copiate nell'eseguibile, e tutte le funzioni utilizzate ma non ancora definite vengono messe in una lista di funzioni indefinite. Quando si incontra la definizione di una funzione presente nella lista, questa viene copiata nell'eseguibile e rimossa dalla lista.
Durante la lettura dei file specificati come **librerie**, per ogni definizione di funzione trovata viene controllata la lista delle indefinite: se la funzione compare nella lista viene copiata nell'eseguibile, altrimenti viene ignorata.
In questo modo avremo un eseguibile in cui sono state incluse **solo le funzioni effettivamente utilizzate**, riducendo così le dimensioni dell'eseguibile, e quindi lo spazio di memoria occupato dall'eseguibile quando viene eseguito. D'altra parte questo meccanismo implica che una funzione di libreria utilizzata in un object file che viene letto solo **dopo** aver già analizzato il file di libreria che la definisce, non sarà caricata nell'eseguibile e risulterà pertanto **indefinita**. Da qui nasce l'esigenza di specificare le funzioni di libreria in coda alla linea di comando.

Nota: i file di libreria possono essere inclusi nella linea di comando come se fossero degli object file:

```
g++ -o pitagora /home/libmatematica.a pitagora.cpp
```

In questo caso vengono trattati come gli object file: tutte le funzioni definite vengono caricate nell'eseguibile. Il vantaggio è che non si hanno restrizioni nell'ordine in cui le librerie vanno specificate nella linea di comando, lo svantaggio è che l'eseguible risulterà più grosso.

### Librerie di sistema

Esistono diverse librerie standard che vengono rese disponibili assieme al compilatore. In particolare le **libc.a**, **libg++.a**, **libstdc++.a** sono le librerie che contengono la maggior parte delle definizioni che abbiamo visto o che vedremo in seguito, quali ad esempio (**cout**, **cin**, **fstream**, **string**, etc.), o la **libm.a** che contiene le definizioni delle funzioni matematiche (**sqrt**, **pow**, etc.).

Quando scriviamo un programma in un dato linguaggio, alcune di queste librerie devono essere caricate anche se non si utilizzano direttamente le funzioni ivi definite, in quanto è il compilatore stesso che ne fa un uso implicito.
Il comando **g++**, quando chiama il linker, passa in modo automatico al linker le opzioni per caricare tutte queste librerie. Questo è il motivo per cui il comando **ld** non viene utilizzato direttamente: se si vuole chiamare il linker tramite **ld**, bisogna includere nelle librerie da utilizzare anche quelle specifiche del compilatore, e spesso non è banale sapere quale sia l'elenco completo.

### 5.11.5 Include files

Come abbiamo visto, l'utilizzo di funzioni di libreria richiede che nel sorgente del programma vengano dichiarate le funzioni utilizzate.
Accade spesso che una libreria contenga la definizione di decine di funzioni, ed ogni volta che ne si utilizza una si deve andare a vedere se il prototipo è già stato dichiarato nel sorgente.
Per ovviare a questo problema, usualmente una libreria è accompagnata da un **include file**, spesso con un nome che ha per estensione **.h** o **.hh**, che contiene la dichiarazione di tutte le funzioni definite nella libreria.
Il programmatore non deve far altro che includere questo file nel suo sorgente tramite la direttiva **#include**, per avere la certezza che qualsiasi funzione di questa libreria si utilizzi nel seguito del sorgente, tale funzione è stata correttamente dichiarata.
Nell'esempio visto, potremo creare un file, di nome **matematica.h** del tipo:

```
double seno(double);
double coseno(double);
double logaritmo(double, double);
...
```

L'utilizzatore della libreria, potrà sostituire nel suo sorgente le esplicite dichiarazioni delle funzioni utilizzate, con la singola istruzione:

```
#include "matematica.h"
```

All'atto della compilazione, dovremo dire al compilatore dove cercare gli include file inclusi in questo modo. Questo si fa utilizzando l'opzione **-I** seguita dal nome della directory in cui sono messi gli include file. Se ad esempio l'include file matematica.h si trova nella directory /home/inclide, potremo compilare il nostro sorgente con il comando:

```
g++ -I/home/include -o pitagora pitagora.cpp -L/home -lmatematica
```

#### Include file di sistema

Se l'istruzione **#include** specifica il nome del file racchiuso tra `<>`, il compilatore cercherà gli include file in una serie di directory predefinite (**/usr/include**, **/usr/local/include**, etc.). Queste directory contengono gli include file delle librerie del compilatore, e delle librerie di sistema, pertanto quando si utilizzano tali librerie, gli include file vanno specificati in questo modo:

```
#include <iostream>
#include <string>
#include <fstream>
...
```

Così facendo non saremo costretti ad utilizzare l'opzione **-I** per questi file.

### Inclusioni multiple

Spesso capita che un include file venga incluso più volte nello stesso sorgente, ad esempio se lo includiamo direttamente, e se includiamo un altro file che a nostra insaputa lo include a sua volta.
Questo può portare ad errori di compilazione, ad esempio se il file in questione contiene non solo dichiarazioni ma anche definizioni di classi, variabili globali, etc.
Per non incorrere in questo problema generalmente viene utilizzata una direttiva che impedisce l'esecuzione delle righe di codice dell'include file più di una volta:

```
#ifndef MATEMATICA_H
#define MATEMATICA_H 1
   ... // corpo dell'include file
#endif
```

Le direttive **#ifndef MATEMATICA_H** e **#endif** dicono al compilatore di leggere le righe di codice comprese tra le due istruzioni **se la variabile MATEMATICA_H non è già stata definita** (il comportamento è simile alle istruzioni c++ **if endif**). La prima istruzione dopo **#ifndef** definisce la suddetta variabile (ad un valore qualunque), tramite la direttiva **#define**. In questo modo, se nel nostro sorgente accidentalmente viene nuovamente incluso lo stesso file, la variabile risulterà definita ed il compilatore ometterà di rileggere per la seconda volta le istruzioni dell'include file.
Va osservato che è importante che il nome della variabile sia scelto in modo univoco. Generalmente un include file utilizza una variabile costituita dal nome del file stesso, sostituendo il **.** con un **_** per non incorrere in un errore sintatico della direttiva **#ifndef**. Nella maggior parte dei casi questa variabile garantisce l'univocità della scelta.

## 5.12 Strutture e classi

### 5.12.1 Definizione di una struttura

Abbiamo visto come sia possibile definire aggregato di elementi dello stesso tipo tramite le variabili di tipo **array**.
Il C++ permette di definire anche aggregati di variabili di tipo diverso, tramite la definizione di strutture (**struct**).
Ad esempio, se vogliamo scrivere un programma che gestisca un database degli atomi , potremo definire la seguente struttura:

```
struct Atomo {
  int neutroni;
  int protoni;
  double pesoAtomico;
};
```

La struttura viene definita tramite la *keyword* **struct**, seguita dal nome che vogliamo dare alla struttura, e quindi in un blocco l'elenco delle variabili - con il loro tipo - che costituiscono l'aggregato dei dati di questa struttura. Le variabili di cui è costituita una struttura si chamano **membri**. Il blocco va terminato con un **;**.
Come per gli array, anche per le strutture gli elementi vengono allocati in memoria sequenzialmente nell'ordine di definizione, ma non è possibile accedere i singoli elementi utilizzando i simboli

.

### 5.12.2 Utilizzo di una struttura

La definizione di una struttura in C++ è a tutti gli effetti la **definizione di un nuovo tipo di variable**, e può essere utilizzata per definire variabili di quel tipo:

```
Atomo a;
```

In questo modo ho definito una variabile **a** di tipo *struttura Atomo*. L'accesso in letture e scrittura ai membri di una struttura si attua tramite l'operatore `"."`:

```
a.neutroni = 8;
a.protoni = 8;
a.pesoAtomico = 15.9994;
```

In questo modo abbiamo assegnato un valore a tutti i membri della variabile **a**.
Ogni membro della struttura viene considerato una variabile del tipo definito nella definizione della struttura stessa: nell'esempio, con il nome **a.pesoAtomico** indicheremo una variabile di tipo **double**, con il nome **a.neutroni** una variabile di tipo **int**. I membri vanno utilizzati esattamente come le variabili del tipo corrispondente:

```
int p = a.protoni;
int neutroni = a.neutroni;
```

Nota: si osservi che il nome **neutroni** ed il nome **a.neutroni** sono diversi, e non c'e' ambiguità perché indicano due variabili differenti.
Una struttura può essere inizializzata con la stessa sintassi utilizzata per gli array:

```
Atomo trizio = { 2, 1, 3.0 };
```

Ad una struttura può essere assegnata una struttura dello stesso tipo:

```
Atomo a = trizio;
```

L'assegnazione comporta l'assegnazione **elemento per elemento** dei componenti della struttura.
Nota: poiché la struttura non è un tipo nativo, non sono definiti i comportamenti relativi al confronto (<, >, ==, !=), nè è definito il modo di fare I/O, cioè non si possono utilizzare gli operatori << e >>. Queste operazioni vanno gestite elemento per elemento.

### 5.12.3 Puntatore a struttura e allocazione dinamica di una struttura

Come per le variabili native, anche per le strutture possono essere definiti dei puntatori:

```
Atomo * ap = NULL;
ap = \&a; // assegno ad ap l'indirizzo di a
```

Quando si utilizza un puntatore, esistono due modi per riferirsi agli elementi della struttura puntata: tramite dereferenziazione del puntatore

```
cout << (*ap).pesoAtomico << endl;
```

o, più frequentemente, tramite l'operatore ->, specifico per questo scopo.

```
cout << ap->pesoAtomico << endl;
```

L'allocazione dinamica di una struttura si opera in modo equivalente alla allocazione dinamica di un qualsiasi altro tipo di variabile:

```
Atomo * ap = new Atomo;
```

### 5.12.4 Array di strutture

Come conseguenza del fatto che la struttura non è altro che un *tipo* di variabile, è possibile definire array di strutture:

```
Atomo idrogeni[3];
a[2] = trizio;
```

In questo esempio, **idrogeni** è un array i cui elementi sono variabili di tipo **Atomo**, cioè strutture.

### 5.12.5 Strutture come membri di altre strutture

È possibile definire una strutt ura come elemento di un'altra struttura:

```
struct Molecola {
  Atomo componenti[10];
  int nAtomi[10]
  double massa;
};
```

In questo esempio abbiamo definito una struttura (**Molecola**) i cui membri sono un **array di strutture Atomo**, un **array di int** ed una **double**.
Potremo ora scrivere:

```
Atomo idrogeno = { 0, 1, 1.0079 };
Atomo ossigeno = { 8, 8, 15.9994 };
Molecola acqua;
acqua.componenti[0] = idrogeno;
acqua.nAtomi[0] = 2;
acqua.componenti[1] = ossigeno;
acqua.nAtomi[1] = 1;
acqua.massa = 18.0;

cout << acqua.componenti[0].neutroni << endl;
...
```

Come si può vedere, il nome **acqua.componenti[0]** è il primo elemento dell'array di variabili di tipo **Atomo**, ed il suo membro **neutroni** andrà indicato aggiungendo **.neutroni** al nome della variabile, quindi con **acqua.componenti[0].neutroni**, secondo una struttura gerarchica di nomi.

### 5.12.6 Struttura come argomento di una funzione

Una struttura può essere passata ad una funzione come argomento.
Va tuttavia considerato che il passaggio di una struttura come argomento *by value* comporta la creazione di una struttura nello spazio di memoria utilizzato dalla funzione, e la copiatura della struttura passata come argomento nella nuova struttura. Talvolta le strutture sono di grandi dimensioni, e la copiatura risulta inefficiente.
Per questo motivo usualmente le strutture vengono passate *by pointerøby reference*, anche se non c'e' la necessità di modificarle all'interno della funzione.

### 5.12.7 Osservazione sulle strutture

Storicamente il C++ è nato come una estensione del linguaggio C di tali proporzioni da trasformarlo in un linguaggio che col C ha in comune solo alcune regole sintattiche di base.
Le strutture esistevano già in C, e nei paragrafi precedenti sono state presentate secondo la definizione del C. Vedremo alla fine dei prossimi paragrafi come anche il concetto di struttura sia stato esteso di significato.

### 5.12.8 Classi

L'idea che sta alla base del concetto di classe è di fornire al programmatore uno strumento per definire nuovi tipi di variabile che possano essere trattate come le variabili native.
Come abbiamo visto nei paragrafi precedenti le strutture (del C) sono una parziale realizzazione di tale obiettivo.
Possiamo vedere una classe come una estensione del concetto di struttura già visto, pertanto quanto detto per le strutture resta valido per le classi.
In aggiunta ai concetti già visti, una classe è capace di avere come membri delle funzioni, di controllare l'accesso ai membri della classe stessa, di avere definita una inizializzazione ed una procedura di distruzione, e molte altre cose che non saranno coperte da questo corso.
Spesso in manuali sulla programmazione in C++ si fa riferimento agli **oggetti**. Mentre la classe è la definizione di un tipo di variabile, di come è costituita e di come si può utilizzare, ogni variabile di questo tipo definita nel programma si chiama **oggetto**.

### 5.12.9 Funzioni membro

Le **classi** sono strutture di dati che possono avere come membri, oltre a variabili di qualsiasi tipo, anche delle funzioni, dette **metodi della classe**.
Supponiamo di voler scrivere una classe che rappresenti i vettori tridimensionali della geometria euclidea:

```
class Vettore {
  double x, y, z;
};
```

In questo modo abbiamo definito un nuovo tipo, il **Vettore**, i cui membri sono le componenti cartesiane del vettore (in modo analogo a come avremmo fatto con una struttura).
Supponiamo ora di voler scrivere una funzione che ne calcoli il modulo.
Programmando con le classi si può inglobare tale funzione nella definizione della classe, in modo che diventi parte integrante della classe stessa, definendo come membro della classe una funzione che esegua questa operazione:

```
class Vettore {
  double x, y, z;
  double Modulo();
};

double Vettore::Modulo() {
  return sqtr(x*x+y*y+z*z);
}
```

La funzione **Modulo()** membro della classe **Vettore** si potrà richiamare nello stesso modo in cui si fa riferimento agli altri membri della classe stessa:

```
main() {
  Vettore v;
  v.x = 3.9;
  v.y = v.z = 0.0;
  double mod = v .Modulo();
}
```

Ora facciamo alcune osservazioni.
La funzione **Vettore::Modulo()** è membro della classe **Vettore**, e questo fatto va evidenziato: per fare ciò, nella sua definizione si deve anteporre al nome della funzione il nome della classe a cui questa funzione appartiene, seguita da **::** (in questo caso **Vettore::**).
La funzione **Vettore::Modulo()** non ha bisogno di ricevere un argomento specificante il **Vettore** di cui calcolare il modulo: questa funzione calcola il modulo dell'oggetto di cui lei stessa è membro. Inoltre, essendo membro della classe **Vettore**, ha accesso agli altri membri della classe **x**, **y** e **z** direttamente attraverso il loro nome, senza dovervi accedere tramite il nome dell'oggetto seguito dall'operatore **.**

### 5.12.10 Controllo di accesso ai membri della classe

Il programmatore ha la possibilità di decidere se un membro della classe possa o meno essere utilizzato all'esterno della classe. Per fare ciò esistono due *keywords*, **public** e **private** che, utilizzate all'interno della definizione della classe, determinano se sia lecito o meno accedere ad un membro della classe. Vediamo un esempio:

```
class Vettore \{
public:
  double GetX();
private:
  double x, y, z;
};

double Vettore::GetX() {
  return x;
}
```

In questo esempio il metodo **GetX()** è pubblico, e può essere utilizzato da chi utilizza la classe per ottenere la componente x del vettore, mentre il valore diretto **x** è privato, e solo i metodi della classe possono utilizzarlo:

```
...
double ascissa = v.GetX(); // lecito
double ascissa2 = v.x; // non lecito: si avra'
                       // un errore in compilazione
...
```

In questo modo è possibile fornire una serie di regole per accedere alle funzionalità della classe (**l'interfaccia della classe**), ad esempio creando un metodo opportuno per ottenere la componente x del vettore, proteggendo i dettagli sulla implementazione della stessa, cioè il fatto che si utilizzano le componenti cartesiane per trattare il vettore internamente alla classe. Supponiamo che ad un certo punto dello sviluppo si ritenga più opportuno utilizzare per l'implementazione del vettore le coordinate polari:

```
class Vettore {
public:
  double GetX();
private:
```

```
  double r, tetha, phi;
};

double Vettore::GetX() {
  return r*cos(theta)*cos(phi);
}
```

Se noi avessimo concesso agli utilizzatori della classe l'accesso al membro **x**, questa modifica avrebbe costretto a riprogrammare tutto il loro codice. Attraverso il controllo sull'accesso, che costringe ad utilizzare il metodo **GetX()**, la modifica dell'implementazione non comporta alcuna riscrittura della parte di codice che usa la classe, in quanto la **GetX()** si comporta - dal punto di vista dell'utilizzatore - sempre nello stesso modo, anche se fa il calcolo in modo diverso.
Per default i membri di una classe sono privati. È comunque buona norma di programmazione specificarene esplicitamente il carattere pubblico o privato.
Le *keyword* **public** e **private** possono comparire più volte all'interno della definizione della classe. Tutte le definizioni di una classe hanno il carattere specificato dal'ultima *keyword* inserita prima di esse.

### 5.12.11 I costruttori

Il costruttore di una classe è un metodo che viene eseguito ogni volta che si istanzia un oggetto (cioè quando si definisce staticamente o dinamicamente una variabile del tipo definito da una classe).
Il costruttore è un metodo che ha lo stesso nome della classe, ed è **privo di tipo**, in quanto non ha valore di ritorno perché non è un metodo chiamato in modo attivo dal programmatore.
Proseguendo nell'esempio, vogliamo che alla definizione di un oggetto **Vettore**, questo abbia componenti (0, 0, 0). Per fare ciò dovremo definire il costruttore in modo opportuno:

```
classe Vettore {
public:
  Vettore();
  ...
};

Vettore::Vettore() {
  x = y = z = 0.0;
};
```

Come con tutte le funzioni, anche con i costruttori possiamo utilizzare la tecnica dell'*overload* per definire più modi di inizializzare un oggetto.
Potremo quindi definire più costruttori, differenziati da numero e tipo di argomenti, a seconda di come vogliamo poter inizializzare un oggetto:

```
classe Vettore {
public:
  Vettore();
  Vettore(double xx, double yy, double zz);
  Vettore(double *v);
  ...
```

```
};

Vettore::Vettore() {
  x = y = z = 0.0;
};

Vettore::Vettore(double xx, double yy, double zz) {
  x = xx;
  y = yy;
  z = zz;
};

Vettore::Vettore(double *v) {
  x = v[0];
  y = v[1];
  z = v[2];
};
```

La nostra classe ci permette adesso di utilizzare diversi modi per inizializzare un oggetto **Vettore**; i diversi costruttori si attivano inserendo gli argomenti tra parentesi dopo il nome della variabile che vogliamo definire:

```
int main() {
  Vettore v1(1.2, 3.3, 0.4);
  double v[3] = {1.2, 2.2, 3.2};
  Vettore v2(v);
  Vettore * v3 = new Vettore(v);
  ...
```

In questo esempio vediamo la definizione di oggetti **Vettore** inizializzati prima tramite il costruttore **Vettore::Vettore(double xx, double yy, double zz)**, poi tramite il costruttore **Vettore::Vettore(double *v)**, quindi un oggetto **Vettore** allocato dinamicamente ed inizializzato ancora con il costruttore **Vettore::Vettore(double *v)**.
Va osservato che, come per le altre funzioni, se si desidera inizializzare un oggetto con determinati parametri, la funzione atta a fare questa operazione, cioè il costruttore opportuno, deve essere definita, altrimenti avremo un errore di *undefined object* in compilazione.

### Il default constructor

Il **default constructor** è un costruttore - definito dal compilatore - che viene chimato quando si definisce un oggetto senza usare argomenti, qualora non sia stato dichiarato esplicitamente il costruttore opportuno (cioè quello con prototipo **Classe::Classe();**).
Il default constructor si limita ad inizializzare tutte le variabili argomento tramite la loro inizializzazione di default, vale a dire che per tutti gli elementi di tipo non nativo (cioè classi) viene chiamato il loro default constructor. Si deve però fare attenzione al fatto che nelle classi il default constructor **non inizializza le variabili di tipo nativo**, che quindi risulteranno indefinite.
L'esistenza del default constructor ci permette di scrivere classi senza dover necessariamente scrivere il codice per un costruttore che non ha bisogno di fare nulla.

### Il copy constructor

Per default un oggetto può essere inizializzato con un oggetto dello stesso tipo, anche se non è definito appositamente un costruttore. L'inizializzazione di default avviene tramite copia di tutti i membri dell'oggetto inizializzante.

```
Vettore v1(1, 2, 3);
Vettore v2(v1); // v2 avra' componenti (1, 2, 3)
```

Va però osservato che a volte questo non è quello che si desidera. Prendiamo ad esempio una classe per la manipolazione degli array (si può pensare ad una classe del genere per poter inserire un controllo di accesso ad elementi in modo da non eccedere i limiti della memoria allocata):

```
class Array {
public:
  Array(double * v, int size);
  double * arr;
  int dim;
};

Array::Array(double * v, int size) {
  arr = new double[size];
  for(int i=0; i<size; ++i) arr[i] = v[i];
  dim = size;
}
```

Se eseguiamo un codice del tipo:

```
int main() {
  double v[5] = {1, 2, 3, 4, 5};
  Array a1(v, 5); // definisco a1 con gli elementi di v
  Array a2(a1);  // definisco a2 tramite il
                 // default copy constructor
  ...
```

otterremo che **a2.arr** punterà alla stessa allocazione di **a1.arr**, quindi una modifica agli elementi di **a2.arr** corrisponderà ad una modifica degli elementi di **a1.arr**.
Quello che si vuole invece ottenere è che venga allocata una zona di memoria equivalente, e che le due zone di memoria contengano gli stessi valori.
Per evitare di utilizzare il copy constructor di default, si deve definirlo esplicitamente. Il prototipo del copy constructor è (in generale) **Classe::Classe(Classe &);**:

```
class Array {
public:
  Array(double * v, int size);
  Array(Array \& a); // copy constructor
  ...
};
Array::Array(Array & a) {
  dim = a.dim;
```

```
  arr = new double[dim];
  for(int i=0; i<dim; ++i) arr[i] = a.arr[i];
}
```

In questo modo otterremo il risultato voluto. Alla istruzione

```
Array a2(a1);
```

verrà eseguito il copy constructor, verrà allocata la memoria per **a2.arr** e verrà effettuata copia del contenuto.

### 5.12.12 Il distruttore

Quando un oggetto di una classe termina il suo scopo (quando si esce dal blocco delle istruzioni in cui è definito o quando si eseguie il **delete** di un oggetto allocato dinamicamente), viene chiamato un metodo detto distruttore (**destructor**).
Il distruttore è un metodo che si chiama come il default constructor, ma con una ~ che precede il suo nome.
Se questo metodo non viene definito alla definizione della classe, viene eseguito un distruttore di default, che libera la memoria di tutti i membri che sono stati allocati all'atto della costruzione dell'oggetto.
Tuttavia, se l'oggetto contiene tra i suoi membri puntatori a cui è stata assegnata una allocazione di memoria dinamica, questa memoria non viene rilasciata (viene rilasciata solo la memoria occupata dal puntatore). In questi casi, per evitare problemi di *memory leak* in esecuzione, si deve definire appositamente il distruttore per rilasciare correttamente la memoria.
Per riprendere l'esempio precedente, un distruttore opportuno sarà fatto come segue:

```
class Array {
  ...
  ~Array();
  ...
};
Array::~Array() {
  delete[] arr;
}
```

### 5.12.13 Overload degli operatori

Come detto, lo scopo delle classi è di definire un tipo di variabile che sia utilizzabile come un tipo nativo. Una delle cose che si possono desiderare è utilizzare tutti o alcuni degli operatori +, -, * etc. tra gli oggetti di una classe. Ad esempo, si potrebbe voler confrontare due oggetti **Vettore** tramite il simbolo ==. Questo è possibile utilizzando la tecnica dell'**overload degli operatori**.
Vediamo l'esempio:

```
class Vettore {
  ...
  bool operator==(Vettore & v);
  ...
```

```
};
bool Vettore::operator==(Vettore & v) {
  return (GetX() == v.GetX()) && (GetY() == v.GetY()) && (GetZ() == v.GetZ());
}
```

In questo modo abbiamo definito l'operatore $==$ che opera tra due oggetti di tipo **Vettore**, che ritorna una variabile booleana vera o falsa a seconda che le componenti coincidano oppure no (vale a dire il comportamente che ci aspetteremmo). A questo punto sarà lecita l'istruzione:

```
int main() {
  Vettore v1, v2(1.2, 2.3, 3.4);
  if (v1 == v2) {
    ...
  }
```

### 5.12.14 Strutture e classi

Da quanto si è visto, in C++ la classe è una estensione del concetto di struttura del linguaggio C. Il C++ ha esteso queste funzionalità anche alle strutture, che in sostanza hanno come unica differenza dalle classi il fatto di avere per default i membri pubblici.
Tuttavia, è convenzione di programmazione molto diffusa quella di utilizzare le strutture ogni qualvolta si ha bisogno di un semplice aggregato di dati, mentre si utilizzano le classi quando si vuole definire un tipo di variabile che abbia in se non solo la definizione dei dati di cui è costituito, ma anche controllo di accesso dei membri, funzionalità, operatori etc.

## 5.13 Libreria matematica

La libreria standard del C++ include molte funzioni definite in `cmath` e `cstlib`. La tabella riassume le più usate:

| Funzione | Libreria | Operazione |
|---|---|---|
| abs(x) | cstdlib | valore assoluto di x (x intero) |
| acos(x) | cmath | arco il cui coseno di è x (arco-coseno) |
| asin(x) | cmath | arco il cui seno è x (arco-seno) |
| atan(x) | cmath | arco la cui tangente è x (arco-tangente) |
| atan2(y,x) | cmath | arco la cui tangente è y/x (arco-tangente) |
| ceil(x) | cmath | ritorna il minor intero maggiore o uguale a x |
| cos(x) | cmath | coseno di x |
| cosh(x) | cmath | coseno iperbolico di x |
| exp(x) | cmath | $e^x$ |
| fabs(x) | cmath | valore assoluto di x (x floating-point) |
| log(x) | cmath | logaritmo naturale di x |
| log10(x) | cmath | logaritmo in base 10 di x |
| pow(x,y) | cmath | $x^y$ |
| random() | cstdlib | ritorna un intero random tra 0 e RAND_MAX |
| sin(x) | cmath | seno di x |
| sinh(x) | cmath | seno iperbolico di x |
| sqrt(x) | cmath | radice di x |
| tan(x) | cmath | tangente di x |
| tanh(x) | cmath | tangente iperbolica di x |

## 5.14 Stringhe

Una stringa è una sequenza di caratteri alfanumerici che rappresentano parole, frasi, o dati numerici tramite la rappresentazione dei numeri come caratteri.
Il C++ utilizza una classe definita appositamente per trattare le stringhe, differenziandosi fortemente dalla gestione delle stringhe operata in C; tuttavia esiste un grande numero di librerie di sistema che fanno uso di stringhe stile C, per cui non si può prescindere da una introduzione di queste ultime prima di passare alla trattazione delle stringhe in C++.

### 5.14.1 I caratteri alfanumerici: le variabili char

Esiste una codifica standard che associa ad un valore numerico la rappresentazione di un particolare carattere: la codifica ASCII (Tab. 5.2). In base a questa codifica, ad esempio, al numero 65 (0x41) è associato il carattere 'A', al numero 97 (0x61) è associato il carattere 'a', etc.
La codifica ASCII, nella sua versione non estesa, associa i numeri da 0 a 127 a tutti i caratteri alfanumerici, caratteri di punteggiatura, parentesi e alcuni caratteri di controllo di terminale (come il *Carriage Return*, il *Line Feed*, etc.). Nella sua versione estesa, assegna i numeri da 128 a 255 ad altri caratteri tra i quali quelli specifici di alcuni alfabeti particolari (i caratteri accentati dell'alfabeto italiano o francese, i caratteri degli alfabeti danese, norvegese, etc.).
Per descrivere i 256 caratteri della codifica ASCII è possibile utilizzare una variabile di tipo **unsigned char**, che può appunto assumere valori tra 0 e 255.
Per assegnare un carattere ad una variabile di tipo char si deve specificare il carattere tra singoli

apici; è anche possibile assgnare alla variabile il corrispondente valore numerico, anche se questa pratica è raramente utile:

```
unsigned char c1 = 's';
unsigned char c2 = 115; // Assegnazione equivalente
cout << c1 << endl << c2 << endl;
```

Il valore alfanumerico di una variabile unsigned char si può visualizzare utilizzando l'operatore di I/O <<; nell'esempio verrà visualizzato sul terminale due volte il carattere **s**.
Nota: la variabile naturale per la rappresentazione dei caratteri è **unsigned char**, ma è diffuso l'utilizzo della variabile **char** allo stesso scopo. Benché formalmente la variabile **char** assuma valori compresi tra -128 e 127, l'assegnazione dei caratteri non è ambigua:

```
char c1 = 'a';   // OK
char c2 = 0x64;  // OK: assegnazione equivalente
char c3 = 210;   // formalmente incorretta, ma accettata
                 // dal compilatore
```

Nel seguito parleremo indifferentemente di **char** o **unsigned char**.

### 5.14.2 String literal

Analogamente alle costanti numeriche, è possibile definire una *costante stringa*, che prende il nome di **string literal**.
Una string literal viene definita specificando i caratteri della stringa tra doppi apici:

```
"Ciao, mondo."
```

Le string literal, come le costanti numeriche, sono generalmente utilizzate per inizializzare variabili di tipo opportuno, come vedremo in seguito.

### 5.14.3 C-style string

La stringa di tipo C è un array di char in cui ogni elemento dell'array assume il valore (secondo la codifica ASCII) del carattere corrispondente della stringa. Alla fine della stringa deve essere aggiunto un elemento di valore **0** come terminatore. La presenza del terminatore elimina la necessità di dover specificare la lunghezza della stringa. Tutte le funzioni di libreria che fanno utilizzo di stringhe di tipo C si basano sulla presenza del terminatore per poter trattare correttamente le stringhe.
La definizione di una stringa è banalmente la definizione di un array di char della dimensione sufficiente a contenere i caratteri della stringa (più uno per il terminatore).
Una stringa può essere inizializzata utilizzando una string literal:

```
char buffer[13] = "Ciao, mondo.";
```

Questa inizializzazione viene eseguita dal compilatore aggiungendo automaticamente il terminatore.
È possibile inizializzare una stringa assegnando il valore opportuno a ciascuno dei suoi elementi:

| Dec | Oct | Hex | Char | Comments | Dec | Oct | Hex | Char | Comments |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 000 | 00 | | NUL (Null) | 64 | 100 | 40 | @ | |
| 1 | 001 | 01 | | SOH (Start of heading) | 65 | 101 | 41 | A | |
| 2 | 002 | 02 | | STX (Start of text) | 66 | 102 | 42 | B | |
| 3 | 003 | 03 | | ETX (End of text) | 67 | 103 | 43 | C | |
| 4 | 004 | 04 | | EOT (End of transmission) | 68 | 104 | 44 | D | |
| 5 | 005 | 05 | | ENQ (Enquiry) | 69 | 105 | 45 | E | |
| 6 | 006 | 06 | | ACK (Acknowledge) | 70 | 106 | 46 | F | |
| 7 | 007 | 07 | 'a' | BEL (Bell) | 71 | 107 | 47 | G | |
| 8 | 010 | 08 | 'b' | BS (Backspace) | 72 | 110 | 48 | H | |
| 9 | 011 | 09 | 't' | HT (Horizontal tab) | 73 | 111 | 49 | I | |
| 10 | 012 | 0A | 'n' | LF (Linefeed) | 74 | 112 | 4A | J | |
| 11 | 013 | 0B | 'v' | VT (Vertical tab) | 75 | 113 | 4B | K | |
| 12 | 014 | 0C | 'f' | FF (Form feed) | 76 | 114 | 4C | L | |
| 13 | 015 | 0D | 'r' | CR (Carriage return) | 77 | 115 | 4D | M | |
| 14 | 016 | 0E | | SO (Shift out) | 78 | 116 | 4E | N | |
| 15 | 017 | 0F | | SI (Shift in) | 79 | 117 | 4F | O | |
| 16 | 020 | 10 | | DLE (Data link escape) | 80 | 120 | 50 | P | |
| 17 | 021 | 11 | | DC1 (Device Control, X-ON) | 81 | 121 | 51 | Q | |
| 18 | 022 | 12 | | DC2 (Device Control) | 82 | 122 | 52 | R | |
| 19 | 023 | 13 | | DC3 (Device Control, X-OFF) | 83 | 123 | 53 | S | |
| 20 | 024 | 14 | | DC4 (Device Control) | 84 | 124 | 54 | T | |
| 21 | 025 | 15 | | NAK (Negative Acknowledge) | 85 | 125 | 55 | U | |
| 22 | 026 | 16 | | SYN (Synchronous idle) | 86 | 126 | 56 | V | |
| 23 | 027 | 17 | | ETB (End transmission blocks) | 87 | 127 | 57 | W | |
| 24 | 030 | 18 | | CAN (Cancel) | 88 | 130 | 58 | X | |
| 25 | 031 | 19 | | EM (End of medium) | 89 | 131 | 59 | Y | |
| 26 | 032 | 1A | | SUB (substitute) | 90 | 132 | 5A | Z | |
| 27 | 033 | 1B | | ESC (Escape) | 91 | 133 | 5B | [ | |
| 28 | 034 | 1C | | FS (File separator) | 92 | 134 | 5C | | |
| 29 | 035 | 1D | | GS (Group separator) | 93 | 135 | 5D | ] | |
| 30 | 036 | 1E | | RS (Record separator) | 94 | 136 | 5E | ^ | |
| 31 | 037 | 1F | | US (Unit separator) | 95 | 137 | 5F | _ | |
| 32 | 040 | 20 | | Space | 96 | 140 | 60 | ` | |
| 33 | 041 | 21 | ! | | 97 | 141 | 61 | a | |
| 34 | 042 | 22 | " | | 98 | 142 | 62 | b | |
| 35 | 043 | 23 | # | | 99 | 143 | 63 | c | |
| 36 | 044 | 24 | $ | | 100 | 144 | 64 | d | |
| 37 | 045 | 25 | % | | 101 | 145 | 65 | e | |
| 38 | 046 | 26 | & | | 102 | 146 | 66 | f | |
| 39 | 047 | 27 | ' | | 103 | 147 | 67 | g | |
| 40 | 050 | 28 | ( | | 104 | 150 | 68 | h | |
| 41 | 051 | 29 | ) | | 105 | 151 | 69 | i | |
| 42 | 052 | 2A | * | | 106 | 152 | 6A | j | |
| 43 | 053 | 2B | + | | 107 | 153 | 6B | k | |
| 44 | 054 | 2C | , | | 108 | 154 | 6C | l | |
| 45 | 055 | 2D | - | | 109 | 155 | 6D | m | |
| 46 | 056 | 2E | . | | 110 | 156 | 6E | n | |
| 47 | 057 | 2F | / | | 111 | 157 | 6F | o | |
| 48 | 060 | 30 | 0 | | 112 | 160 | 70 | p | |
| 49 | 061 | 31 | 1 | | 113 | 161 | 71 | q | |
| 50 | 062 | 32 | 2 | | 114 | 162 | 72 | r | |
| 51 | 063 | 33 | 3 | | 115 | 163 | 73 | s | |
| 52 | 064 | 34 | 4 | | 116 | 164 | 74 | t | |
| 53 | 065 | 35 | 5 | | 117 | 165 | 75 | u | |
| 54 | 066 | 36 | 6 | | 118 | 166 | 76 | v | |
| 55 | 067 | 37 | 7 | | 119 | 167 | 77 | w | |
| 56 | 070 | 38 | 8 | | 120 | 170 | 78 | x | |
| 57 | 071 | 39 | 9 | | 121 | 171 | 79 | y | |
| 58 | 072 | 3A | : | | 122 | 172 | 7A | z | |
| 59 | 073 | 3B | ; | | 123 | 173 | 7B | { | |
| 60 | 074 | 3C | ¡ | | 124 | 174 | 7C | — | |
| 61 | 075 | 3D | = | | 125 | 175 | 7D | } | |
| 62 | 076 | 3E | ¿ | | 126 | 176 | 7E | ~ | |
| 63 | 077 | 3F | ? | | 127 | 177 | 7F | | DEL |

Tabella 5.2: Codice ASCII.

```
char buff[10];
buff[0] = 'O';
buff[1] = 'k';
buff[2] = 0; // aggiungo il terminatore;
             // la stringa vale "Ok"
```

L'assegnamento elemento per elemento ci obbliga ad assegnare anche il valore del terminatore. Va osservato che l'array allocato può essere più grosso di quanto serva, in quanto è il terminatore che stabilisce la fine della stringa. Chiaramente se allochiamo meno spazio del necessario, andremo a scrivere oltre i limiti di allocazione dell'array, generalmente provocando un errore in esecuzione.
Le **C-style string** sono manipolabili tramite svariate funzioni di libreria che ne rendono piuttosto complicato l'utilizzo. Operazioni come la ricerca di caratteri, di sottostringhe, etc., spesso devono essere fatte a mano. Per questo motivo in C++ si gestiscono le stringhe tramite una classe opportuna, descritta nel prossimo paragrafo.

### 5.14.4 La classe string

In C++ esiste una classe, **string**, che ha lo scopo di utilizzare e manipolare le stringhe di caratteri.

#### Definizione di una string

Per utilizzare variabili di tipo **string** si deve includere il file opportuno, che contiene le dichiarazioni necessarie:

```
#include <string>
```

La definizione di una string si opera in modo analogo alla definizione delle variabili di altri tipi:

```
string s;
```

#### Assegnazione ad una variabile string

È possibile assegnare un valore alle variabili di tipo string utilizzando una **string literal**, o un'altra variabile di tipo string:

```
string s1, s2; \\
s1 = "Ciao, mondo."  // assegnazione tramite string literal \\
s2 = s1; // assegnazione tramite altra string \\
```

#### Inizializzazione di una string

Le istruzioni di assegnazione possono essere utilizzate per inizializzare il valore di una string all'atto della sua definizione. In aggiunta a queste, è possibile inizializzare una stringa, chiamando l'opportuno costruttore, utilizzando una string literal, una string, o un char * (C-style string):

```
string s1 = "Ciao, mondo.";
string s2 = s1;
string s3("Ciao, mondo.");
```

```
string s4(s2);
char buffer[80] = "C-style string";
string s5(buffer); // inizializzazione tramite char *
```

Se viene definita una variabile di tipo string senza inizializzazione, questa viene automaticamente inizializzata come stringa nulla ("").

### Input/Output da/verso stream di variabili string

Come per i tipi nativi, anche la **string** può essere scritta su output stream o letta da input stream tramite gli operatori usuali:

```
string s = "Ciao, mondo";
cout << s << endl;
cin >> s;
```

L'ultima istruzione legge da terminale una sequenza di caratteri e li assegna alla variabile **s**. Va osservato che **cin** utilizza come separatore lo spazio o il *return*, quindi non è possibile assegnare in questo modo ad una stringa un valore costituito da più parole.
Per fare ciò si deve utilizzare una funzione di libreria (dichiarata nell'include file **<string>**) di nome **getline()**, che vuole come argomenti l'input stream e la variabile di tipo stringa da assegnare; il suo comportamento è di leggere l'input stream fino al successivo *new line* e di assegnare quanto letto alla variabile string:

```
string s;
getline(cin, s);
```

In questo modo ad **s** verrà assegnata l'intera stringa scritta su terminale, compresi gli spazi, fino al *return* (escluso).

### Concatenazione delle string

L'operatore + applicato alle **string** ha il significato di **concatenazione**. La concatenazione può operare tra string e tra string literal, generando una string il cui valore è la concatenazione delle due string (o string literal) utilizzate:

```
string s1 = "Ciao, ", s2 = "mondo.";
string s3 = s1+s2; // s3 vale "Ciao, mondo."
s1 += s3; // s1 ora vale "Ciao, Ciao, mondo."
string s4 = s2 + " Hello."; // s4 vale "Ciao, mondo. Hello."
```

### Accesso ai singoli elementi della string

Quando si utilizza una variabile string è possibile accedere al singolo elemento della stringa tramite l'utilizzo delle parentesi quadre ([]), trattando la stringa come se fosse un array di char:

```
string s = "Get";
s[0] = 'S'; // ora s vale "Set"
s[10] = 'b'; // errore
```

Nota: non è possibile utilizzare questo metodo per modificare la lunghezza della stringa, o più in generale per assegnare un valore ad un elemento della stringa oltre l'ultimo valore definito. In questo caso l'operazione potrebbe generare errori in esecuzione, poiché si andrebbe a scrivere in una zona di memoria indefinita.

### Confronto di variabili string

Le stringhe possono essere confrontate tramite gli operatori ==, **!=**, < e >, dove per *minore* si intende quella che viene prima in ordine alfabetico:

```
string s1 = "uno";
string s2 = "due";
if (s1 != s2) { ... } // vero
if (s1 < s2) { ... } // falso
if (s2 =\,= "duetto") { ... } // falso
```

Due stringhe sono uguali se sono uguali il contenuto e la lunghezza.

### Metodi delle variabili di tipo string

I principali metodi di cui si può fare uso con le variabili di tipo **string** sono descritti di seguito. Negli esempi ipotizziamo di utilizzare una string **s** di valore "Ciao, mondo.":

- **length()** o **size()**: ritornano la lunghezza di una stringa.

  ```
  int l = s.length(); // assegna ad l la lunghezza
                      // di s, cioe' 11
  ```

- **find()**: ricerca di una sottostringa. Accetta come primo argomento una stringa, un char * o una string literal, e come secondo argomento opzionale un intero che specifica la posizione di partenza per la ricerca (0 se non specificato, cioè ricerca a partire dall'inizio); ritorna un int che specifica la posizione in cui comincia la sottostringa:

  ```
  string s = "Ciao, mondo.";
  int pos = s.find("mon") // ritorna il valore 6
  ```

  Nota: per segnalare l'assenza della sottostringa cercata, **find()** ritorna un valore speciale che significa *posizione invalida*. Questo valore viene definito nell'include file **<string>** e si indica come **string::npos**:

  ```
  if(s.find("Hello") == string::npos) {
    cout << "sottostringa non trovata" << endl;
  }
  ```

- **find_first_of()** ricerca la prima occorrenza di un insieme di caratteri. Accetta come primo argomento una string, una string literal o un char *, e come secondo argomento opzionale un intero che specifica la posizione di partenza per la ricerca (0 se non specificato); ritorna un int che specifica la posizione in cui compare il primo tra i caratteri che costituiscono il primo argmento (**string::npos** se non ne viene trovato alcuno):

  ```
  s.find_first_of("aeiou"); // ritorna 1 (la i di "Ciao")
  ```

  Il metodo **find_first_not_of()** è del tutto analogo, ma ritorna la posizione del primo elemento della stringa che **non è** compreso nei caratteri che costituiscono il primo argomento.

- **replace()** sostituisce una porzione di una stringa con un'altra stringa. Vuole come argomenti un intero indicante la posizione del pezzo da sostituire, un intero indicante la lunghezza del pezzo da sostituire, ed una string (o char *, o string literal) indicante i caratteri con cui sostituire la porzione. La lunghezza delle parti da sostituire può essere diversa:

```
s.replace(6, 5, "a tutti") // s diventa "Ciao, a tutti"
```

  Nota: se il secondo argomento è zero, l'istruzione equivale ad inserire la parte nuova nella stringa vecchia, nella posizione specificata dal primo argomento.

- **substr()** ritorna una porzione della stringa. Vuole come argomenti un int che specifica l'inizio della sottostringa, ed un int che ne specifichi la lunghezza:

```
string ss = s.substr(3, 5); // ss sara' "ao, m"
```

- **c_str()** ritorna uno char * che costituisce una stringa di tipo C, completa di terminatore, che ha lo stesso valore della string. Questo metodo è indispensabile per tutte le funzioni che vogliono come argomento una stringa di tipo C:

```
string fileName = "dati.lis";
ifstream inFile;
inFile.open(fileName.c\_str());
```

## 5.15 Input ed output: lo stream

Il C++ gestisce l'Input/Output dei dati attraverso un concetto astratto, lo **stream**, che permette di trattare le operazioni di lettura e scrittura su terminale, su disco (cioè su file) o in memoria su variabili di tipo stringa, in modo omogeneo, cioè tramite gli stessi operatori.

Lo stream di input è un oggetto dal quale i dati arrivano come un flusso continuo di caratteri; questo può essere associato tramite opportune definizioni ad un terminale, ad un file aperto in lettura o ad una stringa.
In ogni momento è definita una posizione sullo stream (la posizione del cursore sul terminale, un punto specifico di un file aperto in lettura, un determinato carattere di una stringa) dalla quale la successiva operazione di lettura inizierà a leggere un dato.
Completata la lettura, il posizionamento sullo stream si sarà spostato oltre il dato letto, cioè sul carattere successivo all'ultimo carattere letto, im modo da poter leggere il dato successivo con la prossima lettura.

Similmente, lo stream di output è un oggetto sul quale i dati vengono scritti come un flusso continuo di caratteri. Come per l'input, anche l'output stream può essere associato ad un terminale, ad un file aperto in scrittura o ad una stringa.
Anche l'output stream è gestito tramite un puntatore alla posizione in cui verrà srcitto il dato specificato nella operazione di scrittura successiva.
Nel seguito del paragrafo si indicherà con il termine generico **stream** un qualunque tipo di stream (terminale, file, stringa o altro).

### 5.15.1 Stream di terminale: cin, cout, cerr

La Standard Template Library definisce degli stream di default per eseguire I/O da terminale. Per utilizzarli si deve inserire l'opportuna istruzione nel codice per includere il file **iostream** che li definisce:

```
#include <iostream>
```

In questo file vengono definite le classi per la gestione dell'input da terminale (**istream**) e dell'output da terminale (**ostream**).
Vengono inoltre definiti, tra le altre cose, i seguenti oggetti:

cin: lo stream per effettuare operazioni di input da terminale (lo *standard input*); è un oggetto della classe **istream**;

cout: lo stream per effettuare operazioni di output verso il terminale (lo *standard output*); è un oggetto della classe **ostream**;

cerr: lo stream per effettuare operazioni di output verso il terminale (lo *standard error*); è un oggetto della classe **ostream**;

Utilizzando queste definizioni possiamo effettuare tutte le operazioni di I/O su terminale senza dover definire nulla di esplicito nel nostro sorgente.

### 5.15.2 Gli operatori di input ed output

Le operazioni di scrittura su stream e di lettura da stream vengono fatte utilizzando specifici operatori (<< e >>), metodi delle classi stream o funzioni globali.

#### Input

L'operazione di input di un dato consiste nella interpretazione di una sequenza di caratteri come valore da assegnare ad una variabile. Dato uno stream di input (ad esempio **cin**), potremo utilizzare la seguente istruzione:

```
int i;
double d;
string s;
cin >> i >> d >> s;
```

Per poter leggere la seguente riga da terminale:

```
12 33.564 Mario
```

ed assegnare alle variabili **i**, **d** e **s** rispettivamente i valori **12**, **33.564** e "**Mario**". L'operatore >> si occupa di interpretare la sequenza di caratteri sullo stream per identificare il corretto valore da assegnare alla variabile che lo segue, a seconda del suo tipo.
Si può vedere come sia possibile concatenare le operazioni di input tramite l'utilizzo dell'operatore >> in sequenza, uno per ciascuna operazione.

L'operatore >> legge i caratteri relativi ad una singola operazione di input fino alla presenza di un carattere che non è compatibile con il tipo di dato da leggere. Lo spazio ed il ritorno a capo (*line feed*) sono sempre considerati terminatori della sequenza da leggere, ed è bene separare da spazi o da ritorno a capo i valori delle singole variabili da inserire. Se noi, nell'esempio precedente, avessimo omesso gli spazi:

```
1233.564Mario
```

la prima operazione di lettura (quella relativa alla variabile intera **i**) sarebbe proseguita fino al punto (il primo carattere incompatibile con una variabile **int**), la seconda sarebbe quindi iniziata con il punto e proseguita fino al carattere **M** (il primo carattere incompatibile con una variabile **double**) e la terza sarebbe iniziata dalla **M** per proseguire fino alla fine della riga, assegnando così i valori **1233** ad **i**, **0.564** a **d** e "**Mario**" a **s**.
Data la facilità con cui risulta ambigua la separazione tra i valori da leggere, è vivamente sconsigliato affidarsi a queste regole per l'input dei dati; separare i dati in lettura con spazi o con ritorno a capo rende meno frequente il cadere in errore.

Poiché lo spazio è sempre considerato dall'operatore >> come terminatore del dato da leggere, non è possibile leggere con questo operatore parole separate da spazi ed inserirle in una unica variabile di tipo **string**. Se ad esempio si vogliono leggere da terminale il nome ed il cognome di una persona e metterli in una unica variabile, si potrà fare nel seguente modo:

```
string nome, cognome, nomeCompleto;
cin >> nome >> cognome;
nomeCompleto = nome+" "+cognome;
```

L'operatore >> è capace di gestire tutte le variabili di tipo nativo; è inoltre definita l'interpretazione di altri tipi di variabile, tra le quali, ad esempio, le **string**.
Come vedremo in seguito, per ogni classe è possibile definire il comportamento dell'operatore >> in modo da poter leggere da stream una variabile di una qualunque classe in modo opportuno.

### Output

Quanto detto per l'input vale anche per l'output.
L'operazione di output di un dato consiste nella scrittura di una sequenza di caratteri idonea a rappresentare il valore di una certa variabile. Dato uno stream di output (ad esempio **cout**), potremo utilizzare la seguente istruzione:

```
int i = 20;
double d = -1.3;
string s = "Carlo";
cout << i << " " << d << " " << s << endl;
```

per poter visualizzare su terminale i valori delle variabili indicate. In questo caso otterremo su terminale la scritta:

```
20 -1.3 Carlo
```

con ritorno a capo.
L'operatore << si occupa di costruire e scrivere sullo stream la sequenza di caratteri idonea a rappresentare i valori delle variabili indicate.
Il ritorno a capo è dovuto all'ultima operazione di scrittura (**<< endl**) che viene interpretata come inserimento di un ritorno a capo. Il *manipolatore* **endl** è definito anch'esso nell'include file **<iostream>**.
Anche in output è possibile concatenare le operazioni tramite l'utilizzo dell'operatore << in sequenza, uno per ciascuna operazione.

L'operatore << scrive su output solo i caratteri necessari a visualizzare i valori delle variabili, senza inserire spazi tra una scrittura e l'altra; sta al programmatore separare l'output delle sue variabili inserendo spazi o ritorno a capo in modo opportuno (come fatto nell'esempio).
Tramite l'operatore << si possono leggere tutte le variabili di tipo nativo e molte delle variabili definite nelle STL.
Anche per l'output è inoltre possibile definire il comportamento dell'operatore << in relazione ad un oggetto di tipo non nativo, in modo da poter scrivere una variabile di una qualunque classe su stream in modo opportuno.

### Funzioni alternative per le operazioni di I/O

Esistono altri metodi per effettuare operazioni di I/O senza una formattazione predefinita.
Per l'input si dispone, tra le altre, di:

- get(char &c) per la lettura di un singolo carattere in c;
- read(char * p, int n) per la lettura di n caratteri nel buffer p;

Per la scrittura in output, si possono utilizzare i metodi corrispondenti:

- put(char &c) per la scrittura di un singolo carattere;

- read(char * p, int n) per la scrittura di un buffer lungo **n**;

Quelli appena descritti sono metodi di oggetti di tipo stream; ad esempio, per copiare un input stream **in** in un output stream **out** si potrà scrivere:

```
char c;
while(in.get(c)) { // leggo il carattere dall'input stream
   out.put(c); // scrivo il carattere nell'output stream
}
```

Vale la pena citare una funzione globale, definita nell'include file **<string>**, per leggere una riga da input stream e metterla in una variabile di tipo **string**:

```
string s;
getline(in, s);
```

Utilizzando la **getline()** alla variabile **s** viene assegnato il contenuto dell'intera riga scritta sull'input stream **in**, anche se comprende spazi.

### 5.15.3 Stato dello stream

Dato uno stream **s**, la variabile **s** può essere sottoposta a test per verificare che le operazioni effettuate sullo stream fino a quel momento non abbiano incontrato errori:

```
s << "Ciao, mondo" << endl;
if(!s) cout << "Errore in scrittura";
```

È opportuno effettuare il test ogni qualvolta non si ha la garanzia che le operazioni abbiano successo (ad esempio nell'utilizzo di I/O su file).

### 5.15.4 Formattazione dell'I/O

Può capitare di voler scrivere in output o leggere da input il valore delle variabili secondo un formato particolare. L'utilizzo dello stream permette di definire il formato dell'I/O a seconda delle esigenze. Sono infatti definiti metodi per le classi che costituiscono gli stream e manipolatori (oggetti utilizzabili come argomento degli operatori << e >>) per controllare il formato dell'I/O.

#### Output di interi

L'output dei numeri interi viene eseguito per default in notazione decimale. È possibile modificare questo comportamento tramite opportuni manipolatori:

```
int i = 16;
cout << i << endl;
cout << hex << i << endl;
cout << oct << i << dec << endl;
```

Il manipolatore **hex** viene utilizzato per scrivere in output gli interi in forma esadecimale, **oct** per il formato ottale, **dec** per il formato decimale. Le istruzioni dell'esempio comporteranno la scritta su terminale di **16 10 20**, corrispondenti al numero 16 nei diversi formati.
Va osservato che l'utilizzo di un manipolatore modifica il comportamento dello stream in modo permanente: una volta specificato un formato, questo verrà utilizzato per tutte le scritture successive, fino a che non verrà specificato un formato diverso. Nell'esempio l'utilizzo del manipolatore **dec** alla fine non produce nessun output, ma riporta lo stream **cout** nella sua condizione di default, cioè ad utilizzare in futuro il formato decimale.
I manipolatori **hex**, **dec** e **oct** non modificano il comportamento dello stream per l'output di variabili diverse da interi.
I manipolatori **hex**, **dec** e **oct** sono definiti nell'include file **<iostream>**.

### Output di floating point

Il formato dell'output per i numeri a virgola mobile viene determinato dal tipo di formato e dalla precisione.
Vi sono tre tipi di formato:

- **default**: la scelta del formato viene effettuata internamente a seconda del valore della variabile da scrivere in output.

- **scientific**: il numero viene scritto in notazione scientifica (la prima cifra non nulla seguita da punto, dalle restanti cifre e dall'esponente, in due cifre con segno, preceduto dalla lettera **e**. La precisione determina il massimo numero di cifre riportate dopo la virgola. Ad esempio, il numero **123.45678** in precisione 4 sarà rappresentato come **1.2346e+02**. Va osservato che il numero non viene troncato, ma arrotondato.

- **fixed**: il numero viene scritto senza l'indicazione dell'esponente, con la parte intera, il punto e la parte decimale. La precisione determina il massimo numero di cifre decimali riportate in output. Ad esempio, il numero **0.000123** in precisione 5 verrà rappresentato come **0.00012**.

Purtroppo non tutti i compilatori rendono disponibili manipolatori per modificare il formato dei numeri in virgola mobile, per cui verrà mostrato un esempio di come ottenere ciò tramite l'utilizzo del metodo **setf()** delle classi stream.
L'utilizzo del formato **scientific** si ottiene nel seguente modo:

```
double d = 123.45678;
cout.setf(ios::scientific, ios::floatfield);
cout << d << endl; // scrivera' 1.2346e+02 in output
```

L'utilizzo del formato **fixed** si ottinene nel seguente modo:

```
cout.setf(ios::fixed, ios::floatfield);
cout << d << endl; // scrivera' 123.4568 in output
```

Si può tornare al formato di default con l'istruzione:

```
cout.setf(0, ios::floatfield);
```

I valori di output riportati negli esempi sono relativi ad una **precision** pari a 4.

Il controllo della **precisione** si ottiene tramite il manipolatore **setprecision()**, che accetta un parametro intero per indicare il valore desiderato. Il valore di default è 6:

```
double d = 123.456789;
cout.setf(ios::fixed, ios::floatfield);
cout << setprecision(3) << d << endl; // avremo in output 123.457
cout << setprecision(5) << d << endl; // avremo in output 123.45679
```

Il manipolatore **setprecision()** è definito nell'include file **<iostream>**.

#### Lunghezza dell'output

Talvolta è opportuno controllare il numero di caratteri scritti in output a prescindere dalla lunghezza del numero che vogliamo scrivere. Per fare questo possiamo utilizzare il manipolatore **setw()** che accetta un parametro intero indicante il numero di caratteri minimo da scrivere in output in corrispondenza della successiva operazione di scrittura.
Il manipolatore **setw()** è definito nell'include file **<iomanip>**:

```
#include <iomanip>
...
int i = 12, j = 34;
cout << i << setw(5) << j << endl;
```

Queste istruzioni comporteranno la scrittura su terminale della variabile **i**, e della variabile **j** scritta in 5 caratteri, con giustificazione a destra. I caratteri non occupati dal valore della variabile da scrivere vengono riempiti con spazi (nell'esempio avremo in output **12** seguito da tre spazi e da **34**).

È possibile scegliere il carattere da utilizzare come riempitivo (cioè per i caratteri non occupati dal valore della variabile da scrivere in output) tramite il manipolatore **setfill()**, che accetta come parametro una variabile di tipo **char** contenente il carattere da utilizzare:

```
double d = 1.23;
cout << setw(5) << setfill('#') << d << endl;
```

Queste istruzioni produrranno in output **##1.23**.
Anche il manipolatore **setfill()** è definito nell'include file **<iomanip>**.

Nota: a differenza delle altre operazioni sulla formattazione, le modifiche operate tramite i manipolatori **setw()** e **setfill()** sono relative alla sola operazione di output che segue l'utilizzo di questi manipolatori. Se si desidera utilizzare questo formato per diverse operazioni di output, anche consecutive, i manipolatori devono essere utilizzati ogni volta.

### 5.15.5 Definizione degli operatori di I/O per variabili di tipo non nativo

Come detto, una volta definita una classe è possibile definire il comportamento degli operatori di I/O per questa classe. Consideriamo come esempio la classe **complex** per la gestione dei numeri complessi:

```
class complex {
public:
   ...
   void Set(double r, double i) { real = r; img = i; }
   double GetReal() { return real; }
   double GetImg() { return img; }
   ...
private:
   double real, img;
};
```

Per definire l'operazione di scrittura in output di un oggetto di tipo **complex** dovremo definire la seguente funzione globale:

```
ostream &operator<<(ostream &out, complex &c)
{
   return out << "(" << c.Real() << "," << c.Img() << ")";
}
```

Come si può vedere, l'operazione di output dell'oggetto **complex** si risolve nella scrittura su stream delle sue componenti reale ed immaginaria, tra parentesi tonde e separate da virgola. Si osservi che il fatto che la funzione ritorni una referenza all'**ostream** utilizzato permette di concatenare successive operazioni di scrittura in output:

```
...
complex c(1.2, -3.4);
cout << c << endl;
```

L'operazione **<< endl** opera sul valore ritornato dalla operazione precedente (**cout << c**), che è appunto una referenza a **cout**.

In modo analogo si può definire l'operazione di input:

```
istream &operator>>(istream &in, complex &c)
{
   double r, i;
   in >> r >> i;
   c.Set(r, i);
   return in;
}
```

Questo permette di eseguire l'istruzione:

```
...
complex c;
cin >> c;
```

leggendo due double consecutive da input, interpretandole rispettivamente come parte reale e parte immaginaria del numero complesso.

Si osservi come anche in questo caso la funzione globale ritorni una referenza all'input stream utilizzato, in modo da poter concatenare operazioni di input.
Si osservi infine che l'argomento **complex** va passato come referenza non solo per motivi di efficienza (come nel caso della funzione per l'output), ma necessariamente in quanto la lettura da stream comporta una modifica dell'oggetto passato come argomento.

### 5.15.6 Input/Output su file

Le operazioni di I/O su file in C++ vengono eseguite tramite una serie di classi definite nell'include file **<fstream>**, che permette di utilizzare tutta la struttura dello stream per i file.
Tutto quanto detto sulle operazioni di I/O da terminale valgono anche per l'I/O da file stream; l'unica differenza è che le librerie non definiscono a priori variabili per i file, che andranno quindi definite ed inizializzate.

#### Lettura da file

Per leggere un file la cosa più semplice è definire una variabile di tipo **ifstream** (input file stream) e di associarla al file che si vuole leggere, mediante l'utilizzo del metodo **open()** o direttamente alla definizione della variabile stessa, attivando l'opportuno costruttore:

```
#include <fstream>
...
ifstream file1("dati1.dat");
if(!file1) cout << "Non posso leggere il file dati1.dat" << endl;
ifstream file2;
file2.open("file2.dat");
if(!file2) cout << "Non posso leggere il file dati2.dat" << endl;
```

Sia il costruttore che il metodo **open()** accettano come argomento una C-style string (cioè un char * che rappresenta un array di char contenente il nome del file, terminato da `'0'`). Se utilizziamo una variabile di tipo **string** per il nome del file, dovremo quindi utilizzare il suo metodo **c_str()**:

```
#include <fstream>
#include <string>
...
string s = "dati1.dat";
ifstream file1(s.c_str());
if(!file1) cout << "Non posso leggere il file " << s << endl;
s = "dati2.dat";
ifstream file2;
file2.open(s.c_str());
if(!file2) cout << "Non posso leggere il file " << s << endl;
```

Quando la lettura dei dati è terminata, si può utilizzare il metodo **close()** per chiudere il file e rimuovere la associazione tra la nostra variabile di tipo **ifstream** ed il file stesso.
Dopo una **close()**, la stessa variabile può essere utilizzata per aprire un altro file in lettura, tramite una nuova chiamata al metodo **open()**:

```
ifstream inFile("dati1.dat");
if(!inFile) return 1;
...
inFile.close();
inFile.open("dati2.dat");
if(!inFile) return 2;
...
inFile.close();
```

Una volta aperto il file, la variabile di tipo **ifstream** si utilizza esattamente nel modo in cui si utilizza **cin**, e vale tutto quanto visto fin'ora per l'operatore >> e per i modificatori di formato. Il seguente esempio riempie un array di double leggendo i valori da un file:

```
double v[100];
ifstream in("array.dat");
if(!in) return 1;
for(int i=0; i<100; ++i) in >> v[i];
in.close();
```

### Scrittura su file

Per scrivere su file si deve utilizzare una variabile di tipo **ofstream** (output file stream) ed associarla ad un file aperto in scrittura, in modo del tutto simile a quanto visto per l'input da file:

```
#include <fstream>
...
ofstream outFile("dati.lis");
double v[5] = {1., 2., 3., 2., 1.};
if(!outFile) return 1;
for(int i=0; i<5; ++i) outFile << v[i] << endl;
outFile.close();
```

Nell'esempio visto, viene creato un file in cui vengono scritti i valori degli elementi dell'array **v**, uno per riga.
Quando si apre in scrittura un file già esistente, il contenuto precedente viene eliminato; è possibile specificare un parametro (sia per la **open()** che all'atto della definizione della variabile) per poter aprire il file in *append mode*, vale a dire per conservare il contenuto esistente, e cominciare ad aggiungere caratteri dal fondo del file:

```
...
ofstream outFile("dati.lis", ios::app);
ofstream ou;
ou.open("lista.dat", ios::app);
```

Una volta aperto il file in scrittura, si può utilizzare la variabile di tipo **ofstream** esattamente come si utilizza **cout**, utilizzando l'operatore << e tutti i modificatori e membri già visti:

```
#include <fstream>
ifstream in("dati.dat");
ofstream out("lista.lis");
if(!in || !out) return 1;
out << "Copia del file dati.dat" << endl;
char c;
while(in.get(c)) out.put(c);
in.close();
out.close();
```

In questo esempio si esegue una copia del file **dati.dat** nel file **lista.lis**, con l'aggiunta di una riga iniziale. Si vede come gli stream di file si trattano in modo del tutto omogeneo agli stream di terminale.

### 5.15.7 Stream di stringhe

Il concetto di stream è stato esteso per poter leggere da una stringa o scrivere su una stringa mantenendo la stessa struttura di funzionamento degli altri stream.
Tutte le definizioni necessarie ad utilizzare le classi dello string stream sono contenute nell'include file **<sstream>**, che va pertanto incluso nel nostro sorgente.

#### Input da string

Per poter leggere il contenuto di una stringa come se fosse uno stream si deve definire una variabile di tipo **istringstream** (input string stream) ed associarla alla stringa desiderata:

```
#include <sstream>
#include <string>
...
string s = "uno due tre quattro", w;
istringstream ist;
ist.str(s);
while(ist >> w) cout << w << endl;
```

In questo esempio si associa la variabile **ist** di tipo **istringstream** alla stringa **s**, e poi la si utilizza in modo analogo a **cin** per leggere dallo string stream le diverse parole di cui è costituita la stringa **s**, che vengono poi scritte su terminale una per riga.
L'associazione tra input string stream e stringa viene operata dal metodo **str()** che vuole come argomento la stringa da associare allo stream.
Lo stesso risultato si può ottenere all'atto della definizione della variabile **istringstream**, richiamando l'opportuno creatore:

```
...
istringstream ist(s);
...
```

Va osservato che, una volta associata la stringa allo stream di input, modifiche successive alla stringa non vengono viste attraverso lo stream. Finché lo string stream è associato ad una stringa, dallo stream si potrà vedere il contenuto della stringa al momento della sua associazione.
Ogni chiamata al metodo **str()** passando come argomento una stringa determina una nuova

associazione dello stream; l'associazione precedente viene automaticamente chiusa e lo stato dello stream riportato alle condizioni iniziali.
Il modo per modificare una stringa già associata allo stream e vedere la stringa modificata attraverso lo stream è proprio di riassociare lo stream alla stringa stessa:

```
string s = "a b c d", w;
istringstream is(s);
is >> w; // w vale "a"
...
s = "e f g h i";
is.str(s);
is >> w; // w vale "e"
...
```

### Output su string

In modo del tutto analogo esiste una classe per la gestione dell'output su stringa attraverso la definizione degli stream, la **ostringstream** (output string stream).
La stringa che contiene quanto scritto nell'output string stream si ottene tramite il metodo **str()** senza argomento:

```
#include <sstream>
ostringstream ost;
double d = 12.4;
ost << "Valore di d: " << d << endl;
int i = 20;
ost << "Valore di i: " << i << endl;
string s = ost.str();
```

In questo esempio, la string **s** conterrà la stringa:

```
Valore di d: 12.4
Valore di i: 20
```

con un ritorno a capo in mezzo ed uno alla fine (dati dai modificatori **endl**).

## 5.16 Bibliografia sulla sintassi del C++

Esistono molti manuali che illustrano la sintassi del **C++**, a diversi livelli di completezza e di complessità.
Il **C++** è un linguaggio progettato per la programmazione *object oriented*, per cui tutti i manuali di sintassi dedicano molti capitoli a questo aspetto della programmazione, che non è materia del corso di Laboratorio di Calcolo.
Ne consegue che il manuale del **C++** va visto come una referenza piuttosto che un libro da leggere dall'inizio alla fine.
Ad una prima lettura è molto opportuno isolare gli argomenti di interesse per non ritrovarsi ad affrontare questioni la cui complessità può essere fuorviante.
Per chi volesse approfondire ed ampliare lo studio sulla sintassi del **C++** e delle cose viste dirante il corso, si consiglia il manuale scritto dall'inventore del C++:

**Bjarne Stroustrup, C++ Programming Language, 3rd edition, Ed. Addison-Wesley**.

In particolare per approfondire quanto scritto nelle dispense si consiglia, oltre alla introduzione, la lettura dei capitoli **4** e **5** per i tipi di variabile, il capitolo **6** per le istruzioni di controllo di flusso, il capitolo **7** per le funzioni e la programmazione strutturata, il capitolo **9** per quanto concerne la programmazione modulare, gli include files e le librerie, il capitolo **10** per le classi, il capitolo **11** paragrafi 1, 2 e 3 per l'overload degli operatori, il capitolo **20** per le **string** ed il capitolo **21** per gli **stream**.
Un ottimo testo e referenza sulla **Standard Template Library**, è il

**Nicolai M. Josuttis, The C++ Standard Library, Ed. Addison-Wesley**.

Questo testo va visto come ulteriore approfondimento in quanto copre in modo esauriente le questioni relative alla **STL**, ma presuppone la conoscenza del linguaggio e delle caratteristiche della programmazione *object oriented*. I capitoli **11** e **13** coprono ed espandono quanto visto su **string** e **stream**.
Per ultimo va citato il libro degli inventori del **C**, che costituisce una lettura indispensabile per tutti coloro che vogliano cimentarsi nella programmazione in questo linguaggio:

**B.W. Kernigan, D. M. Ritchie, The C Programming Language, Ed. Prentice Hall P T R**

disponibile anche in lingua italiana. Benchè il corso abbia come argomento la programmazione in **C++**, in questo manuale si può trovare un valido aiuto, soprattutto all'inizio, in quanto la parte comune ai due linguaggi viene trattata in modo semplice e di facile comprensione.

# Capitolo 6

# Tecniche algoritmiche e numeriche di base

## 6.1 Introduzione

Illustreremo nel seguito alcune tecniche elementari di manipolazione dei dati e di calcolo numerico. Per ogni argomento tratteremo essenzialmente i soli concetti fondamentali e spesso ometteremo discussioni, anche rilevanti, su questioni di efficienza ed ottimizzazione degli algoritmi presentati. Lo scopo è, come al solito, quello di fornire le informazioni di base che possano consentirvi di risolvere semplici problemi e che possano servire da stimolo per uleriori approfondimenti.

## 6.2 I vettori

Il vettore è l' elemento chiave per la realizzazione di procedure indicizzate, nelle quali cioè ci si trovi a dover manipolare un insieme numerabile di oggetti simile che devono essere elaborati in sequenza o comunque organizzati in modo ordinato. Ad ogni oggetto si assegna un indice, ovvero un numero positivo che corrisponde alla posizione dell' oggetto all'interno del vettore.

### 6.2.1 Manipolazione dei vettori

La struttura sintattica di base per la manipolazione dei vettori è ovviamente il ciclo `for`; in effetti i vettori hanno lunghezza definita a priori (a livello di compilazione o, nel caso della allocazione dinamica, a livello di esecuzione del programma), e di conseguenza il ciclo `for` consente di manipolare sequenzialmente tutti gli elementi. Consideriamo l' esempio seguente, nel quale leggiamo una lista di nomi, la copiamo in un vettore e rileggiamo il contenuto:

```
#include <string>
#include <iostream>

main(){
  int i,n_names;
  string *names = NULL;

  // Legge il numero di nomi e alloca il vettore
  cout << "Quanti sono i nomi ?  ";
```

```
  cin >> n_names;
  names = new string[n_names];

  // Carica nel vettore i nomi
  for (i=0; i<n_names; i++) {
    cin >> names[i];
  }

  // Stampa il contenuto del vettore
  for (i=0; i<n_names; i++) {
    cout << names[i] << endl;
  }

  delete[] names;

}
```

È chiaro da questo esempio come il vettore sia uno strumento molto andatto in tutte quelle circostanze in cui il numero di oggetti da manipolare è noto all' inizio dell' elaborazione e resta costante.
Le matrici, avendo due indici, richiedono due cicli `for` uno dentro l' altro, come si vede nell' esempio seguente dove stampiamo il contenuto di un array a due indici di float:

```
float matrix[10][20];
int i,j;
...

for (i=0; i<10; i++) {
  for (j=0; j<20; j++) {
    cout << matrix[i][j];
  }
  cout << endl;  // Va a capo al termine di ogni riga
}
```

È importante ricordare, specie nella manipolazione di array di grandi dimensioni (un megabyte o più), di rispettare con la sequenza degli indici la naturale disposizione in memoria degli elementi. In C++ questo significa che l' indice più a destra deve scorrere per primo, e quindi deve essere nel ciclo `for` più interno.
In alcuni casi è conveniente utilizzare un vettore per implementare un "buffer circolare"; si tratta di una struttura a $n$ elementi nella quale vengono memorizzate informazioni. Nel momento in cui è necessario scrivere l' infromazione $n+1$ si cancella la prima informazione scritta e si mette la $n+1$-ima al suo posto.
Nell' esempio seguente vediamo un modulo che implementa un buffer circolare di 100 numeri interi. La routine `add` consente di aggiungere un numero al buffer, la routine `read` mostra gli ultimi numeri scritti (fino a un massimo di 100).

```
#define BUFLEN 100

int nel = 0;      // numero di elementi presenti nel buffer
```

```
int pointer = 0;  // indice dell'elemento siccessivo
int buffer[BUFLEN];

void add(int i) {
  buffer[pointer] = i;
  if (nel < BUFLEN) nel++;
  pointer++; if (pointer >= BUFLEN) pointer = 0;
}

void read() {
  int i;
  int ip = pointer;

  for (i=0; i<nel; i++) {
    ip--; if (ip <0) ip = BUFLEN - 1;
    cout << buffer[ip] << endl;
  }
}
```

I buffer circolari possono essere utilizzati per implementare strutture di tipo FIFO (First In First Out) o LIFO (Last In First Out) di dimensione fissa.
La ricerca in un vettore è spesso di tipo euristico: si leggono tutti gli elementi e si confronatano con il valore desiderato; quando si trova l' elemento giusto si può usare l' istruzione `break` per interrompre il ciclo senza scandire gli elementi rimanenti; nel modulo appena visto aggiungiamo ad esempio una function che ritorna la posizione nel buffer di un dato numero o -1 se il numero non è nel buffer:

```
int find(int val) {
  int i;
  int ip = pointer;
  int pos = -1;

  for (i=0; i<nel; i++) {
    ip--; if (ip <0) ip = BUFLEN - 1;
    if (buffer[ip] == val) {
      pos = ip;
      break;
    }
  }
  return pos;
}
```

Nel caso si abbia a che fare con vettori di grnde dimensione, la ricerca euristica risulta inefficiente. Si usa talvolta la ricerca "a chiave". Supponiamo ad esempio di aver caricato un elenco telefonico in un vettore (gli elementi del vettore saranno tipo composti da un nome e un numero di telefono), e di voler ricercare un dato nome. Potremmo fabbricare un vettore ausilirario che memorizzi, per ogni lettera dell' alfabeto, l' indice del primo nome nell' elenco che comincia per quella lettera. Invece di cercare euristicamente in tutto l' elenco potremmo con questa tecnica cercare soltanto

in un sottoinsieme. Naturalmente il vettore ausilirario deve essere mantenuto aggiornato ogni volta che si aggiunge o si toglie un elemento dall' elenco.

### 6.2.2 Ordinamento degli elementi di un vettore

Ci sono molte tecniche per ordinare gli elementi di un vettore; la più intuitiva consiste nel cercare euristicamente nell' intero vettore l' elemento più piccolo, copiarlo nella prima locazione di un vettore ausiliario eliminandolo dal vettore originario e procedere coiìfino ad avere esaurito gli elementi. Una tecnica un po' più raffinata, detta "bubble sort" consiste nell' analizzare tutti gli elementi, partendo dall' ultimo, scambiando ogni elemento $x(i)$ con il precedente $x(i-1)$ se $x(i) < x(i-1)$; in questo modo si porta l' elemento più piccolo nella posizione iniziale. La procedura va poi ripetuta nel sotto-vettore ottenuto ignorando la prima componente, ed iterata fino a quando non si esegue più alcuno scambio o si è rimasti cin un sotto-vettore di un elemento. Ecco un esempio di codifica in C++ del "bubble sort":

```
void sort(int *vect, int n) {
  int i,swap,top,temp;

  if (n < 2) return;

  top = 0;
  do {
    swap = 0;
    for (i=n-1; i>top; i--) {
      if (vect[i] < vect[i-1]) {
        swap = 1;
        temp = vect[i];
        vect[i] = vect[i-1];
        vect[i-1] = temp;
      }
    }
    top++;
  } while(swap != 0 && top < n-1);
}
```

### 6.2.3 Aggiunta di elementi a un vettore

Siccome un vettore consiste in una sequenza statica e di lunghezza definita di elementi, a rigore non è possibile aggiungere un elemento a un vettore. In pratica si usa spesso il trucco di definire in modo generoso le dimensioni dei vettori, usandone poi solo una parte; in questo caso è possibile aggiungere un elemento anche nel mezzo del vettore, a patto di spostare tutti gli elementi successivi di una posizione. Inoltre, se si usa l' alloazione dinamica, è possibile sostituire il vettore con uno più grande se manca lo spazio. Come esempio, vediamo una routine che aggiunge elementi ad un vettore allocato dinamicamente mantenendo costantemente l' ordinamento crescente degli elementi:

```
int *vect = NULL;
int n_el = 0, max_el = 0;
```

```
void increase_size(int siz) {
  int i,*temp;

  // Crea un nuovo vettore
  max_el = max_el + siz;
  temp   = new int[max_el];

  // Copia il vecchio vettore nel nuovo
  for (i=0; i<n_el; i++) temp[i] = vect[i];

  // Cancella il vecchio vettore e riassegna il puntatore
  if (vect != NULL) delete[] vect;
  vect = temp;

}

int add_element(int val) {
   int i;

  // Controlla se c'e' ancora spazio
  if (n_el + 1 > max_el) increase_size(100);

  // Aggiungi il nuovo elemento al suo posto
  for (i=n_el++; i>0; i--) {
    if (vect[i-1]>val) {
      vect[i] = vect[i-1];
    } else {
      vect[i] = val;
      return i;
    }
  }
  vect[0] = val;
  return 0;

}
```

## 6.3 Strutture dinamiche

Come abbiamo visto i vettori posseggono il difetto di essere strutture "statiche", ovvero definite in modo rigido e poco adatte ad essere modificate durante l' elaborazione. Quando si trattano problemi complessi si ricorre quindi spesso ad altri tipi di strutture per l' immagazzinamento dati, che, seppure un poco più laboriose per il programmatore, risultano di gran lunga più flessibili e adattabili alle diverse esigenze. Tali strutture (che fanno largo uso di tecniche di allocazione dinamica della memoria) vengono dette strutture "dinamiche".

### 6.3.1 Liste

Una lista è un insieme lineare di elementi; ogni elemento possiede un collegamento all' elemento precedente ed a quello successivo nella lista. Dal punto di vista della programmazione, gli elementi della lista sono realizzati per mezzo di tipi composti, e le connessioni sono realizzate per mezzo di puntatori. Nell' esempio che segue vediamo la definizione degli elementi di una lista che contiene il solito elenco del telefono:

```
struct element {
  element *previous;
  element *next;
  string name;
  string city;
  int number;
};
```

Come si vede, a parte i dati, nella `struct element` sono presenti due puntatori che implementano il link all' elemento precedente ed al successivo. Vediamo ora come si può implementare una routine per per aggiungere un elemento alla nostra lista:

```
struct element *first = NULL;

void add(string name, string city, int number) {
  struct element *new_el,*i, *tmp;

  /* Per prima cosa creiamo un nuovo elemento copiamoci dentro i dati */
  new_el = new element;
  new->name = name;
  new->city = city;
  new->number = number;

  /* Adesso andiamo a cercare il primo elemento gia' presente nell'elenco con
     il nome successivo in ordine alfabetico a quello che stiamo inserendo */
  if (first == NULL) {
    first = new;
    new->previous = new->next = NULL;
    return;
  }
  i = first;
  while (i != NULL) {
    if (name < i->name) {
      /* A questo punto inseriamo il nuovo elemento subuto
         prima di quello trovato */
      new->previous = i->previous;
      i->previous = new;
      new->next = i;
      if (new->previous != NULL) (new->previous)->next = new;
      if (i == first) first = new;
      return;
```

```
    }
    if (i->next == NULL) {
      /* Se siamo arrivati all'ultimo elemento,
         aggiungiamo il nuovo alla fine */
      i->next = new;
      new->previous = i;
      new->next = NULL;
    }
    i = i->next;
  }
}
```

Questo semplice esempio illustra tutte le potenzialità delle liste: non ci si deve preoccupare delle dimensioni, in quanto una lista può essere espansa a piacimento, ed inoltre i nuovi elementi possono essere piazzati in qualunque posizione senza doversi preoccupare di spostare nessun altro elemento ma semplicemente sistemando in modo opportuno i valori di quattro puntatori. Le liste rendono inutili le procedure di ordinamento, in quantoi dati vengono sistemati a priori nell' ordine desiderato.

### 6.3.2 Ricerca di un elemento in una lista

Come abbiamo visto nell' esempio precedente la sintassi per la ricerca euristica in un albero è il ciclo `while` che si addice in effetti a tutti i loop in cui non sia noto a priori il numero di iterazioni.
Per quel che concerne le ricerche a chiave, si deve dire che è in questa situazione che le liste forniscono i migliori risulati. L' idea è quella di realizzare una lista contenente le chiavi di ricerca (nel caso dell' esempio fatto al Par. 6.2.1 si tratterebbe di una lista contenente le lettere dell' alfabeto; già qui però l' pproccio dinamico fornisce dei vantaggi: non è infatti necessario creare un elemento per ogni lettera, ma si può attendere che venga inserito il primo nome con una data iniziale per creare la chiave corrispondente; inoltre si può decidere, durente l' eleborazione, di usare le prime due lettere come chiave se i nomi con una data iniziale superano un dato valore massimo). Tra i dati degli elementi della lista delle chiavi, oltre a tutti i parametri che servono per eseguire correttamente il confronto con i nomi via via inseriti, ci sarà il puntatore al primo elemento di una lista di dati (nell' esempio gli elementi dell' elenco telefonico) che corrispondono a quella data chiave. Nel seguio vediamo una possibile implementazione della struttura, raffigurata schematicamente in Fig. 6.1:

```
struct element {
  element *previous;
  element *next;
  string name;
  string city;
  int number;
};

struct key {
  struct key *previous;
  struct key *next;
```

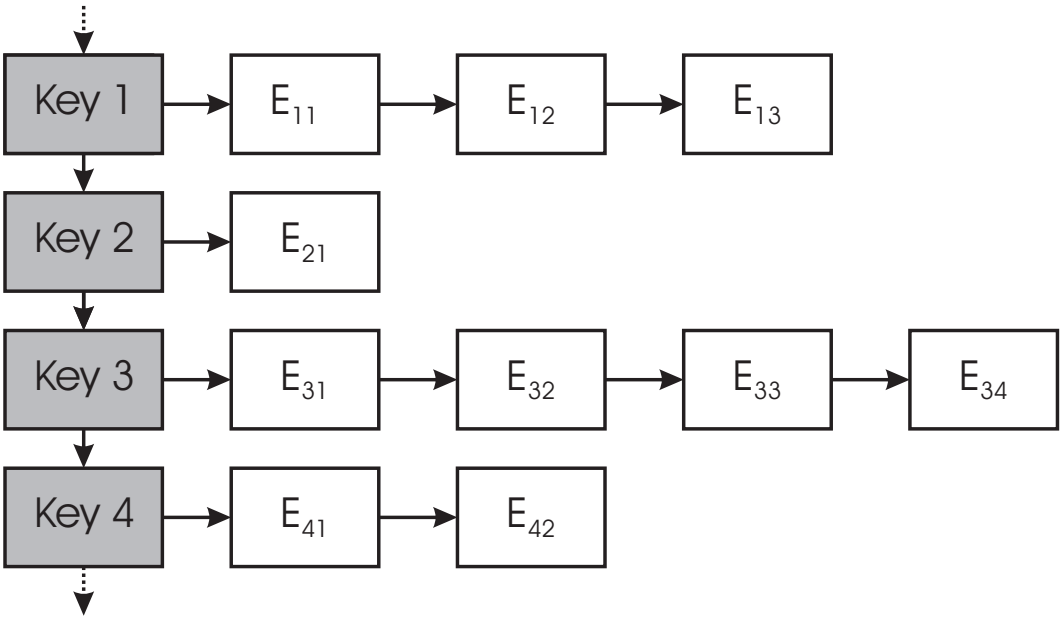


Figura 6.1: Schema di una struttura dati per la ricerca a chiave basata sulle liste.

```
  struct element *first;
  string keyval;
};
```

Anche senza scendere nei dettagli di questo esempio, possiamo capire come il concetto di "lista nella lista" possa essere ripetuto più volte, creando così strutture gerarchiche che possono essere utili nelle realizzazione di ricerche con molti livelli di chiavi (ad esempio un livello di chiavi per ogniuna delle prime 5 lettere del nome).
Vale la pena di osservare, infine, come ogni singolo elemento possa appartenere a più liste, a patto che si aggiungano nella struttura più coppie di puntatori "previous" e "next". In questo modo si possono implementare diversi criteri di ordinamento e di ricerca a chiave.

### 6.3.3 Alberi

Gli alberi sono strutture dinamiche ottenute come evoluzione delle liste. Ogni elemento possiede, oltre ai link "previous" e "next" anche un link al primo elemento di una lista figlia ed un link ad un elemento genitore (Fig. 6.2).
Di conseguenza l' implementazione in C potrebbe essere:

```
struct tree_element {
  tree_element *previous;
  tree_element *next;
  tree_element *up;
  tree_element *down;
  int type;
  string keyval;
  string name;
  string city;
```

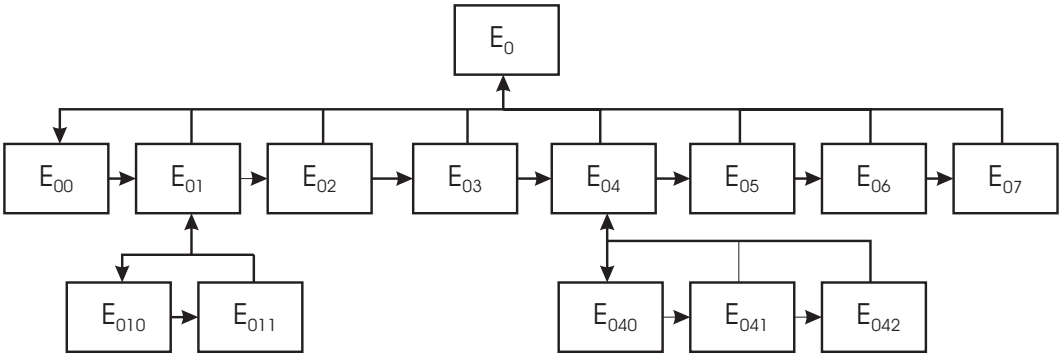


Figura 6.2: Schema di un albero.

```
  int number;
};
```

Sempre pensando al nostro elenco del telefono, una struttura ad albero ci consente di implementare una ricerca a chiavi su più livelli: in ogni ramo, la lista di livello più basso contiene i veri elementi dell' elenco, mentre le liste degli altri livelli sono liste di chiavi corrispondenti alle prime lettere del nome. Dato che gli elementi sono tutti uguali, per distinguere le chiavi dai dati (e le chiavi di diverso tipo tra loro) useremo la variabile `type`: varrà zero per i dati e `n` per la chiave corrispondente alla n-esima lettera del nome.
Un altro possibile esempio di uso degli alberi deriva dalla fisica delle particelle: nell simulazioni si devono seguire le traiettorie di particelle che attraversano mezzi densi. In caso di interazione, una particella può decadere in n particelle figlie, le quali a loro volta possono decadere e così via. La struttura dei decadimenti può essere mappata in un albero.

### 6.3.4 Ricerca di un elemento in un albero

Il modo naturale di eseguire una ricerca in un albero è di utilizzare il metodo ricursivo: si scrive una procedura che riceva un dato elemento come parametro e lo analizzi. Se si tratta dell' elemento cercato la procedura deve ritornare un codice di successo; se invece l' elemento non è quello giusto si deve eseguire un loop su tutti gli elementi della lista figlia; all' interno del loop la procedura richiamerà se stessa, con l' elemento i-mo della lista figlia come parametro. Se la chiamata su uno degli elementi della lista ritorna il codice di successo (il che significa che quell' elemento o uno dei suoi figli sono l' elemento cercato) il loop va interrotto ed il codice di successo va restituito al chiamante. In caso contrario si ritornerà un codice di fallimento (che vol dire che nessuno dei discendenti dell' elemento passato come parametro è quello buono.
Con questa tecnica si può esplorare l' intero albero chiamando una sola volta la routine con l' elemento di più alto livello come parametro. Vediamo un esempio di ricerca nel nostro elenco telefonico ad albero:

```
int find_tree(string name, tree_element *el) {

  tree_element *i;

  /* Controllioamo se el e' una chiave o un dato */
  if (el->type == 0) {
```

```
    /* Se e' un dato, controlliamo se il nome e' quello cercato */
    if (name == el->name) return 1;
  } else {
    /* Se e' una chiave, controlliamo se la lettera type-ima del nome cercato
       coincide con il valore della chiave */
    if (name[el->type-1] == el->keyval) {
      /* in caso affermativo esploriamo tutti gli elementi della lista figlia
         richiamando questa procedura */
      i = el->down;
      while (i != NULL) {
        if (find_tree(name, i) == 1) return 1;
        i = i->next;
      }
    }
  }
  /* 0 vuol dire che non si e' trovato nulla in questo ramo */
  return 0;
}
```

La procedura qui sopra contiene (almeno) un errore ed un grave difetto; trovateli per esercizio. Va anche notato che il nostro esempio contiene un' anomalia: gli elementi di tipo zero (i dati) non hanno figli per definizione. Questo non è vero in un albero generico, dove tutti gli elementi possono a priori avere discendenti.

## 6.4 Tecniche Numeriche

Il calcolo numerico consiste nella soluzione di equazioni o nel calcolo di integrali ottenuta per sostituzione di particolari valori numerici. Si tratta ovviamente di un metodo molto meno elegante rispetto alla soluzione analitica del problema, ma che rischia di essere la sola possibile in molti casi in cui una soluzione analitica non è realizzabile. Malgrado l' apparente rozzezza, non si deve commettere l' errore di sottovalutare la complessità delle tecniche numeriche: un approccio troppo "spensierato" può condurre a risultati totalmente sbagliati che pure hanno una apparenza del tutto ragionevole.
Nel seguito illustreremo i rudimenti del calcolo numerico. I metodi che illustreremo hanno di fatto solo valenza didattica, e sono del tutto indadatti (per precisone, efficienza e dominio di applicabilità) alla soluzione di problemi "seri". Per maggiori approfondimenti si può ricorrere ad un qualunque testo di calcolo numerico; citiamo il seguente:

`W. H. Press, S. A. Teukolsky, W. T. Wetterling, B. P. Flannering -`
`Numerical Recipes in C - Cambridge University Press`

che ha il singolare vantaggio di essere disponibile on–line all' indirizzo `http:`
`www.nr.com`.

### 6.4.1 Calcolo di integrali definiti

Il metodo intuitivo di calcolare un integrale definito di una funzione $f(x)$ non singolare, consiste nel dividere l' intervallo di integrazione in $N$ sottointervalli uguali. Siano $x_0, x_1, ..., x_N$ i punti che delimitano i sottointervalli e $h$ la lunghezza dei sottointervalli; potremo allora approssimare

l' integrale nell' intervallo $[x_i, x_{i+1}]$ con l' area sotto la retta che passa per i punti $(x_i, f(x_i))$ e $(x_{i+1}, f(x_{i+1}))$. In formule scriveremo:

$$\int_{x_0}^{x_n} f(x)dx \simeq \sum_{i=0}^{N-1} \frac{h}{2}(f(x_i) + f(x_{i+1}))$$

È facile convincersi che questa formula è esatta per una retta; si può anche dimostrare che l' errore $E$, ovvero la differenza tra l' integrale e la somma, è proporzionale ad $h^3 f''(\bar{x})$ dove $\bar{x}$ è un punto non noto interno all' intervallo di integrazione (da cui si ritrova il risultato che per una retta, che ha derivata seconda sempre nulla, l' errore è zero).
Si può altresì dimostrare che, raggruppando due a due gli intervalli si ottiene una formula del tipo

$$\int_{x_i}^{x_{i+2}} f(x)dx = h\Big(\frac{f(x_i)}{3} + \frac{4f(x_{i+1})}{3} + \frac{f(x_{i+2})}{3}\Big) + O(h^5 f^{(4)}(\bar{x}))$$

dove l' errore è proporzionale ad $h^5$ moltiplicato per la derivata quarta di $f$ in un punto interno all' intervallo (significa che è esatta per la polinomiali fino all' ordine $x^3$).
Le tecniche di integrazione presentate non possono ovviamente funzionare su intervalli illimitati e neppure nel caso che la unzione presenti singolarità all' interno dell' intervallo di integrazione.

### 6.4.2 Ricerca del minimo di una funzione

Per trovare il minimo (o il massimo) di una funzione continua e derivabile si sfrutta il fatto che in tale punto la derivata prima della funzione si annulla e cambia segno. partendo da un punto sufficientemente vicino al minimo ricercato (vuol dire che non ci devono essere altri minimi relativi nei paraggi) si calcola la derivata approssimandola con il rapporto incrementale su un intervallo finito di dimensione $h$:

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h}$$

Ci si sposta quindi di una quantità pari ad $h$ e si continua fino a che il rapporto incrementale non cambia segno. A questo punto si dimezza il passo di campionamento e si ripete l' operazione nella direzione inversa. Ci si può fermare quando l' intervallo di campionamento scende al di sotto di una quantità minima definita come errore tollerato sulla posizione dell' estremo.
Questo metodo funziona in modo soddisfacente solo con funzioni regolari e prive di estremi relativi. Inoltre è di facile applicabilità solo con funzioni di una variabile. Nelle esercitazioni utilizzeremo le routines della librerie ROOT per la determinazione dei minimi di funzione a molti parametri.

### 6.4.3 Il metodo dei minimi quadrati

Il metodo dei minimi quadrati viene utilizzato per determinare quale tra le funzioni appartenenti ad una data famiglia descriva al meglio un insieme di punti sperimentali affetti (ovviamente) da errore. La situazione tipica è quella in cui si dispone di un set di misure $y_i$ con $1 < i < N$ ciascuna affetta da un errore $\sigma_i$ e ciascuna corrispondente ad una particolare scelta di variabili indipendenti $\vec{x}_i$ (assumeremo che non ci siano errori sugli $\vec{x}_i$, ipotesi di solito non verificata. Nel corso di ESP1 vedrete come trattare gli errori sulle variabili indipendenti). Si dispone poi di una teoria che prevede una relazione funzionale tra $y$ e $\vec{x}$ espressa sotto forma di una famiglia di

funzioni del tipo $y = f(\vec{x}, \vec{p})$, dove $\vec{p}$ rappresenta un insieme di parametri fisici che determinano l' andamento delle funzioni $f$ ma che non sono a priori note. Il problema è quello di determinare quale scelta di $\vec{p}$ fornisce il migliore accordo tra i dati sperimentali e la funzione teorica.
Il metodo dei minimi quadrati consiste nell' affermare che la migliore scelta di $\vec{p}$ è quella che minimizza la funzione $\chi^2(\vec{p})$ definita come

$$\chi^2(\vec{p}) = \sum_{i=0}^{N} \Big( \frac{(y_i - f(\vec{x}_i, \vec{p}))^2}{\sigma_i^2} \Big)$$

Nel caso la funzione $f$ sia una retta, cioè

$$f(x, \vec{p}) = ax + b; \quad \vec{p} = (a, b)$$

il problema della minimizzazione del $\chi^2$ ha una soluzione analitica che avete visto a ESP1. In tutti gli altri casi ci si deve accontentare di una soluzione numerica, che può essere ottenuta con i metodi illustrati nel paragrafo precedente.
Se la miglior funzione teorica si adatta in modo accettabile ai dati sperimentali, ovvero se il valore del minimo $\chi^2$ è sufficientemente piccolo (imparerete nel corso del secondo semestre a dare un significato quantitativo, di tipo statistico, a queste affermazioni) il metodo dei minimi quadrati fornisce in sostanza una misura di $\vec{p}$ derivata dalla misura degli $y_i$. Questo significa che è necessario calcolare un errore sul valore di $\vec{p}$ ricavato: se ripetessimo la misura degli $y_i$ otterremmo valori diversi (sebbene entro gli errori), e quindi una diversa funzione $\chi^2$ da minimizzare ed un diverso valore di $\vec{p}$ che la minimizza. Quindi in qualche modo l' entità delle fluttuazioni degli $y_i$ determinano una più o meno grande fluttuazione dei valori di $\vec{p}$. Si può dimostrare che l' incertezza su $\vec{p}$ è data dall' insieme dei punti tali che:

$$\chi^2(\vec{p}) - \chi^2(\vec{p}_{min}) < 1$$

dove $\vec{p}_{min}$ è il valore di $\vec{p}$ che minimizza la funzione $\chi^2$. Pur senza dimostrare l' affermazione precedente, possiamo osservare che se abbiamo una sola misura $y_0$ ed un singolo parametro $p$ potremo scegliere il valore della funzione teorica $f$ in modo che sia identico a $y_0$, per cui il valore minimo del $\chi^2$ sarà zero in corrispondenza di un dato $p_0$. Supponiamo ora di ripetere la misura e di trovare un valore $y_0 + \sigma_y$; il nuovo valore di $p$ che annulla $\chi^2$ sarà tale per cui

$$f(p_o + \Delta p) = y_0 + \sigma_i$$

Se calcoliamo la prima funzione $\chi^2$ ottenuta in questo nuovo punto otteniamo:

$$\chi^2(p_0 + \Delta p) = \frac{(y_0 - f(p_0 + \Delta p))^2}{\sigma_y^2} =$$

$$\frac{(y_0 - y_0 - \sigma_y)^2}{\sigma_y^2} = 1$$

Cioè la variazione di $p$ in corrispondenza della variazione di un sigma della misura $y_0$ corrisponde ad una variazione di $\chi^2$ pari a 1.
Osserviamo che, nel caso di molti parametri, l' insieme dei $\vec{p}$ per cui $\chi^2$ differisce dal minimo per meno di uno può avere una forma complicata, ed eventualmente potrebbe non trattarsi di una regione connessa. Nel caso la funzione $\chi^2$ intorno al minimo sia assimilabile ad un paraboloide in $M$ dimensioni (dove $M$ è il numero di parametri ovvero la dimensione di $\vec{p}$) tale regione sarà un ellissoide.

### 6.4.4 Soluzione numerica di sistemi di equazioni differenziali

Introduciamo un metodo di integrazione di equazioni differenziali lineari che può essere utile per ricavare le equazioni del moto a partire dalla legge $\vec{F} = m\vec{a}$ nei casi in cui non sia possibile una soluzione analitica.
Consideriamo inizialmente una equazione del tipo

$$x'(t) = a(t)x(t) + b(t) = f(t, x(t))$$

dove le funzioni $a$ e $b$ sono note e $x(t)$ è la funzione (la traiettoria) che vogliamo ricavare. In questo senso diremo che la funzione $f$, intesa come funzione di $t$ e $x$ è nota, anche se non è nota in quanto funzione del solo $t$.
La "condizione al contorno" di questo problema sarà specificata dal valore di $x$ al tempo $t = 0$.
Osserviamo ora che, utilizzando uno sviluppo in serie di Taylor al primo ordine potremo scrivere:

$$x(\Delta t) = x(0) + \Delta t x'(0) + O(\Delta t^2)$$

Siccome $x(0)$ è noto e $x'(0) = f(0, x(0))$ possiamo calcolare $x(\Delta t)$ e quindi iterare la procedura per ottenere la traiettoria ad ogni $t$.
Questo metodo, essendo al primo ordine, è poco preciso; di conseguenza di devono usare valori del passo di campionamento $\Delta t$ molto piccoli per mantenere errori accettabili. Sarebbe ovviamente meglio utilizzare un metodo al secondo ordine, ovvero

$$x(\Delta t) = x(0) + \Delta t x'(0) + \frac{1}{2}\Delta t^2 x''(0) + O(\Delta t^3)$$

Abbiamo però il problema di non saper calcolare la derivata seconda. Possiamo però osservare che, sviluppando al prim' ordine la derivata prima otteniamo:

$$x'\Big(\frac{\Delta t}{2}\Big) = x'(0) + \frac{1}{2}\Delta t x''(0) + O(\Delta t^2)$$

Ricavando il termine $\frac{1}{2}\Delta t x''(0)$ e sostituendo nello sviluppo al secondo ordine otteniamo:

$$x(\Delta t) = x(0) + \Delta t x'(0) + \Delta t x'\Big(\frac{\Delta t}{2}\Big) - \Delta t x'(0) + O(\Delta t^3) =$$

$$x(\Delta t) = x(0) + \Delta t x'\Big(\frac{\Delta t}{2}\Big) + O(\Delta t^3)$$

Di conseguenza, per ottenere una valutazione di $x(\Delta t)$ al secondo ordine dobbiamo conoscere $x(0)$ e $x'(\Delta t/2)$. D'altro canto

$$x'\Big(\frac{\Delta t}{2}\Big) = f\Big(\frac{\Delta t}{2}, x\Big(\frac{\Delta t}{2}\Big)\Big)$$

e

$$x\Big(\frac{\Delta t}{2}\Big) = x(0) + \frac{\Delta t}{2}x'(0) = x(0) + \frac{\Delta t}{2}f(0, x(0))$$

Di conseguenza procederemo così: calcoliamo

$$k_1 = \Delta t f(0, x(0))$$

quindi calcoliamo

$$k_2 = \Delta t f\Big(\frac{\Delta t}{2}, x(0) + \frac{k_1}{2}\Big)$$

ed infine

$$x(\Delta t) = x(0) + k_2 + O(\Delta t^3)$$

A parole potremmo dire che si deve prima calcolare l' incremento della funzione nell' intervallo $\Delta t/2$ con la derivata calcolata in zero; quindi si usa tale incremento per calcolare l' incremento nell' intervallo $\Delta t$ con la derivata calcolata in $\Delta t/2$. Ovviamente anche in questo caso la procedura può essere iterata per ottenere il valore di $x$ ad un $t$ qualsiasi.
Il metodo appena illustrato si chiama metodo di Runge–Kutta di ordine due; più di frequente si adopera quello di ordine quattro, per il quale diamo la sequenza di calcolo:

$$k_1 = \frac{\Delta t}{2} f(0, x(0))$$

$$k_2 = \Delta t f\Big(\frac{\Delta t}{2}, x(0) + \frac{k_1}{2}\Big)$$

$$k_3 = \Delta t f\Big(\frac{\Delta t}{2}, x(0) + \frac{k_2}{2}\Big)$$

$$k_4 = \Delta t f(\Delta t, x(0) + k_3)$$

$$x(\Delta t) = x(0) + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(\Delta t^5)$$

Resta adesso da risolvere un solo problema: come superare il vincolo delle equazioni di primo grado? In realtà la soluzione è semplice; consideriamo

$$x''(t) = a(t)x'(t) + b(t)x(t) + c(t)$$

Questa equazione è equivalente al sistema

$$\begin{cases} x'(t) = y(t) \\ y'(t) = a(t)y(t) + b(t)x(t) + c(t) \end{cases}$$

dove abbiamo introdotto la nuova funzione incognita $y(t)$. Per questo problema serviranno due condizioni iniziali, in particolare i valori di $x(0)$ e $y(0) = x'(0)$. Il sistema può essere risolto con il metodo di Runge–Kutta.