1

Capitolo B82
c++, definizione del linguaggio

Contenuti delle sezioni

48 pagine

B820.01 Questo capitolo è dedicato alla presentazione i termini generali e sistematici della parte del linguaggio di programmazione C++ che abbiamo chiamato C++ ridotto, in sigla c++,.

Con c++ intendiamo un sottoinsieme del linguaggio C++ che, grosso modo, è anche un sottoinsieme del linguaggio C; più precisamente la sua portata è contenuta nella portata di C, ma adotta alcune regole formali di C++ assenti dal linguaggio C.

Come si è detto c++ comprende gli elementi che consentono di implementare, in modo prettamente procedurale gli algoritmi dei tipi che più intervengono nell'*esposizione*, algoritmi riguardanti strutture combinatorie, gestione di testi, matrici e analisi numerica di base.

Esso intende costituire uno strumento di programmazione utilizzabile concretamente e piuttosto facilmente con qualcuno dei molti sistemi software per lo sviluppo del linguaggio C++ attualmente disponibili su tutti i tipi di computers.

I programmi e i frammenti di programma presentati possono essere ripresi per essere esaminati e riutilizzati, tenendo conto che sono stati collaudati con il sistema di sviluppo MicroSoft Visual Studio.

B820.02 Questo capitolo, il secondo sulla introduzione al linguaggio c++, inizia [:a] introducendo i numeri di solito chiamati "numeri reali", ma che in realtà costituiscono un sottoinsieme finito di $\mathbb{Q}$ e che qui chiamiamo **numeri real-fp**.

Questi consentono di servirsi delle espressioni in grado di controllare nel modo più diretto le costruzioni computazionali della geometria, dell'analisi infinitesimale, della fisica e di tutte le loro applicazioni quantitative.
Queste costruzioni per la maggior parte portano a risultati che al livello idealizzato della matematica sono numeri reali, mentre al livello della effettività operativa si devono servire dei numeri real-fp.

I problemi che conducono a risultati per i quali bastano i numeri interi sono relativamente pochi e in genere riguardano problemi con finalità strettamente organizzative (ad esempoio quelle dalla ricerca operativa) che si trovano nei più vasti problemi applicativi con ruoli di sottoproblemi circoscritti.

Successivamente, in :b si affronta un importante ampliamento della la gamma dei dati controllabili con il linguaggio C++ introducendo gli arrays di una e di più dimensioni, entità che aprono rilevanti possibilità per il controllo delle grandi quantità di dati e per l'automatizzazione delle elaborazioni dei processi entro ambienti modellabili con la geometria cartesiana.

La sezione :d riguarda l'introduzione dei sottoprogrammi e la organizzazione modulare dei programmi, un altro fondamentale ampliamento della portata della programmazione che porta alla scalabilità quasi illimitata delle elaborazioni con automatismi.

Il primo genere di sottoprogrammi che si esamina con una certa ampiezza [:e] è quello che si rivolge alle manovre per l'entrata e l'uscita dei dati.

L'ultima parte consiste nella discussione di vari piccoli programmi che risolvono problemi circoscritti che si possono incontrare in vari contesti e quindi si possono considerare di rilevante importanza propedeutica.

B82 a. lessico del linguaggio

B82a.01 Chiamiamo **tokens** o lessemi le componenti formali elementari di un testo sorgente nel linguaggio C++.
Quindi un programma sorgente si può considerare una sequenza di tokens.

Tra le unità lessicali distinguiamo le seguenti categorie lessicali:

parole riservate: brevi parole fissate dal linguaggio che fanno da demarcatori del testo per organizzarlo nel suo complesso e nelle sue porzioni aventi compiti specifici;

identificatori: stringhe scelte dal programmatore per rappresentare nel sorgente i dati da elaborare e altre informazioni ausiliarie;

commenti: scritture che non hanno effetti interpretativi o operativi, ma servono a rendere più comprensibile la lettura del testo;

scritture di costanti: scritture che rappresentano dati fissi, non modificabili nel corso di una elaborazione;

operatori e segni di punteggiatura: stringhe prefissate molto corte; gli operatori denotano azioni da effettuare su operandi loro contigui; i segni di punteggiatura o punctuators servono a demarcare e a strutturare il testo sorgente.

B82a.02 Le parole riservate di C++, dette anche **keywords**, sono stringhe formate da lettere minuscole che hanno ruoli ben definiti nel linguaggio.
Per evitare ambiguità si impone che le altre stringhe a disposizione del programmatore, gli identificatori, non possono coincidere con alcuna delle parole riservate.
Il loro elenco si trova in B78 e i rispettivi ruoli vengono spiegati nel corso della esposizione della sintassi.
Qui ci limitiamo a presentarne alcune:

`if`, precede una clausola che può essere rispettata o meno;
`int`, serve a definire variabili a valori intere;
`char`, drfinisce variabil aventi come valori caratteri tendenzialmente leggibili;
`do`, ordina una certa azione;
`goto`, comporta di passare da un punto del codice sorgente a u altro;
`new`, richiede nuova memoria.

B82a.03
I commenti in C++ sono scritture che il programmatore può inserire liberamente nel testo sorgente e che hanno lo scopo rendere il contesto più leggibile fornendo informazioni sulle linee di codice accanto alle quali sono collocate.

Vi sono due tipi di commenti: commenti di fine linea e commenti tra delimitatori.

I primi iniziano con due barre "`//`" e si concludono con la fine della linea corrente del testo.

I secondi possono occupare più linee, interamente o in parte, iniziano con "`/*`" e finiscono con "`*/`".

Esempi di commenti si trovano in quasi tutti gli esempi di codice sorgente presentati.

Nel corso dell'intero ciclo di vita di un programma il programmatore può avvalersi dei commenti in vari modi.

Di fronte a errori sintattici che si possono trovare in più punti del testo non facili da individuare il programmatore può effettuare prove parziali rendendo invisibili al compilatore alcuni frammenti del

testo, per potersi concentrare su altri; dopo avere corretti o trovati legali i frammenti testati si può concentrare su altri precedentemente nascosti.

I commenti possono essere usati per documentare un lavoro di programmazione di lunga durata e per ricordare criteri adottati in fasi di lavoro precedenti. Questo serve in particolare:
per programmi che richiedono il lavoro di più persone;
per programmi che si prevede debbano essere aggiornati in tempi successivi;
per programmi che sono portati avanti con parti lasciate incomplete per le quali si sono adottate soluzioni parziali facili e provvisorie, con il proposito di migliorarle in seguito; in particolare le migliorie vengono precisate dopo opportune sperimentazioni con dati di prima prova giudicati poco rischiosi e con conseguenze facili da riscontrare.

B82a.04 I dati in un programma sono le entità informative che sono oggetto delle manipolazioni. Possono essere elaborati dati singoli e dati collettivi; tra questi si distinguono gli schieramenti di dati o arrays e le enumerazioni; agli arrays si possono attribuire una o più dimensioni.

Si distinguono diversi tipi di dati; il tipo di un dato determina quale cella-mm si assegna ad ogni dato singolo, quali valori, costanti o variabili può assumere e in quali modi può essere utilizzato o ottenuto dalle operazioni (di circuiti operativi) che possono agire su di esso.

Si distinguono i dati costanti e i dati variabili nel corso di ogni elaborazione; in un programma sia le costatnti che le variabili devono essere dichiarate, ossia introdotte formalmente con frasi specifiche.
In conseguenza di una dichiarazione il compilatore assegna alla costante o alla variabile una porzione di memoria che corrispond alle esigenze del suo tipo.

C++ supporta, ossia consente di utilizzare, molti tipi di dati e tra questi il programmatore può scegliere per ogni dato che entra nella sua applicazione il tipo più appropriato.
I dati fanno parte di diversi tipi tipi dei dati

Consideriamo il seguente programma

```
#include <iostream>
using namespace std;
int main() {// serve a ricordare alcune costanti matematiche
    // introduce tre variabili cui assegna precisi valori
    float pi = 3.14;
    float sqrtofpi = 1.7, pipw2 =9,4;
    cout << "pi = " pi, endl;
    cout << "sqrtofpi = sqrtofpi = " sqrtofpi, "pipwr2 = " pipwr2, endl;
    return 0; }
```

B82a.05 Nel C++ si distinguono tre categorie di tipi di dati:

tipi di dati basici : sono tipi di dati fissati nel linguaggio e riguardano dati singoli, ossia portatori di valori singoli;

Si hanno sei sottocategorie caratterizzate dalle keywords `int, float, double, char, bool, void`.

tipi di dati derivati: fissati nel linguaggio, si possono ricondurre ai vari dati basici.

Si hanno quattro sottocategorie caratterizzate dalle keywords `array, pointer, reference, function`

tipi di dati definiti dall'utente: decisi dal programmatore per avere facilitata la rappresentazione nel testo sorgente delle sue richieste operative.

Sono previste le sottocategorie caratterizzate dalle keywords `class, struct, union, typedef, using`

B82a.06 Il tipo di dati character viene usato per registrere in una cella-8b, in 8 bits, un carattere ASCII che nel sorgente viene scritto tra due segni "'".
Un esempio `char tomo = 'G'`

B82a.07 Il tipo di dati integer riguarda le variabili che possono registrare numeri interi in celle-4B, in 32 bits; più precisamente possono essere trattati gli interi da $-2\,147\,483\,648$ a $2\,147\,483\,647$.
I valori possono essere forniti da scritture binarie, ottali, decimali ed esadecimali.
Esempi `int ini = 12; trm = 0x4d;  int length = o326; altz = b1011011;`

B82a.08 Il tipo di dati boolean viene usato per trattare i valori di verità `true` e `false`, corrispondenti a 1 e 0, registrabili in celle– da 1 byte.
Esempi: `bool valid = true;`
in conseguenza della frase imperativa `cout << valid ;` si ha l'emissione sulla console del valore 1.

A ogni variabile booleana si può assegnare un valore numerico che porta a false se vale zero, porta a true in ogni altro caso; anche una costante carattere si può assegnare a una variabile booleana e fornisce false solo NUL = b00000000.

```
// esempio sui dati del tipo booleano
#include <iostream>
using namespace std;
int main()
{
int x1 = 10, x2 = 20, m = 2;
bool b1, b2;
b1 = x1 == x2; // fornisce false
b2 = x1 < x2; // vale true
cout << "b1 = " << b1 << "\n";
cout << "b2 = " << b2 << "\n";
bool b3 = true;
if (b3) cout << "OK" << "\n";
else cout << "NO" << "\n";
int x3 = false + 5 * m - b3;
cout << x3;
return 0; }
```

Uscita

```
b1 = 0
b2 = 1
OK
9
```

B82a.09 Il tipo di dati floating point, numero in virgola mobile, riguarda numeri (razionali) della collezione dei valori registrabili in celle-4B, in 4 bytes; i valori trattabili devono appartenere all'intevallo

tra `1.2E-38` e `3.E+38`; a rigore questo vale per i sistemi che operano prevalentemente su 64 bits, ossia con parallelismo interno da 64 bits.
Esempi `float pesoinkg = 63.5:`

Il tipo di dati double riguarda numeri (razionali) della collezione dei valori registrabili in celle-8B, in 128 bits (questo è garantito nei sistemi con parallelismo a a 64 bits).
Questo tipo di dati, evidentemente, consente di lavorare con numeri molto più precisi e con valori assoluti molto maggiori di quelli consentiti dal tipo di dati `float`. Possono essere trattati valori circa da `1.7e-308` a `1.7e+308` (questo sui sistemi a 64 bits).
Esempio `double pi = 3.1415926535;`

B82a.10 Il tipo di dati void serve per stabilire che alcune functions non forniscono alcun dato di ritorno.

Esempio
```
#include <iostream >
using namespace std;
// serve a ricordare alcune caratteristiche degli elettroni
void elettrone() {
    float elMassa = 9.1093837139e-31; // massa
    float elCarica = -1.602176634e-19; // carica
    float elMoMagn = -9.2847646917e-24:  // momento magnetico
    cout << "elettrone - massa " << elMassa << "kg" << endl;
    cout << " carica " << elCarica << "C" << endl;
    cout << " momento magnetico " << elMoMagn << "J/T" << endl;
    }
int main() {
    elettrone();
    return 0;
    }
```

B82a.11 I data type modifiers sono keywords che consentono di utilizzare varianti dei tipi di dati primitivi e quini dati che hanno intrvalli di valori diversi e prcisioni diverse.

I dati di un tipo variante sono introdotti premettendo la keyword del modificatore alla keyword del tipo da modificare.

C++ mette a disposizione 4 type modifiers associati, rispettivamente, alle keywords `short, long, signed, unsigned`.

I modificatore `signed` e `unsigned` si possono applicare ai tipi di dati integer e character.
I tipi integer, per default sono signed, quindi questo modificatore non è indispensabile, ma può servire per chiarire situazioni particolari.
Il modificatore `unsigned` è invece necessario quando si voglioni trattare interi non negativi che possono avere valori fino a $2^{32} - 1$.
Entrambi riguardano assegnazioni di celle-4B.

Le dichiarazioni che iniziano con `signed char` e `unsigned char` introducono dati con valori, rispetivamente tra -64 e +63, e tra 0 e 127; entrambi riguardano assegnazioni di celle-8b; anche `signed char` non è indispensabile ma può essere utile per la leggibilità del sorgente.

B82a.12 I modificatori `short` e `long` hanno l'effetto, rispettivamente, di dimezzarre e di raddoppiare l'ampiezza delle celle-mm dedicate a dati numerici.

La keyword `short` si applica solo ai dati di tipo integer e consente di trattare numeri interi, signe o unsigned, ai quali sono dedicati 16 bits.

La keyword `long` si applica ai dati del tipo integer e del tipo double.
Con dichiarazioni che iniziano con `long int` si introducono interi che occupano 8 bytes e lo stesso effetto si ottiene con dichiarazioni che iniziano con la scrittura (enfatica) `long long int` o con la più concisa `long`.
Con dichiarazioni che iniziano con `long double` si rendono disponibili numeri floating point che occupano 16 bytes.

B82a.13 Consideriamo il programma
```
#include <iostream>
using namespace std;
int main() {// ricorda le estensioni dei diversi tipi di dati
    cout << "Size of int:  " << sizeof(int) << " bytes" << endl;
    cout << "Size of char:  " << sizeof(char) << " byte" << endl;
    cout << "Size of float:  " << sizeof(float) << " bytes" << endl;
    cout << "Size of double:  " << sizeof(double) << " bytes";
    return 0;
}
```

Il suo richiamo provoca la emissione delle linee seguenti

```
Size of int:  4 bytes
Size of char:  1 byte
Size of float:  4 bytes
Size of double:  8 bytes
```

B82a.14 In ogni linguaggio di programmazione compaiono scritture che esprimono informazioni che non cambiano nel corso delle esecuzioni.
Nella definizione del linguaggio C++ si incontra il termine literal, che consideriamo sinonimo di scrittura di costante.
Ci sono di quattro tipi di literals per i dati integer: decimal-literal, octal-literal, hex-literal e binary-literal.

Scritture decimali di numeri interi positivi: cifra decimale positiva seguita da nessuna o più cifre decimali.

Scritture ottali di interi non negativi: 0 seguito da nessuna o più cifre ottali; esempi 036, 0343, 01703.

Scritture esadecimali di numeri nonnegativi: 0x o 0X seguito da una o più cifre esadecimali, cifre decimali e a = A, b=B, c=C,d=D, e=E, f=F; esempi 0x23d, 0XFF, 0X4fa56.

Scritture binarie di interi nonnegativi: 0b o 0B seguiti da una o più cifre binarie: esempi 0b1001, 0B00111100.

Le scritture di interi possono essere seguite da suffissi che precisano il tipo che si intende attribuire all'intero rappresentato.
Per il tipo `int` non è richiesto alcun suffisso, in quanto si tratta della scelta per default.
Per il tipo `unsigned int` il suffisso è u oppure U.
Per il tipo `long int` il suffisso è l oppure L.

Per il tipo `unsigned long int` il suffisso è ul oppure UL.
Per il tipo `long long int` il suffisso è ll oppure LL.
Per il tipo `unsigned long long int` il suffisso è ull oppure ULL.

B82a.15 Si hanno due forme di floating-literals, la forma decimale e la esponenziale.

La forma decimale può consistere nella parte intera seguita da punto, nel punto seguito dalla parte cecimale, e nella parte intera seguita da punto e dalla parte decimale. Esempi 255., .0034, 3900.3565656.

La forma esponenziale richiede una parte significativa che segue la forma decimale seguita da una lettera e o da una E e seguita da un intero decimale positivo o negativo. Inoltre si aggiunge il suffisso l o L per chiedere la assegnazione al tipo `double float`. Esempi: 1.215e-10L, 404.4E-12, 1,423E+23.

B82a.16 I character-literals riguardano singoli caratteri. Si possono tuttavia avere arrays di caratteri. nel caso di caratteri visualizzabili la scrittura si riduce al trigramma formato dal carattere preceduto e seguito da apice singolo, "'": esempi 'W', 'w'.
Per altri caratteri si deve ricorrre al meccanismo chiamato **escape sequence** [B86c].

B82a.17 Gli string-literals sono simili ai character-literals, ma invece che fornire singoli caratteri esprimono sequenze di caratteri e si possono avvicinare agli arrays monodimensionali di caratteri.

Le scritture di stringhe sono sequenze di indicatori di caratteri, caratteri visualizzabili ed escape sequences, delimitate da due occorrenze del carattere doppio apice, "\"".

Una lunga sequenza di caratteri può essere conveniente gestirla con successivi string-literals ciascuno dei quali possa essere contenuto in una linea stampata o presentata su schermo video.

B82 b. numeri ed espressioni reali

B82b.01 Un importante tipo di dati è quello che fornisce la possibilità di servirsi di una gamma, necessariamente finita ma estesa, di numeri reali che qui chiameremo tipo dei **numeri real-fp**.

In effetti questo insieme di configurazioni binarie rappresenta fedelmente un certo insieme di numeri razionali che stiamo per individuare con precisione, ma attraverso considerazioni assai minuziose e dettagliate.

Occorre anche dire che spesso questi numeri vengono chiamati sbrigativamente "numeri reali" e vengono proposti senza vere definizioni, ma basandosi sul fatto che funzionano piuttosto bene per approssimare i numeri reali della matematica che servono per risolvere una significativa gamma di problemi di interesse pratico.
Questo modo di procedere, giustificato per gran parte delle attività computazionali mediamente impegnative, lascia margini di attendibilità che nei progetti di elevata responsabilità devono essere esaminati con studi specificie e risolti con tecniche che particolari.

B82b.02 Per introdurre i numeri reali-fp conviene partire dai literals che consentono di esprimerli nel testo del programma e dalle configurazioni di bits che costituiscono le loro conseguenti implementazioni nelle celle-mm e che li rendono effettivamente manipolabili.

Un numero real-fp positivo non troppo grande, né troppo piccolo, viene espresso con la notazione che presenta la sua parte intera, il separatore "." e la sua parte decimale.
Esempi `15.085 , 0.000253 . 250000000`
Questa notazione permette di trattare agevolmente numeri il cui ordine di grandezza va dal miliardesimo al miliardo.
In molti calcoli scientifici servono anche numeri ben inferiori al miliardesimo e ben superiori al miliardo.
Per questi si rende necassaria la notazione esponenziale che alla parte intera, al punto e alla parte decimale fa seguire un fattore costituito da una potenza positiva o negativa di 10.

La massa dell'elettrone si conviene che sia $9,109\,383\,701\,5 \cdot 10^{-31}$ kg.
La costante di Avogadro, il numero delle particalle costituenti una mole di sostanza, è definita come $N_A := 6,022\,140\,76 \cdot 10^{23}$ mol^{-1}

B82 c. arrays e geometria cartesiana discreta

B82c.01 arrays monodimensionali

B82c.02

B82c.03 Per rappresentare una grande varietà di situazioni servono gli **arrays a due indici**, schieramenti strutture di dati che consentono di implementare le matrici, entità matematiche con una grande quantità e varietà di applicazioni.

Nel linguaggio C gli arrays a due indici si possono considerare arrays a un indice le cui componenti sono a loro volta degli arrays a un indice, allineamenti di dati tutti costituiti da uno stesso numero di componenti elementari.

Volendo disporre nella sottomatrice 4×4 di una matrice 10×10 dei primi coefficienti binomiali simmetrici [??] si possono utilizzare le seguenti frasi (la prima dichiarativa le successive di assegnazione)

```
int cbs[10][10];
cbs[0][0]=1; cbs[0][1]=1; cbs[0][2]=1; cbs[0][3]=1;
cbs[1][0]=1; cbs[1][1]=2; cbs[1][2]=3; cbs[1][3]=4;
cbs[2][0]=1; cbs[2][1]=3; cbs[2][2]=6; cbs[2][3]=10;
cbs[3][0]=1; cbs[3][1]=4; cbs[3][2]=10; cbs[3][3]=20;
```

Se in un programma è sufficiente disporre di una matrice 4×4, basta la seguente frase di inizializzazione

```
int cbs[4][4] = {
                    {1, 1, 1, 1}
                    {1, 2, 3, 4}
                    {1, 3, 6, 10}
                    {1, 4, 10, 20}
                }
```

A questo arrays si dedicano 16 celle-mm per interi normali, celle-16B.
In questa sequenza di celle si devono distinguere 4 sottosequenze, ciascuna costituita da 4 celle: la prima sottosequenza è dedicata alla prima riga della matrice, accessibile per intero scrivendo `cbs[0]`, la seconda alla seconda riga accessibile mediante la scrittura `cbs[1]` e così via.

B82c.04 Talora servono arrays con 3 o più indici: con arrays a 3 indici si possono trattare grandezze fisiche attribuibili a una griglia spaziale, ad esempio le temperature in un certo numero di punti all'interno di una caldaia a forma di cuboide. L'organizzazione in memoria di tali complessi di dati viene effettuata generalizzando il procedimento precedente.
Un array a tre dimensioni potrebbe essere introdotto con una dichiarazione come la

```
int valspaz[5][3][7];
```

la quale comporta la predisposizione di un array con 5 componenti ciascuna delle quali è un array bidimensionale di profilo 3×7.

B82 d. puntatori, operatori ed espressioni

B82d.01 Pointers are symbolic representations of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. Iterating over elements in arrays or other data structures is one of the main use of pointers.
The address of the variable you are working with is assigned to the pointer variable that points to the same data type (such as an int or string).
How to use a pointer?
Define a pointer variable Assigning the address of a variable to a pointer using the unary operator (') which returns the address of that variable. Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.
The reason we associate data type with a pointer is that it knows how many bytes the data is stored in. When we increment a pointer, we increase the pointer by the size of the data type to which it points.

B82d.02

```
// C++ program to illustrate Pointers
#include <bits/stdc++.h>
using namespace std;
void es_pointer_1()
    {
    int var = 20;
    // dichiara una variabile del tipo pointer
    int* ptr;
    // si tenga presente che ptr e var devono appartenere allo stesso tipo
    ptr = &var;
    // assegna al puntatore l'indirizzo della variabile var
    cout << "Value at ptr = " << ptr << "\n";
    cout << "Value at var = " << var << "\n";
    cout << "Value at *ptr = " << *ptr << "\n";
    }
// Driver program, programma che serve a provare quanto sopra
int main()
{
    es_pointer_1();
    return 0;
}
```

Si ottiene una emissione con la seguente forma

Value at ptr = 0x7ffe454c08cc
Value at var = 20
Value at *ptr = 20

B82 e. functions e loro richiami

B82e.01 Per affrontare problemi di interesse pratico si devono scrivere programmi di molte migliaia e anche di molti milioni di istruzioni.

In effetti lo sviluppo delle apparecchiature elettroniche programmabili iniziato a metà del secolo scorso ha portato alla messa a punto di procedure molto elaborate e la tecnologia della programmazione costituisce da decenni uno dei fattori industriali chiave per lo sviluppo dell'economia.

Le attività concernenti la produzione di programmi sono ormai sviluppate con criteri industriali e in esse si rende necessaria una sistematica suddivisione dei compiti; questa suddivisione riguarda sia le azioni svolte dai programmi, sia le attività delle persone che i programmi progettano, redigono, collaudano, fanno evolvere e analizzano criticamente.

Per la suddivisione delle azioni controllate dai programmi, quali che siano i linguaggi di programmazione utilizzati, assume grande importanza la organizzazione dei sottoprogrammi.

B82e.02 Un sottoprogramma si può definire come una porzione di un programma chiaramente delimitata, sia al livello del testo che costituisce la sua formalizzazione, che al livello semantico del suo significato operativo, ossia del suo compito di effettuare un servizio specifico in risposta di ciascuna richiesta che gli viene rivolta.

Ciascuna richiesta a un sottoprogramma ***S***, in linea di massima, contiene specificazioni sulle particolari prestazioni che gli vengono richieste.

Le prestazioni richieste a un sottoprogramma devono essere adeguate alla portata operativa consentita al sottoprogramma stesso.
Tuttavia i sottoprogrammi in generale, in accordo con la portata complessiva della programmazione e con l'efficienza e la versatilità delle risorse oggi disponibili, sono in grado di effettuare una vastissima varietà di servizi.
Ed in effetti tutti i programmi che risolvono problemi applicativi di interesse concreto demandano la gran parte delle loro prestazioni a un numero rilevante di sottoprogrammi.
Molti programmi mediamente elaborati negli obiettivi e nelle prestazioni richiedono l'intervento di molte migliaia di sottoprogrammi con compiti e caratteristiche anche molto differenziate.

B82e.03 A questo punto facciamo riferimento ad alcuni esempi che riguardano sottoprogrammi dei tipi che incontreremo più spesso nel seguito.

Ad un programma di calcolo tecnico-scientifico sono necessari vari sottoprogrammi incaricati di effettuare calcoli specifici:
calcoli di funzioni matematiche che avremo modo di incontrare più volte (radici quadrate, funzioni esponenziali e trigonometriche, altre funzioni speciali, trasformazioni di coordinate, conversioni, ...);
soluzioni di sistemi di equazioni lineari e loro varianti; determinazione di zeri di polinomi e di altre funzioni di variabile reale; elaborazioni di figure geometriche, di processi meccanici, di evoluzioni finanaziarie, di analisi statistiche, ...;
manovre su strutture combinatorie e in particolare su grafi variamente arricchiti (per esempio grafi in grado di rappresentare reti di trasporto, evoluzioni di automi, schemi organizzativi, ... ;
generazioni di grafici e di animazioni che consentono di presentare i risultati di elaborazioni tendenzialmente complessi e numerosi attraverso immagini la cui comprensione è aiutata dalla intuizione visiva; in particolare possono esser d'aiuto grafici che forniscono dati di sintesi statistiche riguardanti

i parametri di sintesi relativi a popolazioni di migliaia o milioni di esemplari; possono anche essere molto efficaci le animazioni riguardanti le evoluzioni di processi materiali o di comportamenti umani.

B82e.04 Alle operazioni di ingresso e uscita vengono dedicati tanti tipi di sottoprogrammi.
Molti di questi si incaricano del controllo dei dispositivi hardware delle molteplici apparecchiature che si possono connettere a una macchina programmabile.
Oggi si possono considerare periferiche tradizionali i terminali video con tastiera e mouse, le stampanti, i plotters, i dischi, i nastri, le memorie flash.
A queste in tempi più recenti si sono aggiunti le apparecchiature per l'immissione e l'emissione dei suoni, le macchine fotografiche, le videocamere, monitors televisivi e i collegamenti con reti della telefonia e reti di computers.
Tutte queste apparecchiature richiedono routines agenti sull'hardware che evidentemente dipendono da una varietà di caratteristiche tecniche, richiedono programmatori con competenze deinite e approfondite e possono anche richiedere linguaggi specifici.

Venendo a prestazioni più vicine alle applicazioni e formulabili con linguaggi come C++, per le operazioni di ingresso dei dati ricordiamo: lettura di stringhe e loro codifica in valori interni, tipicamente lettura di scritture che seguono notazioni standard di numeri interi o razionali; lettura di denominazioni leggibili e loro trasformazioni in codifiche convenzionali che rendono più agevole la determinazione di loro caratteristiche (ad esempio le codifiche con una lettera e 3 cifre dei comuni italiani).
Ancora più variegate sono le prestazioni di scrittura: operazioni di decodifica in chiari speculari delle accennate operazioni di codifica; presentazione di grafici che raffigurano dati numerici (istogrammi, areogrammi, ...); arricchimento dei risultati con dati di sintesi (medie, varianze, totali parziali, ...); generazione di animazioni.

Assimilabili ai sottoprogrammi di entrata e uscita sono i sottoprogrammi che si incaricano di operazioni di inserimento e di reperimento di dati entro strutture informative e archivi.
Molti altri sottoprogrammi si possono incaricare di conversioni e di transcodifiche relative a formati e a strutture di dati definiti per rendere calcolabili determinati modelli e per trattare più agevolmente particolari manovre; in particolare vi sono programmi che organizzano le "visite" di strutture informative per analizzarle o individuarne elementi di rilievo (minimi, massimi, estremali, ottimali).
Per concludere, non possiamo non ricordare i sottoprogrammi che si occupano della gestione di situazioni erronee o anomale, operazioni che nei più recenti linguaggi di programmazione hanno ottenuto specifici riconoscimenti anche al livello della sintassi.

B82e.05 Nel linguggio c++ i sottoprogrammi assumono la forma di quelle che chiameremo **functions**, entità con alcune caratteristiche delle funzioni della matematica, ma per le quali useremo il termine inglese per sottolinearne le differenze.

Una function è una sezione sintatticamente ben definita del testo sorgente di un programma che può essere richiamata per eseguire manovre specifiche che è opportuno siano ben definite. La forma di una function prevede una intestazione e un corpo e si può schematizzare come segue.

tipo-della-function *identificatore-della-function* (*lista-degli-argomenti*)
{
corpo-della-function
}

Ogni esecuzione di una function può produrre un risultato primario che risulta associato al suo richiamo ed è disponibile nell'espressione che contiene il richiamo stesso.

Nelle pagine che seguono incontreremo prevalentemente functions che producono un risultato primario intero. Sono comunque importanti le functions che producono un risultato primario dei tipi `double` e `char` e quelle che producono un indirizzo di dati singoli o multipli dei vari tipi.
A una fuction si attribuisce il tipo del dato che produce e che va dichiarato esplicitamente nel programma.
Occorre tuttavia segnalare che alcune functions non producono un risultato esplicito e ad esse va attribuite lo specifico tipo corrispondente alla parola riservata `void` .

B82e.06 Gli identificatori delle functions seguono le stesse regole lessicali degli identificatori delle variabili e come per questi si pone il problema della non omonimia.

È opportuno che l'identificatore di ogni function sia scelto in modo da evidenziare il suo compito; questa scelta, in genere piuttosto delicata, va fatta cercando di avere chiari i suoi possibili utilizzi e quali altre functions saranno utilizzate in collegamento o in alternativa a essa.

La lista degli argomenti di una function può contenere uno o più argomenti o anche nessun argomento. Di ogni argomento si deve esplicitare il tipo e un identificatore può essere accompagnato da un modificatore.

Il corpo di una function ha forma simile a un blocco di istruzioni. In genere inizia con dichiarazioni di variabili locali la cui visibilità è limitata al solo corpo della function.

Successivamente può presentare una qualsiasi composizione di sequenze di comandi e di blocchi per selezioni e per iterazioni. È opportuno che queste composizioni siano ben strutturate.

Nel corpo di una function deve comparire almeno una istruzione `return` accompagnata da un argomento costituito da un'espressione che fornisce un risultato avente lo stesso tipo della function.
L'esecuzione di un comando `return` fornisce come risultato primario di ogni richiamo della function il valore attualmente calcolato per l'espressione.

B82e.07 Ogni richiamo alla function è costituito dal suo identificatore seguito dalla cosiddetta **sequenza degli argomenti attuali**, espressioni costituite da operandi valutabili all'atto dell'esecuzione del richiamo le quali dovranno essere in grado di fornire un valore per ciascuno degli argomenti formali della intestazione della function richiamata.

Una function per essere utilizzabile in un modulo di programma deve essere dichiarata nel testo di tale modulo.
Una dichiarazione di una function può essere una semplificazione della sua intestazione nella quale ogni argomento viene sostituito dalla semplice dichiarazione del suo tipo.
Questo enunciato semplifica il lavoro al traduttore del testo sorgente del programma nel suo equivalente utilizzabile per l'esecuzione (dopo la manovra di linkaggio dei vari moduli che possono costituire un programma che vedremo più oltre), in quanto annuncia le caratteristiche essenziali, cioè i tipi, dei suoi argomenti e del suo risultato primario.
Alla dichiarazione devono adeguarsi l'intestazione della function e tutti gli enunciati nei quali viene richiamta.

In una dichiarazione di function la specificazione di un argomento può presentare, oltre al suo tipo, un identificatore fittizio che conviene scegliere in modo di fornire a chi legge il testo del modulo, un'indicazione del ruolo dell'argomento stesso, ovvero del suo significato operativo.
In tal modo una dichiarazione di function riesce a documentare in forma concisa il suo significato.

B82e.08 Anche il modulo principale di ogni programma viene strutturato come una function. Tuttavia l'identificatore di tale function è prefissato, deve essere la parola riservata `main`.

Anche il modulo `main` può avere un tipo, ma spesso si riduce a `void`.

La forma dell'intestazione di un modulo principale è la seguente:

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

Si prevede che la richiesta dell'esecuzione di un programma sia ottenuta digitando il nome del file contenente il cosiddetto **programma oggetto**, ricavato dal testo sorgente dal linker, eventualmente seguito da un determinato numero di stringhe che sono in grado di indirizzare la sua esecuzione.
Il numero di tali stringhe è fornito dalla variabile identificata da `argc` (sta per "argument count") ed esse sono reperibili in sequenze di bytes le cui `argc` posizioni iniziali sono ottenibili dall'array il cui identificatore è `argv` (sta per "argument values").
Questo meccanismo e il ruolo di "`**`" verrà precisato più oltre.

B82e.09 Riprendiamo gli effetti che può avere l'esecuzione di un richiamo di function.
Per questo occorre distinguere tra gli argomenti della function quali vengono "richiamati per valore"' e quali sono "richiamati per indirizzo"'.

All'inizio dell'esecuzione di un richiamo di function ciascuno degli argomenti fornisce un valore che viene fornito alla function.
Ciascuno degli argomenti costituiti da espressioni da valutare o da variabili fornisce un valore che viene riprodotto nella corrispondente variabile fittizia.
A sua volta ciascuno degli argomenti che individuano un indirizzo porta alla riproduzione di tale indirizzo in una variabile interna alla function.
La riproduzione di un argomento (richiamato per valore) costituito da una variabile assicura che la function non sia in grado di modificare il valore di questa variabile nel modulo richiamante. Se questo accadesse si avrebbe una modifica nel modulo richiamante non evidenziata dal suo testo e quindi poco controllabile e con il rischio di un fraintendimento.

La riproduzione degli indirizzi lascia invece aperta la possibilità di modificare i contenuti delle celle di memoria associate agli indirizzi ed accessibili sia al modulo richiamante che alla function.
Queste celle devono essere utilizzate come indirizzi e quindi le modifiche che la function può effettuare sono maggiormente evidenti e controllabili dai programmatori del modulo richiamante e della function delle modifiche che si potrebbero effettuare all'interno di una function che potesse agire direttamente sulle celle nelle quali sono registrati i valori degli argomenti.

Tutta questa organizzazione è indubbiamente elaborata, ma ha lo scopo di rendere meno probabili le modifiche di dati del modulo chiamante da parte di istruzioni formulate nel testo sorgente della function.

La function dopo la riproduzione degli argomenti, procede a eseguire le manovre formulate nel blocco costituente il corpo della function. Queste manovre potrebbero essere molto complesse, anche perché molte functions che prestano servizi complessi in genere richiamano altre functions e questa organizzazione di functions da collocare a diversi livelli può essere assai articolata.
Le manovre implicate da un richiamo di function forniscono il servizio che costituisce la ragion d'essere della function stessa.
Va sottolineato che una function può servirsi delle componenti di più arrays e può modificarle, ma solo attraverso gli indirizzi iniziali di questi schieramenti di dati.

B82e.10 Consideriamo il programmatore P_M di un modulo **M** che richiama una function **F** curata da un secondo programmatore P_F (caso simile a quello in cui si tratta dello stesso programmatore $P_M = P_F$ il quale prima ha scritto **G** e solo dopo parecchio tempo deve curarsi di **M**).

Per servirsi correttamente della **F** in **M**, è necessario che gli effetti sui suoi argomenti a P_M risultino definiti con precisione e completezza.
Viceversa non è necessario che P_M abbia presenti i dettagli delle operazioni previste in **F**.
Anzi, in linea di massima è opportuno che questi effetti siano documentati solo nel testo sorgente della function o in un documento a essa associato e non compaiano nel testo del modulo chiamante in quanto potrebbero complicarne la comprensione.
Infatti ogni function è tanto più utile quanto più agevolmente il programmatore responsabile di qualche modulo che la richiama riesca a tenere sotto controllo i suoi richiami.
Questo programmatore P_M deve potersi documentare in modo completo e preciso, ma limitatamente agli effetti per l'esterno della function e deve riuscire ad evitare di addentrarsi nei dettagli delle operazioni che vengono effettuate nel corso delle esecuzioni della function.

Quindi si favorisce una divisione dei compiti tra il programmatore della function e il programmatore delle sue utilizzazioni: per un buon utilizzo delle risorse umane in una attività impegnativa, costosa e carica di responsabilità come la programmazione è bene che i dettagli delle operazioni effettuate dalla function possano rimanere nascosti ai programmatori che devono solo utilizzarle attraverso i loro richiami.

Questa ripartizione dei compiti si realizza con il cosiddetto **incapsulamento** delle manovre di una function.
Va ribadita l'importanza della documentazione verso l'esterno delle prestazione delle functions: questa deve essere precisa, completa e rapidamente leggibile, soprattutto quando di una function complessa si vogliano utilizzare solo prestazioni circoscritte.
Tenuto conto dell'importanza in quanto prodotti industriali delle functions in un linguaggio di programmazione di largo uso, dovrebbe essere chiaro che nella pianificazione della messa a punto di un programma si devono dedicare molte attenzioni alla definizione e alla scelta di ciascuna una cosiddetta library delle functions utilizzabili, alla documentazione delle loro prestazioni e alle prove sistematiche che possono garantire la loro correttezza e la loro adeguatezza.

B82e.11 Aggiungiamo alcune prestazioni perticolari dei richiami di function.

Si possono usare anche frasi esecutive della forma

espressione **;**

L'effetto di questa frase è la valutazione dell'espressione, processo che può comportare i cosiddetti **effetti collaterali** (*side effects*), cioè modifiche di variabili che possono avere un ruolo importante nelle elaborazioni successive.

Quando un richiamo di function viene seguito dal solo "**;**" il risultato della valutazione della function, se effettivamente costruito, resta inutilizzato; avranno seguito serviti solo gli effetti collaterali che nel modulo chiamante non compaiono e possono restare nascosti a chi esamina il programma.

Gli effetti collaterali dei richiami di function possono essere di vari generi.
Possono essere eseguite varie manovre sopra unità periferiche come riavvolgimenti di nastri, aperture e chiusure di files, letture di rilevanti gruppi di dati e scritture di complessi di risultati.
Possono essere eseguite manovre rilevanti su variabili nella memoria centrale o disponibili sullo spazio chimato heap.
Possono essere inviati messaggi digitali o segnali d'altro genere con conseguenze sull'esterno che possono portare a reazioni sull'andamento della prosecuzione della esecuzione; in particolare si possono avere interazioni con il Web.

Inoltre si possono organizzare functions senza risultati di ritorno, le cosiddette **non-value returning functions**.

Per questi richiami occorre raccomandare che l'effetto del richiamo sia ben chiarito dal suo programmatore agli operatori interessati ai risultati che possono essere influenzati e ad altri programmatori che fossero incaricati di aggiornare o comunque modificare il modulo contenente queti richiami.

B82 f. files e manovre di immissione ed emissione

B82f.01 Le operazioni di entrata e uscita si possono programmare servendosi di diverse modalità, ovvero utilizzando gruppi di sottoprogrammi facenti parte di diverse librerie. Tra queste si deve distinguere, innanzi tutto, tra quelle introdotte con il linguaggio C (tendenzialmente più elementari) e quelle adottate con il linguaggio C++ (che in genere sono di uso più impegnativo, ma sono più complete e danno maggiori garanzie di sicurezza).

Per usare questi sottoprogrammi si deve provvedere all'inserimento dei comandi iniziali per la predisposizione degli header della forma

`#include <`*specifica di header*`>`

L'header per i tradizionali sottoprogrammi di I/O del linguaggio C ha come specifica `stdio.h`.
Nell'ambito dei sistemi di sviluppo per C++ sono disponibili i seguenti headers:

`iostream.h` riguarda sottoprogrammi per l'entrata da e l'uscita su i cosiddetti streams, periferiche o files che gestiscono sequenze di bytes.

`iomanip.h` fornisce prestazioni di manipolazione dei dati;

`fstream.h` riguarda operazioni per l'emissione su e l'immissione da files su memorie di massa.

B82f.02 Le più semplici operazioni riguardano l'immissione e l'emissione di singoli caratteri attraverso le functions `getchar` e `putchar`. Vediamo le schema di un semplice programma che si serve di tali functions.

```
#include <stdio.h>
main();
char c,d;
c = getchar();    d = getchar();    // immissione dei due caratteri
if(('a'<=c && c<='Z') || ('a'<=c && c<='z') {// in c lettera
    if('0'<=d && d<='9') {   // in d cifra:  caratteri accettati.
    putchar(c); putchar(' '); putchar('e'); putchar(d);
        putchar(' '); putchar('o'); putchar('k'); putchar(EOF);
        return(1);
        }
    }
// caratteri rifiutati.
putchar('i'); putchar('i'); putchar('n'); putchar('p'); putchar('u');
putchar('t'); putchar(' '); putchar('n'); putchar('o'); putchar(EOF);
return(0);
}
```

Da questo programma si intuisce come mediante la lettura dei singoli caratteri dei dati si possono controllare tutti i loro dettagli. Si vede anche come si possano precisare le emissioni, ma risulta evidente anche la minuziosità di questo modo di organizzare letture e scritture e di conseguenza l'opportunità di disporre di strumenti che consentano controlli più sintetici.

B82 g. operazioni di lettura e scrittura [1]

B82g.01 La massima parte dei programmi prevede la possibilità di leggere i dati concernenti una istanza del problema da risolvere da un'apparecchiatura di ingresso e la possibilità di scrivere i risultati su un dispositivo di uscita.

Nel caso di dati immessi da un operatore che si serve di una tastiera è opportuno far precedere il suo intervento da richieste esplicite.
Inoltre in genere è opportuno accompagnare i risultati inviati a un destinatario umano o artificiale con segnalazioni sufficientemente chiare dei loro significati.
Per certe applicazioni elaborate in risultano opportune anche segnalazioni emissioni che chiariscano il contesto nel quale si sono ottenuti i risultati; a tale chiarimento possono servire tutti i dati di ingresso, i dati 5critici contenuti nella macchina, alcuni dati intermedi che possono far capire lo svolgimento delle manovre eseguite talora alcuni dati acquisiti dall'esterno nel corso dell'elaborazione.

Dei molteplici dispositivi di ingresso e uscita oggi utilizzabili dai sistemi di calcolo qui ne prendiamo in considerazione solo pochi tipi.

Innanzi tutto consideriamo letture da nastri, dischi e memorie flash preregistrate e scritture su nastri, dischi o memorie flash che verranno elaborate in un secondo tempo.
Per le operazioni di ingresso e uscita, ossia per le **operazioni I/O**, ci limitiamo a trattare quelle che riguardano solo files di dati sequenziali descrivibili come stringhe di bytes; per un trasferimento di dati verso e dal computer parliamo di flusso di bytes e più tecnicamente di **input stream** in lettura e di **output stream** in scrittura.

Nelle prossime pagine ci occuperemo principalmente di letture e scritture di files sequenziali simbolici contenenti stringhe ASCII leggibili in chiaro.

Un tale file può servire solo per la lettura o solo per la scrittura e questa operazione può essere effettuata con una sola manovra oppure in fasi successive.

B82g.02 In molti programmi semplici si prevede una sola lettura iniziale dei dati, un cmplesso di elaborazioni di informazioni in memoria e una sola scrittura finale dei risultati.
In generale invece in un programma letture, elaborazioni e scritture possono essere alternate e dipendere dai dati letti e dai risultati intermedi.

In una elaborazione può accadere che un file sequenziale venga letto inizialmemnte e successivamente venga esteso con un flusso di dati prodotti dalle operazioni programmate.
Un file sequenziale a disposizione del programma potrebbe essere utilizzato per ospitare dati intermedi, oppure potrebbe essere letto e riletto più volte.
Queste evenienze si verificano nel caso di penuria di spazio di memoria; erano comuni nei sistemi del passato ma ora con la disponibilità di memorie molto più estese e con la possibilità di ricorrere a sistemi remoti come depositi di dati interessano solo per programmi destinati a dispositivi molto piccoli contenuti in apparecchiature con compiti come il controllo di veicoli o di sonde nello svolgimento di missioni impegnative.
Qui invece ci occuperemo primariamente di elaborazioni che non incontrano vincoli fisici rilevanti, ma riguardano solo manovre che richiedono operazioni ben definite a priori.

B82g.03 Qui prestiamo una precisa attenzione verso apparecchiature costituite da terminale video e tastiera e che consideriamo servano ad immettere o emettere flussi leggibili e più precisamente stringhe ASCII.

Tastiera e video in genere vengono utilizzate insieme e spesso in modo interattivo per scambi bidirezionali di stringhe leggibili, di solito di lunghezza contenuta, in deteminati momenti di una esecuzione.

Va segnalato che di solito lo schermo del video serve anche per mostrare tutto quello che viene digitato sulla tastiera.

Anzi, quando si opera in quella che viene detta **modalità assistita** spesso l'immissione dei dati viene sollecitata da richieste auspicabilmente chiare e spesso viene aiutata con la presentazione dopo ogni rchiesta di liste di alternative possibili, con il completamento di stringhe prevedibili e con proposta di stringhe giudicate correzioni di quelle immesse e considerate imprecise.

Inoltre il video terminale può dare accesso a testi che possono provenire dall'universo Internet, testi che possono essere utilizzati più o meno direttamente per una elaborazione che l'opratore alla tastiera controlla in tempo reale, ossia mentre si sta svolgendo.

Delle operazioni di entrata/uscita prenderemo in esame solo i tipi più semplici e non entreremo in molti dettagli tecnici; ci dedicheremo primariamente a descrivere il funzionamento e l'utilizzo di pochi sottoprogrammi ai quali demanderemo tutte le manovre di immissione ed emissione.

Occorre precisare che non prevediamo di avere immissioni ed emissioni di sequenze di caratteri leggibili; che nelle stringhe elaborate possono essere presenti anche caratteri ASCII non leggibili incaricati in lettura di presentare le stringhe come costituite da records successivi e in uscita incaricati di organizzare le stringhe come linee successive costituenti pagine successive.

B82**g.04** In questa e nelle prossime sezioni dunque prenderemo in considerazione solo programmi piuttosto semplici ciascuno dei quali, in buona sostanza, presenta richieste di lettura di dati iniziali, un successivo complesso di elaborazioni su informazioni numeriche e simboliche e richieste di scrittura dei risultati ottenuti.

Le informazioni lette saranno costituite da stringhe ASCII che risulta necessario trasformare in corrispondenti dati interni che saranno solo numeri interi da registrare canonicamente in celle-32b.
Per ottenere scritture finali saranno necessarie trasformazioni di numeri interi e stringhe ASCII in stringhe che saranno visualizzate come linee e pagine nell quali si hanno i risultati corredati di qualche commento esplicativo.

Le prime trasformazioni annunciate, in relazione al programma le diciamo **operazioni di codifica dei dati**, le ultime **operazioni di decodifica dei risultati**.

Ci proponiamo di fornire ai programmi files simbolici sequenziali che possono essere preparati agevolmente con uno dei comuni source editors, e files da emettere che possono essere visionati come pagine stampate oppure che possono essere eleborate con un source editor interattivo tramite video che possa essere usato agevolmente per ritoccare i files emessi prima di una loro archiviazione o per adattarli, eventualmente ricorrendo ad altri files in qualche modo compatibili, ai fini di loro successivi utilizzi.

In effetti i files sequenziali simbolici costituiscono un mezzo comodo e facilmente controllabile per passare informazioni da un programma ad uno che si può vedere collocato in una posizione successiva in uno schema di un progetto che si serve significativamente di testi leggibili riguardanti dati interessanti per le finalità applicative del progetto, ad esempio dati che servono ad organizzare e governare una attività articolata, immaginabile come una delle molte attività che traggono vantaggio dall'essere supportate sistematicamente da adeguati programmi.

Per denotare questi sistemi di programmi operativamente collegati si usa il termine **programmi da utilizzare in cascata**.

B82g.05 I programmi massimamente semplici non fanno altro che emettere un messaggio su un dispositivo di uscita scelto come strumento standard; con un tale messaggio il programma non può che segnalare il fatto di essere operativo.

Uno di questi programmi ha il seguente testo sorgente.

```
#include <iostream.h>
int main()
{
    cout << "Il programma con questa sola scrittura e‘ operativo" << endl ;
    return(0);
}
```

Commentiamo rapidamente il testo.

La prima linea rende disponibile una libreria di sottoprogrammi che consente di servirsi del comando `cout` e della costante `endl`, gli unici elementi del linguaggio necessari per formulare l'emissione.
La seconda linea costituisce l'intestazione del programma e il suo testo, detto anche corpo del modulo di programma, segue presentato tra una parentesi graffa aperta e una chiusa; queste costituiscono la coppia dei delimitatori coniugati dl testo.
Nella prima linea del testo il comando `cout` richiede l'emissione della stringa prefissata che segue racchiusa tra due doppi apici a sua volta seguita dal carattere `endl` che determina la fine della linea stessa.
La seconda linea del programma è la semplice richiesta di conclusione delle operazioni e si configura come richiamo di un sottoprogramma, un importante tipo di protagonista della programmazione che esamineremo più avanti.

B82g.06 Programmi lievemente più articolati richiedono solo la lettura di taluni valori e la loro successiva riemissione. Il programma che segue prevede la lettura di pochi numeri interi seguita dalla loro ripresentazione.

```
#include <iostream.h>
int main()
{
    int k, m, n ;
    cin >> k >> m >> n ;
    cout << "k = " << k << ", m = " << m << ", n = " << n ;
    getch(); // chiede approvazione di fine run
    return(0);
}
```

La frase che inizia con `cin` prevede la immissione da un dispositivo scelto come dispositivo di lettura standard di tre scritture decimali di interi, la loro codifica e l'inserimento delle corrispondenti sequenze di 32 bits nelle celle assegnate alle variabili `k`, `m` ed `n`; si ha poi l'emissione di questi tre valori preceduti dagli identificatori dalle variabili con le quali vengono individuati nel programma stesso; questi identificatori hanno lo scopo di chiarire il significato di ciascuno dei numeri.
Queste scritture esplicative qui appaiono banali, ma nei programmi che forniscono numerosi risultati hanno grande utilità, rendono il programma uno strumento agevole all'uso e meritano attenzione.

La linea `getch();` richiama un sottoprogramma che fa richiedere all'utente del programma di immettere un carattere; prima di questa azione l'utente ha la possibilità di osservare l'emissione e quindi

di porre fine all'esecuzione con la battuta di un carattere qualsivoglia; la sua lettura sarà seguita dall'esecuzione della frase conclusiva `return(0);` .

Occorre osservare che la scritta che segue l'enunciato `getch();` costituisce un commento al programma.

B82g.07 Il programma precedente prevede un semplicissimo dialogo interattivo, ossia uno scambio di informazioni tra utente del programma e processo esecutivo; qui l'utente può solo l'istante nel quale effettuare l'esecuzione.

Vedremo in seguito come si possono predisporre delle sessioni interattive più articolate, a cominciare da quelle che permettono di controllare l'adeguatezza, la coerenza e la completezza dei dati immessi da tastiera per evitare che qualche errore o qualche mancanza sui dati immessi comporti la esecuzione di elaborazioni con dati intermedi e risultati finali non voluti.

Dei dati erronei, ossia incoerenti con gli obiettivi del programmatore, potrebbero avviare elaborazioni del tutto inutili o, peggio, elaborazioni che illudono sul significato dei risultati ottenuti.
In effetti le azioni che si devono effettuare per garantire la bontà dei dati immessi richiedono controlli e validazioni che si possono effettuare solo utilizzando i comandi di selezione ed iterazione che vedremo, rispettivamente, in B70h e in B70i.
Inoltre la lettura di dati articolati e soggetti a vincoli di coerenza richiede validazioni che seguono logiche che vanno precisate con attenzione alle caratteristiche dei significati applicativi dei dati e in genere conviene che tali controlli siano organizzati con il richiamo di appositi sottoprogrammi.
Queste questioni saranno riprese nella sezione B70j.

B82g.08 Le scritture delimitate da doppi apici consentono di esprimere tutte le stringhe trattabili con sequenze di bytes. In precedenza abbiamo visto solo scritture di stringhe costituite da caratteri semplicemente visualizzabili, lettere, cifre, ",", "'", spazi bianchi.
Vi sono però vari caratteri ASCII, visualizzabili e non, che devono essere rappresentati da sequenze di altri caratteri chiamate **sequenze di escape**. Essi sono precisati dalla seguente tabella.

`\n` new line, inizia nuova linea
`\h` horizontal tab, avanzamento a posizione orizzontale predefinita
`\v` vertical tab, avanzamento a posizione verticale predefinita
`\b` backspace, arretramento
`\r` carriage return, ritorno a inizio linea
`\f` form feed, inizia nuova pagina
`\a` alert, suono di vvertimento
`\\` backslash
`\?` question mark, punto interrogativo
`\'` singolo apice
`\"` doppio apice
`\0` NUL, byte di 8 bits nulli
`\`*o*, `\`*oo*, `\`*ooo* rappresentazione di un byte mediante 1, 2 o 3 cifre ottali
`\x`*h*, ... , `\x`*hhhh* rappresentazione di informazione mediante 1, 2, 3 o 4 cifre esadecimali, ossia mediante 2, 4, 6 o 8 bits

B82g.09 Il linguaggio C++ consente di formulare una ampia gamma di espressioni servendosi di operandi (costanti, variabili e componenti di arrays), di svariati operatori e di parentesi tonde aventi innanzi tutto lo scopo di delimitare sottoespressioni.
Tra gli operatori si distinguono gli operatori aritmetici, gli operatori relazionali e gli operatori logici.

Gli operatori aritmetici riguardano operandi numerici, cioè operandi interi o reali-c, e forniscono un risultato numerico.

- `+` operatore binario di addizione di operandi numerici, usato anche come operatore unario con il solo effetto di evidenziare un valore positivo o un non cambiamento di segno;
- `-` operatore binario di sottrazione tra due operandi numerici e operatore unario prefisso che riguarda un valore negattivo o il cambiamento di segno di un operando numerico;
- `*` operatore binario di moltiplicazione di operandi numerici;
- `/` operatore binario di divisione tra operandi numerici;
- `%` operatore binario di resto di divisione tra due operandi interi;
- `++` operatore unario di incremento per un operando intero che può essere usato come prefisso e come suffisso [:h10];
- `--` operatore unario di decremento per un operando intero che può essere usato come prefisso e come suffisso [:h10].

B82g.10 Una espressione aritmetica costituisce la richiesta di un complesso di operazioni, ciascuna richiesta da un operatore e coinvolgente uno o due operandi; questi possono essere indicati nell'espressione stessa oppure essere forniti come risultato di una operazione eseguita in precedenza. Il succedersi delle operazioni richieste da un'espressione viene retto da regole di precedenza per l'esecuzione tra i due operatori che nel corso del calcolo di un'espressione vengano a essere separati da un solo operando.

Tra gli operatori aritmetici la precedenza maggiore riguarda `++`, `--` e `-` unario; seguono gli operatori `*`, `/` e `%`; infine la precedenza minore è quella di `+` e `-`.

Tra due operatori successivi della stessa precedenza viene eseguito per primo il più a sinistra. La precedenza può essere modificata dalla presenza di coppie di parentesi tonde che vengono a delimitare sottoespressioni che devono essere valutate prima e indipendentemente da quanto sta loro intorno.

B82g.11 Vediamo alcuni esempi di espressioni numeriche molto semplici.

L'espressione `5+7*3` implica per prima cosa il calcolo di 21 (`*` ha la precedenza su `+`) e quindi il calcolo di `5+21` e la messa a disposizione del contesto, cioè della posizione dell'enunciato in cui si trova l'espressione stessa, del valore numerico 26.

Se si vuole invece il calcolo della somma `5+7` seguito dalla moltiplicazione del risultato per 3 va usata l'espressione `(5+7)*3` che richiede di sommare 5 e 7 e successivamente di moltiplicare 12 per 3 ottenendo il valore 36 da rendere disponibile al contesto.

`-31*4-9` fornisce `-133`; l'espressione `a-b-c` equivale alla `(a-b)-c` ed è diversa dalla `a-(b-c)`, equivalente alla `a-b+c` e alla `(a-b)+c`; a sua volta questa fornisce valori diversi da quelli calcolati dalla `a-(b+c)`.

La `(a-b)*(a+b)` equivale alla `a*a-b*b`; `-7*8` fornisce `-56`, come `7*(-8)`, mentre `7*8` e `(-7)*(-8)` forniscono 56.

B82g.12 Per l'operatore divisione applicato a due operandi interi, il dividendo ed il divisore, è opportuno distinguere il caso in cui il primo è multiplo del secondo e quando non vale questa proprietà.

`144/8` fornisce 12, numero chiamato quoziente; `144/8/2` vale 9, come `(144/8)/2`, mentre `144/(8/2)` rende disponibile 36;

`144/8` fornisce 18; `144/8/2` vale 9, mentre `144/(8/2)` produce 36.

`-6+44/11` rende disponibile `-2`, come l'eqivalente `-6+(44/11)`.

Quando il dividendo non è multiplo del divisore la divisione conduce a un quoziente intero ottenuto per troncamento: `18/7` porta a 2, mentre `(-18)/7` e `18/(-7)` producono `-3`.

Solo se il dividendo è multiplo del divisore il quoziente moltiplicato per il divisore riporta al dividendo; questa "ricostruzione" non vale nel caso in cui il divisore non divide il dividendo; in questo caso serve conoscere anche il resto della divisione.

Nel linguaggio c++ (e nel C) il resto della divisione tra gli interi `h` e `k` si ottiene con l'operatore %: quindi `7%3` vale 1 e `44%13` fornisce 5.

Chiaramente per ogni coppia di interi trattabili `h` e `k` vale l'uguaglianza tra `k` e l'espressione `(k/h)*h+(k%h)`.

L'operatore resto si può usare liberamente nelle espressioni numeriche del linguaggio.

Ad esempio `12%5+40%7*23%8` cacola `2+5*7` e in conclusione fornisce 37.

B82**g.13** Presentiamo alcuni esempi di enunciati per la valutazione di espressioni.

I due enunciati

```
int k = 6;  cout << "k*(k+1)/2 fornisce ",k*(k+1)/2," ." << endl ;
```

comportano l'emissione della linea

```
k*(k+1)/2 fornisce 21 .
```

Il frammento di programma che segue mostra gli effetti dell'operatore `++` usato per il preincremento e per il postincremento e dell'operatore `--` usato per il predecremento e per il postdecremento.

```
int n=5; int m=3;
cout << n << ", " << (++n) ", " << (--m) " ;"; // emette   5, 6, 2;
cout << ++n << ", " << m++ ", " << m*n " ;"; // emette   7, 2, 21;
cout << (++m)++ << ", " << (n++)*(++n) ", " << m*(++n) ; // emette   4, 63, 50
```

B82**g.14** **(1) Eserc.** Esprimere le richieste per il calcolo dell'area di un particolare rettangolo con i vertici aventi coordinate intere.

(2) Eserc. Esprimere le richieste per il calcolo del volume di un parallelepipedo retto rettangolo i cui vertici sono esprimibili con coordinate intere.

(3) Eserc. Esprimere le richieste per la emissione delle 4 permutazioni circolari della parola `nove` e in genere di una stringa di 4 caratteri diversi.

(4) Eserc. Esprimere le richieste per il calcolo dell'area di un triangolo con i vertici caratterizzati da coordinate intere pari.

(5) Eserc. Esprimere le richieste per il calcolo dell'area di un quadrilatero con i vertici caratterizzati da coordinate intere pari.

(6) Eserc. Esprimere le richieste per il calcolo dell'area di un poligono con 5 lati con i vertici caratterizzati da coordinate intere pari.

(7) Eserc. Esprimere le richieste per la individuazione e l'emissione di tutti i prefissi di una particolare sequenza di 5 caratteri.

B82**g.15** Come si è detto, nei linguaggi C e C++ si trattano valori booleani, cioè valori interpretabili come `true` e `false`, valori la cui importanza è data dalla possibilità di utilizzarli, vedremo in :h e in :i, per decidere le scelte esecutive sulle quali si basano la versatilità, la adattabilità e la portata applicativa dei programmi.

I due valori di verità true e false sono implementati attraverso valori interi e questo consente di programmare operazioni numeriche sopra valori ottenuti valutando espressioni booleane e viceversa di prendere decisioni di ampia portata basandosi su valori forniti da espressioni numerich e più in generale, su valutazioni quantitative riguardanti dati che possono essere numerosi ed etereogeneri.

Una espressione valutata false fornisce l'intero 0, mentre una espressione che porta al valore true fornisce il valore intero 1. Se invece si vuole ricavare un valore di verità da un numero intero, si ottiene false dal numero 0 e true da ogni altro numero intero.

B82g.16 C, C++ e tutti i linguaggi procedurali dispongono di operatori relazionali su numeri, operatori binari che prevedono due operandi numerici e forniscono un valore che può essere usato sia come valore booleano, che come valore intero binario.

- `<` operatore "minore di";
- `<=` operatore "minore o uguale di";
- `>` operatore "maggiore di";
- `>=` operatore "maggiore o uguale di";
- `==` operatore "uguale a";
- `!=` operatore "non uguale a".

Sono anche disponibili gli operatori logici, operatori binari o unari che prevedono operandi logici e risultato binario.

- `&&` operatore binario AND; fornisce true, 1, sse entrambi gli operandi valgono true;
- `||` operatore binario OR; fornisce false, 0, sse entrambi gli operandi assumono il valore false;
- `!` operatore unario prefisso esprimente negazione, NOT; scambia i valori true e false, o equivalentemente lo scambio tra 0 e 1.

Per quanto riguarda le precedenze esecutive, prevale l'operatore unario !, seguono gli operatori relazionali su numeri e l'operatore `&&`, per ultimo venendo `||`; occorre aggiungere che gli operatori numerici hanno la precedenza sui relazionali come questi precedono i logici.

Per avere espressioni più leggibili tuttavia è consigliabile nelle espressioni con operatori relazionali e logici di far uso di coppie di parentesi anche non strettamente necessarie.

B82g.17 Vediamo alcuni esempi.

L'espressione `35 <= i && i <= 73` fornisce il valore true, ossia l'intero 1, sse il valore attuale della variabile `i`, che supponiamo intera, appartiene all'intervallo $\mathbf{[}35 : 73\mathbf{]}$; essa quindi implementa la funzione indicatrice di questo intervallo entro l'insieme dei valori interi ai quali il compilatore assegna una cella-4B.

L'espressione `i < 0 || 10 <= i` vale true sse il valore attuale della `i` è inferiore a 0 oppure è maggiore o uguale a 10; essa quindi implementa la funzione indicatrice $\mathcal{I}_{\mathbb{Z}}[\mathbf{(} : 0\mathbf{)} \cup \mathbf{[}10 : \mathbf{)}]$ (consideriamo scontata la limitatezza della portata delle celle-4B).

L'espressione `1 <= i&&i <= 6&& - 1 <= j&&j <= 7` fornisce 1 sse il punto-ZZ $\langle i, j\rangle$ appartiene al rettangolo-ZZ caratterizzato dai vertici opposti $\langle 1, -1\rangle$ e $\langle 6, 7\rangle$ e produce 0 in caso contrario; essa quindi implementa la funzione indicatrice del suddetto rettangolo entro $\mathbb{Z} \times \mathbb{Z}$.

L'espressione `1 <= i && i <= 10 && 1 <= j && j <= i` implementa la funzione indicatrice del triangolo rettangolo-ZZ avente come vertici $\langle 1, 1\rangle$, $\langle 1, 10\rangle$ e $\langle 10, 10\rangle$, entro il piano $\mathbb{Z} \times \mathbb{Z}$.

L'espressione `(i <=4) + (j == 6) + (k != 15 && h>i*3)` fornisce il numero delle espressioni relazionali tra le coppie di parentesi tonde che risultano true.

B82g.18 Nei linguaggi C e C++ le variabili di tipo `char` consentono di operare sui caratteri ASCII, visualizzabili o meno, per leggerli, scriverli, confrontarli con altri caratteri, usarli per comporre o per analizzare parole, frasi ed espressioni artificiali (formule, codifiche, ...).
Inoltre le costanti e le variabili `char` variabili possono fornire valori interi a variabili intere e viceversa possono ricevere i loro valori da variabili e costanti intere.
Infatti gli ottetti di bits che forniscono i valori di carattere ASCII possono essere interpretati anche come rappresentazioni di valori interi, in particolare di interi dell'intervallo $[0 : 127]$.

La tabella che segue presenta le codifiche intere di alcuni caratteri visualizzabili.

(1)
$$\left\downarrow \begin{array}{cccccccccccccccc} \texttt{\~{}} & \cdots & \texttt{0} & \texttt{1} & \cdots & \texttt{9} & \cdots & \texttt{A} & \cdots & \texttt{Z} & \cdots & \texttt{a} & \cdots & \texttt{z} & \cdots \\ 32 & \cdots & 48 & 49 & \cdots & 57 & \cdots & 65 & \cdots & 90 & \cdots & 97 & \cdots & 122 & \cdots \end{array} \right\downarrow .$$

Una presentazione più completa si trova in B70c03 e in `ASCII character set (we)`.

Consideriamo i seguenti esempi riguardanti le due interpretazioni dei contenuti delle variabili e delle costanti char.

```
cout << 'a'+'b' " ;"// comporta l'emissione di 195

char carmin, carmai;
cin >> carmin ;  // legge un carattere che supponiamo minuscolo
carmai = carmin - 'a' + 'A' ;  // ottiene il corrispondente carattere maiuscolo
cout << carmai ;   // lo emette
```

B82 h. strutture di controllo selettive

B82h.01 Nelle linee di programma considerate finora sono intervenuti solo pochi e ben definiti componenti elementari del linguaggio: dati, dichiarazioni, variabili, espressioni, sequenze di pochi comandi: assegnazioni, semplici frasi di I/O e richiami di sottoprogrammi con un solo compito ben preciso.
Ora dobbiamo occuparci della redazione di programmi un po' più elaborati e con compiti un po' più compositi.

Cominciamo con il precisare che con il termine **controllo del programma**, in breve **controllo-p**, intendiamo un dispositivo che risulta lecito descrivere in termini antropomorfi come colui che dirige o controlla l'esecuzione di ogni comando espresso nel programma muovendosi sulle sue frasi per far eseguire dai meccanismi interni del sistema hw-sw computer le operazioni che sono implicate dalle frasi che raggiunge nei successivi passi esecutivi.

Per far eseguire le frasi che compaiono nei programmi visti in precedenza precedenti al controllo bastava muoversi procedendo dalla prima all'ultima.
Dato che le operazioni da eseguire per risolvere ogni istanza di problema risolvibile in tempi finiti sono in numero finito si potrebbe pensare che ogni problema potrebbe essere risolto con un controllo che si muove avanzando.
Si obietta che questi programmi possono affrontare solo problemi con istanze che richiedono elaborazioni strettamente simili e quindi hanno una portata molto ridotta, in quanto risulta evidente che tutti i problemi impegnativi riguardano situazioni che possono presentare forti differenze.
In particolare la maggior parte dei problemi impegnativi riguarda insiemi di istanze che richiedono svolgimenti differenti e anche a priori difficilmente prevedibili, sui quali un automatismo non può nutrire incertezze.

Procedimenti e programmi che intendono essere ampiamente applicabili devono essere in grado di distinguere le istanze dissimili e di applicare ad essi manovre diverse; questa capacità di distinzione delle situazioni da affrontare e di scegliere fra successive percorsi operativi diversi costituisce anche un requisito che devono avere i procedimenti e i programmi che aspirano a essere efficaci e a potersi adattare a istanze nuove attraverso semplici ritocchi.
In altre parole quest'ultima qualità riguarda la possibilità di avere programmi in grado di evolversi.

B82h.02 La distinzione delle situazioni che si incontrano deve essere determinata da parametri facenti parte dei dati iniziali o calcolabili a partire da essi; un linguaggio di programmazione capace della suddetta distinzione quindi deve disporre di frasi che consentano di scegliere tra due successive linee di comportamento dall'esame dei parametri disponibili.

Una progressiva crescita delle esigenze dei problemi computazionali che si vogliono risolvere risulta evidente anche dalla storia di tante società in tanti periodi, soprattutto nei tempi più recenti, i più sollecitati ai cambiamenti indotti dal crescere delle tecnologie e dai conseguenti cambiamenti di esigenze e di prospettive.

La crescita delle esigenze computazionali riguarda in particolare la possibilità di sottoporre a elaborazioni automatiche grandi quantità di dati simili: questo è del tutto evidente per le problematiche che richiedono analisi statistiche su popolazioni numerose e sulle problematiche che richiedono valutazioni numeriche ottenibili procedendo per approssimazioni successive.

Per i programmi si tratta della possibilità di far ripetere più volte l'esecuzione di un'intera sequenza di frasi e, come vedremo più in generale, di far ripetere più volte quelli che definiremo "blocchio di frasi".

In altri termini diciamo che è necessario che il controllo possa muoversi, oltre che progressivamente, anche in seguito a proprie scelte e che possa saltare verso le sequenze di frasi (e verso i blocchi esecutivi) che ha stabilito essere gli unici adatti (o i più adatti) alla situazione corrente cheè stato in grado di analizzare quantitativamente e/o qualitativamente.

Si vuole dunque che il controllo possa effettuare anche salti all'indietro per far ripetere azioni e scelte che tengono conto degli effetti delle azioni più recenti.

Si vuole in particolare la possibilità di organizzare iterazioni di sequenze di frasi esecutive, manovre che in particolare consentono l'esplorazione, l'utilizzo e il cambiamento di situazioni che talora si possono raffigurare distribuite come punti di una figura geometrica dotata di regolarità.

Va subito segnalato che si possono avere iterazioni con numeri di ripetizioni definiti in partenza e iterazioni la cui ripetizione solo in conseguenza di dati raccolti dopo ciascuna ripetizione; questo accade in molti calcoli per approssimazioni successive.

Conviene anche far notare che la possibilità del controllo di saltare verso frasi qualsiasi gli consente di ottenere esecuzioni con movimenti ben più articolati delle ripetizioni di sequenze esecutive.

B82**h.03** Per organizzare procedure per esecuzioni non solo sequenziali potrebbero bastare due tipi di istruzioni estremamente semplici (sono quelle adottate nei più elementari linguaggi di macchina): l'istruzione di salto condizionato e l'istruzione di salto incondizionato.

La prima segue il seguente schema:

`se(`*clausola*`) allora salta alla frase` *etichetta*

Qui **clausola** rappresenta in un'espressione logica, in particolare un'espressione relazionale o da una ancor più semplice variabile booleana. Essa però potrebbe anche consistere in un'espressione numerica il cui effetto equivale a quello del valore true se il suo valore è diverso da 0, mentre equivale a false nel caso opposto.

L'entità **etichetta**, o in inglese label, è una scrittura che serve a identificare una e una sola frase esecutiva del programma come possibile meta una o più istruzioni di salto. Ciascuna di queste la chiamiamo **frase bersaglio di salti**

Se il valore attuale della clausola è true, il controllo passa alla frase contrassegnata dall'etichetta indicata; in caso contrario il controllo procede a esaminare ed eseguire la frase che segue l'attuale nel testo del programma.

L'istruzione di salto incondizionato può considerarsi un caso particolare del precedente, segue il semplice schema

`salta alla frase` *etichetta*

e palesemente provoca il salto del controllo alla frase contraddistinta da *etichetta* invece che alla frase successiva.

Nel linguaggio C++ queste frasi assumono, rispettivamente, le forme

`if(`*clausola*`) goto` *etichetta*

`goto` *etichetta*

Occorre aggiungere che nel linguaggio C++ una etichetta è un identificatore e va posta prima della corrispondente frase bersaglio seguita da un segno "`:`".

In seguito presenteremo le ragioni per le quali è buona norma, sia nei programmi C++ che e negli altri linguaggi procedurali, utilizzare solo in pochi casi ben motivati le frasi bersaglio e le etichette.

B82h.04 Stanti le ben note prestazioni della tecnologia elettronica (ma anche elettromeccanica) è ragionevole accettare il fatto che i suddetti dispositivi di programmazione sono effettivamente realizzabili con opportuni circuiti operativi elettronici (ed anche elettromeccanici).

In effetti quando si potevano programmare gli elaboratori elettronici disponendo solo dei linguaggi di macchina (dal 1945 al 1955 all'incirca) e dei primi linguaggi procedurali (il Fortran associato al nome di Backus e il COBOL della McMurray) avvalendosi degli accennati suddetti dispositivi di salto sono stati scritti parecchi programmi efficaci e di ragguardevoli dimensioni che sono riusciti a far capire a non pochi le prospettive del calcolo automatico.

Si può anche ricordare che prima dei computer pionieristici erano stati costruiti automatismi puramente meccanici ed elettromeccanici con notevoli prestazioni. Ci limitiamo ad citare gli automi a controllo idrico e pneumatico dell'epoca ellenistica, le calcolatrici di Pascal e Leibniz e i giocatori di scacchi del secolo XVIII.

Un'altra idea che qui ci limitiamo a tratteggiare, ma che va approfondita, riguarda il fatto che questi dispositivi di salto consentono di organizzare la 'totalità" delle elaborazioni deterministiche concepibili.

Va tuttavia segnalaro che l'adozione di programmi a esecuzione non sequenziale, oltre ad ampliare enormemente la portata della programmazione, ha introdotto anche il rischio di programmi che possono condurre ad esecuzioni che, dopo un numero anche molto elevato di passi esecutivi, non riescono a concludersi e lasciano il dubbio se possano portare a un risultato utile.

Questo comporta la necessità di affrontare un nuovo problema: quello del garantire che un programma non incorra nella possibilità di non arrestarsi mai.
Questo problema può porsi per un particolare programma o per una classe di automatismi e in questo secondo caso viene detto "problema dell'arresto del tipo di automatismo".

Va anche considerato che con programmi a esecuzione non solo sequenziale si giunge a utilizzare strumenti che aprono la possibilità di un infinito potenziale.
È assodato che si vogliono controllare attività concrete, e quindi definite finitamente e che dovrebbero concludersi in tempi finiti con l'impiego di risorse finite; ora si propone di farlo servendosi di strumenti che in linea di principio possono portare avanti le loro operazioni illimitatamente.
In altre parole si adottano strumenti con capacità operative potenzialmente infinite e questo comporta nuovi problemi, a cominciare da quello dell'arresto.
Evidentemente sarebbe stato ingenuo, e anche illusorio, sperare di sfruttare appieno gli automatismi e la loro autonomia senza dover incontrare nuovi seri problemi.

B82h.05 Negli anno dal 1955 al 1965 si sono sviluppati i primi computers (allora si chiamavano calcolatori elettronici o anche, suggestivamente, ordinatori) e contemporaneamente si sono avuti, sinergeticamente, progressi tecnologici, nuovi dispositivi, ampliamenti delle applicazioni, crescita del numero delle macchine, nascita degli studi sulla programmazione e crescita dell'importanza sociale del mondo dell'informatica.
Inoltre si è avuto il prevedibile aumento dei neologismi (software) e della indignazione di molti puristi della nostra lingua.

La crescita della programmazione ha riguardato sia il il numero dei programmi richiesti e messi a punto, sia il numero degli addetti, sia le metodologie della produzione e del mantenimento dei programmi, sia le aspettative riposte nelle soluzioni computerizzate da tanti ambienti, a cominciare dagli amministrativi, dai produttivi, dai militari e dal modo della ricerca.
Intorno al 1960 la programmazione quindi ha cominciato a rivestire notevole importanza culturale, sociale e politica in tutti i paesi industrializzati.

Negli anni successivi, in seguito all'esigenza di disporre di programmi sempre più estesi, incisivi e affidabili e al naufragio di molti progetti, si è andata imponendo l'esigenza di attività di programmazione più consapevoli e più disciplinate.

Sempre più spesso si è reso necessario riprendere programmi preesistenti per modificarli, vuoi per ampliarne la portata e le prestazioni, vuoi per aggiornare e generalizzare i loro obiettivi al fine di usarli per affrontare insiemi di istanze più estesi ed eterogenei, vuoi per aumentare la precisione e la attendibilità dei risultati quantitativi.

A questo punto, poco dopo l'introduzione del termine "software", si è iniziato a prendere in considerazione l'intero **ciclo di vita del software**.

Mentre in precedenza i pregi richiesti ai programmi si limitavano alla attendibilità, alla velocità ed alla precisione dei risultati numerici, sono venuti ad assumere importanza crescente altri pregi: adattabilità, versatilità, estendibilità, scalabilità, esauriente documentazione e quindi leggibilità dei relativi testi sorgente.

B82**h.06** In quel periodo di crescita ci si è accorti che i programmatori erano pochi, che erano portati a seguire abitudini personali spesso estemporanee, che poco si preoccupavano della documentazione dei programmi e della leggibilità dei loro testi sorgente e che poco si interrogavano sui criteri di organizzazione dei loro programmi e poco si confrontavano con colleghi che avevano il compito di rielaborare i relativi testi sorgente.

Più specificamente si è osservato che i programmi nei quali comparivano numerose frasi i `goto` spesso risultavano difficilmente comprensibili anche dagli stessi autori incaricati di riprenderli qualche tempo dopo la loro utilizzazione.
Inoltre si osservava quanto fosse oneroso e privo di linee guida il lavoro per il controllo della correttezza dei sempre più numerosi programmi che si riteneva necessario sottoporre a continui aggiornasmenti.

In effetti il crescere della domanda di eleborazioni automatiche comportava spesso la richiesta di ritoccare e ampliare, spesso con urgenza, le prestazioni di programmi in uso con l'aggiunta di porzioni di altri programmi, compito che spesso si otteneva con l'aggiunta di qualche `goto` verso un blocco di frasi individuato affrettatamente.

In genere questi `goto` avevano bersagli in posizioni poco facili da individuare e i loro effetti complessivi non venivano analizzati con sufficiente completezza. Inoltre quando venivano attuate successive modifiche il controllo e la strategia della organizzazione complessiva delle elaborazioni possibili tendevano a a deteriorarsi progressivamente.

B82**h.07** In quel periodo, in conseguenza delle critiche formulate da vari teorici della programmazione, in particolare da Edsger Dijkstra nel 1968, e grazie a un teorema formulato nel 1966 da Boehm e Jacopini sulla possibilità di evitare istruzioni equivalenti al `goto` nelle macchine di Turing, si è imposta l'opportunità di evitare le istruzioni di salto rimpiazzandole con le strutture di selezione e con le strutture di iterazione che vedremo tra breve.

Si sono quindi progressivamente imposte le accennate strutture di controllo e modalità per la loro organizzazione secondo una certa rigidità ma che, se adottate come criteri finalizzati alla programmazione disciplinata, portano alla disponibilità di programmi più leggibili, più controllabili, piëstendibili e più riutilizzabili.

Queste qualità evidentemente sono a sostegno della prospettiva di attività di programmazione sempre più sistematiche, con obiettivi sempre più ampi e con maggiori possibilità di progressiva evoluzione.

Questo modo di organizzare la programmazione è stato chiamato **programmazione strutturata** e anche di *goto-less programming*.

In seguito cercheremo di seguire diligentemente le prescrizioni della programmazione strutturata.

Tuttavia riteniamo che gli enunciati `goto` in alcune rare situazioni che si possono caratterizzare abbastanza bene consentono soluzioni chiare e poco rischiose re che in questi casi qualche deroga dalla programmazione strutturata può essere conveniente.

B82**h.08** Nel corso di un'elaborazione può accadere che il controllo debba scegliere di affrontare diverse azioni sulla base dei valori attuali di alcuni parametri variabili.
In queste circostanze si dice che si devono organizzare **selezioni** tra diverse linee di azione.

La situazione più semplice riguarda la possibilità, nell'ambito di una sequenza di azioni, di effettuare o meno una manovra più o meno complessa. Con i dispositivi del salto incondizionato e condizionato questa selezione si organizza con frammenti di programma che schematizzabili come segue.

```
azioni precedenti
if(clausola) goto Lsegue;
azioni da eseguire sse non vale clausola
Lsegue :  azioni successive
```

Leggermente più elaborata è l'organizzazione della scelta tra due manovre alternative

```
azioni precedenti
if(clausola goto Laltern;
azioni da eseguire sse non vale clausola
goto Lsegue;
Laltern:
    azioni da eseguire sse vale clausola
Lsegue :  azioni successive
```

Si possono inoltre prevedere selezioni tra tre o più possibilità trattabili con costrutti che sono prevedibili estensioni del precedente.

B82**h.09** Seguendo la programmazione strutturata per la scelta di evitare una azione si adotta il costrutto che segue.

```
azioni precedenti
if(clausola) {
    azioni da eseguire sse vale clausola
    }
azioni successive
```

Una scelta tra due alternativa, chiamata anche scelta dicotomica o dilemma, si presenta come segue.

```
azioni precedenti
if(clausola) {
    azioni da eseguire sse vale clausola
    }
else {
    azioni da eseguire sse non vale clausola
    }
```

azioni successive

In questi costrutti si individuano chiaramente i cosiddetti **blocchi di istruzioni** gruppi di frasi consecutive delimitati da parentesi graffe coniugate, ciascuno dei quali chiaramente condizionato da una clausola che lo precede.

B82**h.10** Veniamo ora alla preannunciata nozione di blocco, che risulta centrale per la programmazione strutturata.

Per blocco si intende un segmento di programma delimitato con chiarezza, in particolare delimitato tra due parentesi graffe coniugate.

Conviene poi distinguere vari tipi di blocchi.
I blocchi del primo tipo sono i moduli di programma caratterizzati da una premessa e da un corpo delimitato da una coppia di parentesi graffe.
Definiamo poi i blocchi primari o di livello 1 i blocchi interamente contenuti in un modulo (al quale si potrebbe attribuire il livello 0).
Abbiamo infine i blocchi di livello superiore ad 1 che sono porzioni di programma interamente contenuti in blocchi diverso da loro modulo; a questi blocchi si attribuisce il livello ottenuto aumentando di uno il livello del blocco che lo contiene.
Si dice anche che un blocco di livello 2 o superiore è **sottoblocco** di quello che lo contiene; questo si può chiamar "sovrablocco" di ciascuno dei sottoblocchi che contiene.

In un modulo di programma quindi si può riconoscere una struttura di arborescenza distesa [] la cui radice è il modulo stesso e i cui archi collegano ciascun blocco ai suoi sottoblocchi.

B82**h.11** Aggiungisamo altre caratteristiche dei blocchi.

Si possono avere sia blocchi semplici costituiti da una sola frase esecutiva, sia blocchi costituiti da una sequenza di frasi formalmente autonome, sia blocchi variamente elaborati possibilmente dotati si sottoblocchi che si basano su costrutti selettivi come quelli introdotti sopra [:h09], su altri costrutti selettivi e su costrutti iterativi.

I blocchi possono contenere anche dichiarazioni (con eventuali inizializzazioni) di variabili che hanno uno **scope**, cioè un ambito di validità, ossia di visibilità e operatività, limitato al blocco stesso.

Va detto anche che per un blocco semplice a una sola frase le parentesi graffe che lo delimitano possono essere tralasciate, in quanto possono alleggerire il testo sorgente.

Nei precedenti costrutti e in gran parte di quelli che stiamo per introdurre viene meno la necessità di introdurre etichette per le frasi alle quali rinviavano le frasi `goto`.
Le caratteristiche strutturali dei blocchi contribuiscono a renderli delle unità operative dotate di rilevante autonomia, simile a quella dei sottoprogrammi.
come vedremo questo porta notevoli vantaggi ale attività di programmazione.

La stesura nel testo sorgente dei costrutti strutturati è opportuno sia realizzata seguendo sistematicamente dei criteri di collocazione delle scritture riservate e delle parentesi graffe.

Adottando diligentemente questi accorgimenti si possono avere programmi di buona leggibilità e che risultano più facili da progettare, da redigere, da adattare al mutare delle esigenze, evento che in molti campi applicativi si verifica di frequente e talora è prevedibile.

Conviene sottolineare che le necessità di aggiornare i programmi, soprattutto quelli di grandi dimensioni, nel tempo sono andate crescendo e che molti problemi legati alle attività concrete richiedono

programmi sviluppati da folti gruppi di programmatori e con aggiornamenti che possono essere preventivati e pianificati.

Infine va aggiunto che i testi ben strutturati si possono presentare abbastanza agevolmente mediante diagrammi di flusso o mediante altre tecniche di visualizzazione che vengono accuratamente studiate nell'ambito dell'ingegneria del software, una disciplina di indubbia importanza industriale ed economica.

B82**h.12** Nel corso di una elaborazione accade spesso che al controllo si pone la scelta tra la successiva esecuzione di diverse manovre alternative. Per esempio si presenti una prima manovra da eseguire sse si verifica una *clausola 1*, una seconda da effettuare sse non si verifica *clausola 1* ma si verifica una *clausola 2* e una terza da eseguire sse non si verifica nessuna del due clausole precedenti.

Il doveroso meccanismo per questa scelta viene implementato dal costrutto rappresentato dagli schemi che seguono (nei quali trascuriamo di indicare azioni precedenti e successive).

```
if(clausola 1) {
    azioni da eseguire sse vale clausola 1
    }
else if(clausola 2) {
    azioni da eseguire sse non vale clausola 1 ma vale clausola 2
    }
else {
    azioni da eseguire sse non valgono né clausola 2 né clausola 2
    }
```

Soluzioni analoghe facilmente intuibili si adottano per 4 o più possibili scelte.

In questo genere di circostanze occorre esaminare l'insieme dei possibili successivi percorsi operativi e ripartirlo in una sequenza di sottoinsiemi di successivi percorsi, ciascun componente della quale sia caratterizzato da un parametro disponibile sul quale si procede a stabilire con il costrutto `if ... then` se imboccare il corrisponebte percorso; l'ultimo sottoinsieme di possibili successivi percorsi è il complementare dell'unione dei precedenti e porta al blocco da far seguire alla occorrenza di `else`, altrimenti.
Si noti che `else` è un esempio di una cosiddetta **parola chiave** o **parola riservata** .

Se tutte le parti dell'insieme dei possibili percorsi successivi sono chiaramente associate a valori di parametri discriminanti conviene organizzare costrutti selettivi privi della possibilità preceduta dalla semplice chiave `else` in modo da avere tutte le vie da selezionare caratterizzate da clausole esplicite chiaramente riconoscibili.

Gli schemi di questi costrutti riguardanti 2 e 3 possibilità (in B70h08 è stato presentato quello con una unica possibilità) sono i seguenti:

```
if(clausola 1) {
    azioni da eseguire sse vale clausola 1
    }
else if(clausola 2) {
    azioni da eseguire sse non vale clausola 1 valeclausola 2
     }

if(clausola 1) {
```

azioni da eseguire sse vale *clausola 1*
}
else if(*clausola 2*) {
}
else if(*clausola 3*) {
azioni da eseguire sse non vale *clausola 1*, non vale *clausola 2*, ma vale *clausola 3*
}

B82**h.13** Presentiamo, ancora sotto forma di frammenti di programma ridotti all'essenziale, alcuni esempi di costrutti con qualche sentore applicativo.

```
// Associa al progressivo del mese il numero dei suoi giorni
in un anno nonbisestile
if(xmese == 2) ngiorni = 28 ;
else if(xmese==4 || xmese==6 || xmese==9 || xmese==11) ngiorni = 30 ;
else ngiorni = 31;
```

```
// Segnalazioni conseguenti al voto vps ottenuto
in una prova universitaria scritta
if(vps <= 12) cout << "deve ripetere prova scritta" << endl ;
else if(vps < 18) {
    cout << "si consiglia di ripetere prova scritta" << endl ;
    }
else if(vps < 24) cout << "presentarsi alla prova orale << endl ;
else {
    cout << "la sola prova scritta comporterebbe il voto 25/30" << endl ;
    cout << "per un voto migliore presentarsi alla prova orale" << endl ;
    }
```

B82**h.14** Come si è detto, in un blocco selettivo possono essere inseriti altri costrutti di selezione e in questi altri ancora; in altre parole si possono organizzare selezioni a più livelli.

Un primo esempio è dato dalla generalizzazione di una selezione in B70g11 la quale non vale per gli anni bisestili.

```
// Dai progressivi dell'anno e del mese ricavare il numero dei giorni del mese
if(xmese == 2) {
    if (xanno%400==0) ngiorni = 29 ;
    else if(xanno%100==0) ngiorni=28 ;
    else if(xanno%4==0) ngiorni=29 ;
    else ngiorni=28 ;
    }
else if(xmese==4 || xmese==6 || xmese==9 || xmese==11) ngiorni = 30 ;
else ngiorni = 31;
```

Un esempio ancor più chiaramente definito di selezione a due livelli riguarda la assegnazione di un punto $\langle x, y\rangle$ del piano sugli interi ad una delle 9 parti di questo insieme determinate dagli assi.

```
if(x<0) {
    if(y<0) cout << "III quadrante" << endl ;
```

```
    else if(y==0) cout << "semiasse orizzontale negativo" << endl ;
    else cout << "II quadrante" << endl ;
    }
else if(x==0) {
    if(y<0) cout << "semiasse verticale negativo" << endl ;
    else if(y==0) cout << "origine" << endl ;
    else cout << "semiasse verticale positivo" << endl ;
    }
else {
    if(y<0) cout << "IV quadrante" << endl ;
    else if(y==0) cout << "semiasse orizzontale positivo" << endl ;
    else cout << "I quadrante" << endl ;
    }
```

B82**h.15** **(1) Eserc.** Redigere un frammento di programma nel quale si controlla che vengano immessi tre numeri interi e si decide se il secondo esprime la posizione sulla retta dei numeri interi di un punto compreso tra i punti aventi come ascisse il primo e il terzo numero.

(2) Eserc. Si chiede un frammento di programma nel quale si decide se tre numeri dati possono esprimere le lunghezze dei lati di un triangolo, distinguendo il caso di tre punti allineati.

(3) Eserc. Si scriva un frammento di programma che sia in grado di porre in ordine due, tre oppure quattro numeri interi positivi, presupponendo che i numeri sono immessi successivamente e sono seguiti dalla immissione di uno 0 conclusivo; va segnalato anche il caso non voluto di immissione di 5 numeri positivi.

(4) Eserc. Si scriva un frammento di programma in grado di porre in ordine alfabetico tre sigle automobilistiche, o più in generale tre digrammi, ossia tre stringhe di due lettere.

(5) Eserc. Redigere un frammento di programma nel quale si esaminano quattro numeri interi e si distinguono i casi nei quali vi sono delle ripetizioni.

B82 i. strutture di controllo iterative

B82i.01 Consideriamo il problema di sommare quattro numeri interi; evidentemente lo si può risolvere con un programma come il seguente:

```
#include <iostream.h>
int main() {
int h,k,m,n,sum;
cin >> h >> k >> m >> n ;
sum = h ;
sum = sum + k ;
sum = sum + m ;
sum = sum + n ;
cout << sum << endl;
return 0 ;
}
```

Un tale programma può dirsi "meramente sequenziale" e in buona sostanza imita il calcolo che si può effettuare a mente o servendosi di una semplice calcolatrice numerica meccanica, elettromeccanica o elettronica.

B82i.02 Se si devono sommare 100 o 1000 numeri questo modo di fare richiede un programma molto lungo, assurdamente prolisso. Inoltre la somma di un numero degli addendi non predeterminato se programmata nel precedente modo puramente sequenziale dovrebbe contenere anche frasi `if` per il controllo della conclusione della somma attualmente richiesta e sarebbe ancor meno ragionevole.

Per avere programmi ragionevolmente gestibili si impongono due nuovi modi di fare. Innanzi tutto è necessario servirsi di gruppi di frasi, che nei casi più semplici si riducono a frasi singole, le quali nel corso di una esecuzione del programma possano essere eseguite più volte, ossia possano essere toccate dal controllo più volte.

Queste gruppi di frasi, come quelli che sono oggetti di scelta nei costrutti selettivi, sono detti blocchi di frasi e sono caratterizzati dall'essere delimitati da coppie di parentesi graffe coniugate; tuttavia ancora questi delimitatori possono essere evitati per i blocchi costituiti da una singola frase.

Questi blocchi più specificamente li chiamiamo **blocchi iterativi** e chiamiamo **ciclo [esecutivo]** ogni loro esecuzione.

Nei blocchi iterativi deve essere possibile richiedere azioni diverse nelle loro diverse esecuzioni, cioè nei diversi cicli iterativi.
In altre parole i blocchi iterativi devono avere una sufficiente versatilità e non devono essere soltanto pedissequamente ripetitivi, ma si deve avere la possibilità di cicli contenenti tutti gli elementi variabili che risultano convenienti.

Per la versatilità dei cicli si possono adottare vari accorgimenti.

Innanzi tutto nei cicli possono comparire operandi che cambiano nelle successive esecuzioni.

Un primo modo per ottenere questo consiste nel porre nel blocco operazioni di lettura che forniscono nuovi dati a ogni nuova esecuzione del blocco stesso, ossia in ciascuna di quelle che chiamiamo **esecuzioni cicliche** o **cicli [esecutivi]** del blocco; talora si usa il termine azione da reiterare.

Un secondo modo di procedere consiste nel servirsi di sequenze di gruppi di dati che possono essere: componenti di arrays disponibili prima del blocco, dati forniti da frasi di lettura appartenenti al blocco risultati di elaborazioni interne al blocco, risultati di richiami di functions che dipendono da parametri fatti variare opportunamente da stadio a stadio.

Tra le elaborazioni interne al blocco conviene segnalare le conseguenze di costrutti selettivi interni al blocco che dipendono da qualche dato variabile e quindi nei diversi cicli consentono di scegliere comportamenti diversi, anche molto.

B82**i.03** Si impone quindi la necessità didisporre inserire in un linguaggio di programmazione dei **costrutti iterativi**, cioè dei complessi di istruzioni costituiti da un blocco che chiamiamo "blocco iterativo", e da elementi organizzativi (parole chiave ed espressioni) con il compito di controllare la sequenza delle esecuzioni delle frasi dal blocco.
Una esecuzione delle frasi di un blocco iterativo la chiamiamoo **ciclo iterativo**, in breve "ciclo", o **manovra ciclica**.
Una esecuzione di tutti i cicli attualmente richiesti da un costrutto iterativo viene detta **iterazione**.

I blocchi iterativi strutturalmente più semplici sono costituiti da una o più frasi esecutive delimitate da parentesi graffe coniugate; queste possono essere trascurate (facotative nel caso di una sola frase.
Si possono anche organizzare blocchi iterativi molto articolati contenenti interamente altri costrutti iterativi e altri costrutti selettivi i quali a loro volta possono essere articolati quanto si vuole.

Le azioni compiute nei diversi cicli presentano differenze tendenzialmente contenute, in quanto devono rendere possibile e conveniente richiederle con un unico blocco di frasi.

Nel linguaggio C++, come in gran oarte dei linguaggi procedurali, si possono organizzare svariati tipi di costrutti iterativi che cercano di soddisfare esigenze diverse e seguono regole sintattiche piuttosto diverse.
Una prima distinzione riguarda: (I) iterazioni la cui sequenza di cicli risulta definita prima dell'inizio della sua esecuzione e dipende tendenzialmente poco dalle circostanze delle diverse esecuzioni; (II) iterazioni la cui sequenza di cicli viene a definirsi nel corso dell'esecuzione delle manovre stesse e dipende in modo determinante dalle circostanze delle singole esecuzioni.

I costrutti del tipo (I) vengono retti da un indice che corre su una sequenza di valori predefinita. La corsa dell'indice tipicamente viene individuata dall'assegnazione all'indice di un valore iniziale, da una modifica, che in genere è un incremento o un decremento dell'indice) da eseguire prima di un nuovo ciclo esecutivo e da una relazione che stabilisce se si avrà ueffettivamente un ciclo successivo o se viceversa l'iterazione è conclusa.

Le iterazioni del tipo (II) vengono governate da una condizione che può dipendere da vari parametri i quali possono cambiare nel corso dell'esecuzione di ogni nuovo ciclo e che determina se si deve proseguire la manovra corrente, oppure interromperla per passare alla (eventuale) successiva, oppure concludere senza ulteriori controlli l'intera iterazione.

Un'altra classificazione delle iterazioni distingue quelle rette da una clausola esaminata prima di eseguire un eventuale nuovo ciclo, quelle governate da una clausola esaminata alla fine di ogni manovra e quelle rette da una clausola esaminata in un certa frase del blocco iterativo o anche in pi¿ ossia in qualche fase dell'iterazione.

B82**i.04** Una semplice tipica iterazione del tipo (I) si trova nel frammento seguente, che si suppone preceduto dalla costruzione dell'array `val`.

```
// Sommare gli interi forniti dalle prime 10 componenti dell'array val[]
    int somma=0;
    for (int i=0; i<10; i++) somma = somma + val[i] ;
```

Qui abbiamo un costrutto `for` con un blocco iterativo ridotto ad una semplice frase di accumulo, un indice di reiterazione, `i`, che viene definito, inizializzato ed utilizzato nel nido tra parentesi tonde che segue la parola chiave `for`; questo indice assume i successivi valori dal valore iniziale 0 fino al valore finale 9 (l'intero che precede 10) con incrementi di 1 a ogni nuovo ciclo, in seguito alla richiesta di incremento `i++`.

L'indice di un costrutto `for` potrebbe anche subire decrementi come nel frammento seguente finalizzato all'emissione dei mesi di un anno procedendo all'indietro, in una sorta di ritorno al passato.

```
    for(int mese=12; mese>0; mese--)
        cout << "mese numero " << mese << val[mese] << endl ;
```

L'indice di un `for` a ogni nuova manovra da reiterare potrebbe subire variazioni diverse dall'aumento o dalla diminuzione di 1 come accade in questo frammento che riguarda la distinzione tra anni bisestili e nonbisestilche vanno dal 2000 al 2099.

```
    for (int anno=2000; anno<2100; anno+=4) {
        ngior[anno-2000] = 366;
        ngior[anno-1999] = ngior[anno-1998] = ngior[anno-1997] = 365 ;
        }
```

L'indice di un costrutto `for` può essere modificato come si vuole da altre frasi che fanno parte del blocco iterativo; una situazione di questo genere tuttavia non è da incoraggiare, in quanto di comprensione non immediata e quindi portatrice di possibili fraintendimenti.

Lo svolgersi delle successive manovre cicliche di una iterazione (I), e in particolare delle iterazioni organizzate dai più semplici costrutti `for`, talora può essere conveniente descriverle come visite di una sequenza di posizioni visualizzabili; questa sequenza potrebbe essere un intervallo numerico, una progressione aritmetica, una progressione geometrica, la successione dei valori contenuti nei componenti di un qualche array, o anche la successione dei valori assunti da una qualche function avente dominio discreto monodimensionale.

B82**i.05** In generale un costrutto `for` presenta una assegnazione iniziale ad una variabile intera (ma potrebbe anche essere una variabile real) che assume il ruolo di **indice del costrutto**, una clausola da testare prima di effettuare un nuovo ciclo e una richiesta di modifica da effettuare dopo ogni esecuzione di ciclo.

L'assegnazione iniziale può contenere anche la dichiarazione dell'indice che in tal caso avrà visibilità limitata al costrutto; viceversa essa può mancare: questo accade quando si è provveduto a una inizializzazione dell'indice prima del costrutto `for` oppure quando si organizza un costrutto `for` che rinuncia a servirsi di un suo indice nel modo che vedremo.

La clausola che condiziona la possibilità di effettuare un nuovo ciclo spesso riguarda il confronto dell'indice con un parametro che può essere modificato a ogni nuovo ciclo, ma che può anche riguardare più parametri nessuno dei quali viene incaricato del ruolo di indice (unico) dell'iterazione.
Una tale clausola può essere molto complessa e/o essere incapsulata in una function [B70e] richiamata nel blocco iterativo; in quest'ultimo caso la clausola può non essere agevolmente rilevabile dal testo

del programma e questa poca visibilità dell'indice comporta un certo rischio di poca controllabilità del costrutto da parte del programmatore.

A loro volta le azioni che vengono eseguite tra una esecuzione di un ciclo e la eventuale successiva possono mancare, oppure possono consistere in un semplice incremento o decremento, oppure essere rette da complessi gruppi di frasi esecutive, in particolare possono venire "incapsulate"' in una function.

B82i.06 Presentiamo altri frammenti con costrutti `for`.

(1)
```
// dato un array int d[100] individuare i suoi valori massimo e minimo
    dMIN = dMAX = d[0] ;
    for(int i = 1 ; i++ ; i<100) {
        if(d[i] < dMIN) dMIN = d[i] ;
        if(d[i] < dMAX) dMAX = d[i] ;
    }
```

(2)
```
// Dato un array int d[1000] e due soglie dTHMI e dTHMA,
    // porre in dacc[<daccL] la sequenza dei suoi valori compresi tra le soglie
    // e porre nell'array daccpos[<daccL] la sequenza delle rispettive posizioni
    daccL=0;
    for(int i = 1 ; i++ ; i<1000) {
    if(dTHM <= d[i] && d[i] <= dTHMA) {
    dacc[daccL] = d[i]; daccpos[daccL++] = i;
    }
```

B82i.07 (1)
```
// Dato un array int d[<dL] di interi che esprimono valori percentuali,
    // costruire l'array decil[<10] dei numeri di occorrenze dei valori
    // che cadono nei 10 intervalli di decili
    int id;
    for(id=0 ; id++ ; id <10) decil[id] = 0;
    for(int i = 1 ; i++ ; i<dL) {
        datt=d[i];
        for(id=0 ; id++ ; id <10) {
            if(datt < 10*id) {
                decil[id]++; break;
                }
            }
        }
```

In questo frammento compare la frase `break;` che si riduce a questa sola parola chiave seguita dal terminatore di frase "`;`"'. Essa ha l'effetto di trasferire il controllo alla prima frase esecutiva che segue la parentesi graffa che conclude il blocco della struttura iterativa nella quale si trova.
Questa frase può trovarsi anche in blocchi iterativi degli altri tipi che vedremo nelle prossime sezioni, blocchi caratterizzati dalle parole chiave `while`, e `do`; inoltre può comparire e nella struttura selettiva caratterizzata dall parola chiave `switch`.

B82i.08 (1)
```
// Lettura di un array monodimensionale di interi vi[] di 31 componenti
    for(int ivi = 0 ; ivi++ ; ivi < 31) {
```

```
        cin >> vi[ivi];
    }
```

(2)
```
// Lettura nell'array code[] di una stringa di caratteri diversi da ';'
    avente al più 40 caratteri e precisazione del loro numero nuc
    int nuc = 0; char carletto;
    for(int nuc = 0 ; nuc++ ; nuc < 40) {
        cin >> carletto ;
        if(carletto == ';') break;
        }
```

(3) Lettura di un elenco di sigle in numero non superiore a 35, ciascuna di al più 10 caratteri alfabetici;
i caratteri delle sigle sono seguite da blank;
le sigle, se minori di 15 sono seguite da sigla blank.

```
    e 0; deteminazione del loro numero Nsigl
    e loro collocazione nei primi Nsigl intervalli di 10 posizioni ciascuno
    facenti parte dell'array sigl[]
    char sigl[350], lett; int Nsigl, c;
    for(c = 0; c++; c<350) sigl[c] = ' '; // predispone blanks in sigl
    for(Nsigl = 0 ; Nsigl++ ; Nsigl < 35) {// corsa sulle sigle
        forc = Nsigl*10; c++; c < (Nsigl*10+10 {// corsa sui caratteri letti
            cin >> lett ;
            sigl[c] = lett;
            if(lett == ' ') break;
            }
        if(c == Nsigl*10) break;
        }
```

B82**i.09** (1)
```
// Scrittura delle stringhe precedentemente lette in righe successive
allineate a sinistra; scrittura delle stringhe precedenti in successive colonne
separate da colonne di un blank da leggersi dall'alto in basso
    char sigl[350], lett; int isigl, Nsigl, c;
    // stampa per righe
    for(Nsigl = 0 ; Nsigl++ ; Nsigl < 35) {// corsa sulle sigle
        if(sigl(Nsigl*10 == ' ') break;
        for(c=0; c++; c<10) {
            cout << sigl[Nsigl*10+c] ;
            endl;
        }
    }
    // stampa per colonne
    for(c=0; c++; c<10) {
        for (isigl = 0 ; isigl = isigl+10 ; isigl < Nsigl) {
            cout << sigl[isigl*10+c] ; cout << ' ';
        }
    }
```

B82**i.10** **(1) Eserc.** Scrivere un frammento di programma che genera le prime 30 potenze di 2.

(2) Eserc. Scrivere un frammento di programma che genera i primi 10 numeri fattoriali a partire dalla loro definizione ricorsiva.

(3) Eserc. Scrivere un frammento di programma che genera le prime componenti della `successione di Fibonacci (wi)` a partire da una sua definizione costruttiva.

(4) Eserc. Scrivere un frammento di programma che genera la tabellina della moltiplicazione tra interi positivi da 1 a 10.

B82**i.11** Il comando `while` si può considerare una semplificazione del comando `for`, in quanto come argomento presenta solo la clausola che serve a decidere se eseguire o meno una nuova manovra ciclica. Il costrutto basato su `while` presenta la forma seguente

azioni di inizializzazione
`while(`*clausola per consentire nuova iterazione*`) {`
    blocco iterativo
    `}`

In alcune esecuzioni il blocco di una iterazione `while` potrebbe non essere affatto eseguito; questo accade sse preliminarmente alla esecuzione della prima manovra ciclica l'espressione condizionale vale false.

Consideriamo un paio di esempi.

(1)
```
si abbia una sequenza di valori interi vi[i] per i < Nvi;
    raccogliamo in viOK[] i primi 20 valori superiori a 273 trovati
    in vi[], tenendo conto che potremmo trovarne meno di 20.
    int vi[300], Nvi;
    lettura o costruzione di vi[0 ... ¡ Nvi
    int ivi = 0, viOK[20], NviOK = 0;
    while(NviOK < 20 && ivi < Nvi) {
        if(vi[ivi] > 273) viOK[NviOK++] = vi[ivi];
        ivi++;
        }
    utilizzo di viOK[0 ...  < NviOK]
```

(2)
```
si abbia una sequenza di valori interi pos[i] per i < Npos;
    raccogliamo in vic1[Nvic1] i primi 10 valori che distano da pos1 meno di 20
    in vic2[Nvic2] i primi 15 valori che distano da pos2 meno di 25;
    Nvic1 e Nvic2 potrebbero essere inferiore a 10; pos1 e pos2 sono molto distanti
    int Npos, ipos=0, pos[200], Nvic1=0, vic1[10], Nvic2=0, vic2[10];
    whileipos < Npos {
        posatt = pos[ipos];
        if(Nvic1 < 10) {
            if(pos1-20 <= posatt &&posatt < pos1+20) {
                vic1[Nvic1++] = posatt; ipos++;
                if(ipos <= Npos) posatt = pos[ipos];
            }
```

```
    if(Nvic2 < 15) {
        if(pos2-25 <= posatt &&posatt < pos2+25) {
            vic2[Nvic2++] = posatt; ipos++;
        }
    if(Nvic1+Nvic2 >= 25) break;
}
```

B82i.12 Presentiamo il costrutto `do - while`, costrutto iterativo che si può considerare una semplificazione del costrutto `for`, oppure come una variante del costrutto `while`.
Esso presenta la forma seguente leggermente più articolata della precedente.

azioni di inizializzazione
`do {`
    blocco iterativo
    `}`
`while(`*clausola di ulteriore iterazione* `) ;`

Esso provoca l'esecuzione di una prima manovra ciclica e successivamente, come dopo l'eventuale esecuzione di ogni nuova manovra ciclica, la valutazione della clausola di reiterazione per stabilire se si deve eseguire una nuova manovra ciclica successiva. Vediamo alcuni esempi.

(1)
```
consideriamo una sequenza di valori interi vi[i] per i < Nvi,
sicuramente inferiori a 2000;
poniamo in vinmadime il primo valore minore di 60 oppure,
in mancanza di meglio, il minimo dei valori in vi[<Nvi]
int vi[300], Nvi;
```
lettura o costruzione di `vi[0 ... < Nvi]`
```
int vinmadime = 2000; posivi = -1; ivi=0;
do {
    if(vi[ivi] < vinmadime) {
        vinmadime = vi[ivi];
        posivi = ivi;
        }
    }
while(vinmadime >= 60) ;
```
utilizzo di vinmadime

B82i.13 Introduciamo i due comandi `continue` e `break` che possono essere utilizzati all'interno dei blocchi iterativi per contribuire agli effetti di ciascuno dei cicli iterativi.

Il comando `continue` può essere posto dopo un comando selettivo `if`, `else if` o `else` oppure nella posizione conclusiva di un blocco subordinato a un comando selettivo.
La sua esecuzione comporta che il controllo salti alla conclusione della manovra ciclica attuale e quindi alla valutazione della clausola di iterazione successiva, evitando tutte le manovre intermedie.

Un frammento schematico che mostra il suo effetto è il seguente.

```
bool completo = false;
while(!completo) {
    bool finissaggio=true;
```

```
    azioni che possono modificare finissaggio e completo
    if(!finissaggio) continue;
    azioni di finissaggio su quanto prodotto nella manovra ciclica
    }
```

B82**i.14** Anche il comando `break` può comparire come comando conclusivo di un blocco subordinato a un comando selettivo all'interno di un blocco iterativo; esso potrebbe anche trovarsi, come vedremo in B70?18, in relazione a blocchi `case` in costrutti `switch`.
La sua esecuzione comporta l'interruzione dell'esecuzione del blocco iterativo; esso quindi viene attivato quando si verificano le condizioni che richiedono questa interruzione.
Tipicamente tale comando compare in una posizione intermedia dei un blocco iterativo in modo da consentire l'esecuzione di una prima parte di una manovra iterativa che risulterà essere l'ultima, evitando l'esecuzione della sua seconda parte.

L'effetto di conclusione della iterazione di un `break` può accompagnare l'effetto di una clausola di reiterazione, oppure servire per controllare la conclusione di una iterazione priva di una sua clausola esplicita.
Un frammento schematico che mostra il suo effetto nella prima situazione di `break` in presenza di clausola reiterativa è il seguente.

```
bool completo = false;
while(!completo) {
    bool stopiteraz = false;
    azioni che possono modificare stopiteraz e completo
    if(stopiteraz) break;
    azioni finali della manovra ciclica
    }
operazioni che si servono di dati modificati nel costrutto while
```

(1) Eserc. Precisare un frammento di programma che inizia con il commento: `// In seq[0..49] si trova una sequenza crescente di interi positivi;`

```
// individuare la posizione del massimo valore inferiore a 100.
int maxvalinf100 = 0;
forint i = 0; i++; i ¡50 {
    if(seq[i] ¿= 100) break;
    if(seq[i] ¿ maxvalinf100) maxvalinf100 = seq[i];
    }
```

B82**i.15** Un'altra situazione che vede una iterazione contenente un `break` e mancante di clausola reiterativa può vedersi come possibilità di servirsi di costrutti iterativi che in apparenza avviano una iterazione illimitata, che ovviamente deve essere evitata.
Lo schema di un frammento riguardante questa situazione è il seguente.

```
while(true) {
    bool stopiteraz=false;
    azioni che possono modificare stopiteraz
    if(stopiteraz) break;
    azioni finali della manovra ciclica
    }
```

L'argomento del `while` è sempre vero e si procede a successive esecuzioni della manovra ciclica fino a che si verificano le condizioni che implicano una esecuzione del comando `break`.

Va osservato che la logica di un tale blocco iterativo deve essere studiata attentamente per garantire che in tempi ragionevoli si abbia effettivamente l'esecuzione di un comando `break`. In caso contrario si avrebbe una ripetizione illimitata della manovra ciclica e il computer resterebbe illimitatamente silenzioso e inutilizzabile.

In una tale situazione si usa dire che "il computer è in loop" e il verificarsi di questa circostanza è del tutto negativo; l'esecuzione va interrotta e il programma va modificato e questo è notevolmente svantaggioso se risulta difficile capire quando si deve arrestare il procedere dell'esecuzione e quali sono le correzioni da apportare al programma.

B82i.16 Vediamo ora il costrutto `switch`, costrutto selettivo che consente di organizzare scelte tra svariate manovre in dipendenza dei valori assunti da una variabile sugli interi o sui caratteri.

Esso si presenta come terna costituita dalla parola chiave `switch`, da un argomento consistente di una variabile o più in generale di una espressione valutabile e da un blocco di istruzioni che si articola in una sequenza di sottoblocchi; ciascuno di questi inizia con una o più coppie costituite dalla parola chiave `case` e da un valore presentato come possibile valore attuale assunto della variabile o dalla espressione che segue `switch`.

Nelle soluzioni più semplici da leggere ciascuno dei successivi sottoblocchi esprime azioni da eseguire se per la variabile di switch si trova uno dei valori presentati all'inizio e si conclude con un comando `break` che implica l'uscita dal costrutto switch. L'ultimo dei sottoblocchi di un costrutto switch può iniziare con la semplice parola chiave `default` collocata prima della formulazione delle azioni che sono da eseguire solo quando il valore della variabile di switch è risultato diverso da tutti quelli previsti per i sottoblocchi presenti.

Vediamo un esempio piuttosto autoesplicativo concernente i 5 `solidi platonici (wi)`.

(1)
```
// Precisazione dei dati caratteristici dei solidi platonici
    switch(numvertici) {
        case 4 :  {    // tetraedro
            numspig = 6; numfacce = 4; n1Schl = 3; n2Schl = 3; break; }
        case 6 :  {    // ottaedro
            numspig = 12; numfacce = 8; n1Schl = 3; n2Schl = 4; break; }
        case 8 :  {    // cubo
            numspig = 12; numfacce = 6; n1Schl = 4; n2Schl = 3; break; }
        case 12 :  {    // icosaedro
            numspig = 30; numfacce = 20; n1Schl = 3; n2Schl = 5; break; }
        case 20 :  {    // dodecaedro
            numspig = 30; numfacce = 12; n1Schl = 5; n2Schl = 3; break; }
        }
```

L'ultimo sottoblocco può mancare del `break` finale, che è evidentemente pleonastico.

In un costrutto `switch` si possono avere sottoblocchi diversi dall'ultimo mancanti del `break` finale; in un tal caso il controllo, dopo l'esecuzione del sottoblocco, invece di passare al comando che segue il blocco switch, procede ad eseguire le operazioni previste nel sottoblocco successivo.

Dunque se più sottoblocchi mancano del `break` finale, dopo l'esecuzione delle manovre previste dal sottoblocco si può avere l'esecuzione delle operazioni di vari sottoblocchi successivi, fino a quelle del primo blocco nel quale viene attivato un comando break.

È prevedibile che in talune di queste situazioni i possibili percorsi del controllo sulle varie frasi possono essere un po' complicati da seguire; questa possibilità consente però di risparmiare molte ripetizioni di comandi.

B82i.17 Negli schemi dei costrutti precedentemente introdotti compaiono blocchi di programma che possono esprimere manovre molto articolate.
In un programma che vuole risolvere un problema ben definito attraverso un procedimento ben organizzato è opportuno che si incontrino blocchi ciascuno dei quali possegga una finalità operativa che possa risultare motivata molto chiaramente agli operatori che hanno il compito di controllare la qualità del programma stesso, o per valutarne il valore tecnico o economico, o per stabilire se e come adattarlo a nuove esigenze.
Questi controlli di qualità possono riguardare diverse scale valutative riguardanti i molteplici parametri collegati agli obbiettivi che si devono tenere presenti.
Ai diversi parametri si possono dare diversi pesi e diverse soglie desiderabili.
Questi parametri in genere riguardano qualità importanti e delicate quali adeguatezza, correttezza, velocità, efficienza esecutiva, facilità d'uso, versatilità, riutilizzabilità e altro ancora.

I blocchi che si possono riconoscere in un programma prendendo in esame le occorrenze delle espressioni riservate e delle parentesi `{`, `}`, `(`, `)`, `[` e `]` possono avere articolazioni e lungheszze molto diverse, dalle più ridotte alle alle più estese.

I blocchi più minuti sono costituiti da un solo enunciato da eseguire o da una sola espressione da valutare.
Altri blocchi sono costituiti da sequenze di enunciati e da sottoblocchi, cioè da blocchi interamente contenuti nel blocco dal quale dipendono; di ogni sottoblocco si dice che è interamente annidato nel blocco nel quale si colloca.
Vi sono poi blocchi costituiti da uno dei costrutti selettivi o iterativi visti in precedenza, costrutti nei quali si individuano dei sottoblocchi.

Analizzando in termini di blocchi e sottoblocchi annidati i testi dei programmi ben strutturati ottenuti a partire da frasi esecutive con le composizioni ottenute con sequenziamenti e costrutti selettivi e curando che tutti i blocchi e tutte la composizioni siano delimitate da proprie parentesi graffe (che possono anche risultare risparmiabili) si possono individuare strutture formali che vengono chiamate arborescenze distese e che sono esaminate in D30.
In queste strutture si possono avere blocchi con livelli di annidamento anche molto diversi, corrisponenti ad arborescenze con cammini massimali di lunghezze molto diverse.

Queste strutture raccomandabili tuttavia non sono le più generali; per schematizzarle tutte occorre considerare anche le composizioni con costrutti iterativi, anch'essi delimitati da proprie parentesi graffe che possono non essere indispensabili.

B82i.18 La programmazione strutturata consente di esprimere tutte le procedure pensabili. Qui non cerchiamo di dimostrare questa affermazione (per la quale rinviamo a), ma ci limitiamo a segnalare questa "illimitata" potenzialità della programmazione strutturata facendo ricorso all'intuizione.

Trovandoci di fronte a un problema da affrontare con una procedura può accadere di individuare un procedimento che richiede di effettuare una dopo l'altra una sequenza di manovre ciascuna delle quali risolve un sottoproblema corrispondente a una porzione del problema complessivo.

Alternativamente si può individuare un insieme di possibilità per i dati che possano essere distinte mediante opportune operazioni e che ciascuna possibilità possa considerarsi un sottoproblema di quello originario.

In una terza alternativa si può individuare una manovra da eseguire con varianti controllabili algoritmicamente e da eseguire secondo una successione determinata, (in particolare in dipendenza di una variabile che corre, ossia che va ssumendo sequenze di valori, in modo controllabile) e ciascuna delle varianti da affrontare possa considerarsi un sottoproblema di quello posto all'inizio.

Si hanno quindi tre modi di ridurre un problema a sottoproblemi e riesce difficile concepire modi diversi da questi per ridurre un problema dato a una composizione di sottoproblemi meno compositi.
In effetti la aoatanziale totalità dei problemi da affrontare con procedure si sono rivelati trattabili con la programmazione strutturata.

I precedenti modi per ridurre un problema portano a delineare un modo di sviluppare un tipo di procedimento complessivo costituito da un programma principale che possa presentarsi come una bozza di programma che si riduce a una struttura sequenziale, selettiva o iterativa di blocchi che andranno singolarmente analizzati e trasformati in strutture contenenti indicazioni più dettagliate.
Questo tipo di riduzione a sottoproblemi da studiare con maggiori dettagli si può attuare a più livelli.
In tal modo si può proseguire fino a che, o si hanno solo blocchi facilmente esprimibili, o ancra più convenientemente blocchi già studiati e formalizzati in precedenza riutilizzabili con facilità, oppure si incontrano sottoproblemi che non si sanno esprimere in termini di soluzione operativa.
Il secondo caso, se si era proceduto correttamente, porta a concludere che non è stato proposto un problema effettivamente risolubile, cosa che può richiedere una riduzione degli obiettivi pretesi, oppure una riformulazione più corretta e accurata del problema stesso e dei fattoriche lo determinano.
Nel primo caso si sta ottenendo un programma ben strutturato in grado di risolvere il problema dato.

B82 j. organizzazione modulare dei programmi

B82j.01 Occorre segnalare che per sviluppare programmi applicativi incisivi in genere servono ampie librerie di functions e che è di grande importanza la facilità del loro riutilizzo per applicazioni diverse da quelle che hanno condotto alla loro prima stesura.

La possibilità di disporre di ampie librerie come accade per i sistemi di sviluppo dei linguaggi che contano su una diffusione ampia e consolidata nel tempo costituisce un elemento di grande importanza per le attività di sviluppo del software e in particolare per la scelta di un linguaggio per ognoi progetto impegnativo.

La disponibilità di librerie di sottoprogrammi versatili e affidabili si può considerare come un importante arricchimento delle prestazioni del linguaggio stesso.
Per un buon uso di queste librerie di programmi è opportuno disporre anche di strumenti che consentano di reperire efficacemente le functions che possono servire. Attualmente sono disponibili vari sistemi di sviluppo per i linguaggi di programmazione più diffusi che posseggono strumenti efficaci e versatili per la gestione delle librerie.
Per lo sviluppo dei programmi da alcuni anni si possono consultare efficacemente anche vari siti Web, in particolare siti attraverso i quali si possono ottenere suggerimenti da intere comunità di programmatori.
Queste considerazioni vogliono solo inquadrare il problema della gestione dei sottoprogrammi e le problematiche della programmazione modulare. Chi intende occuparsi professionalmente di questi problemi dovrà approfondirli adeguatamente.

Qui nel seguito concretamente ci interesseranno invece functions di portata limitata, con finalità ben definite che consentano di esprimere in modo preciso e verificabile una gamma di algoritmi che sia significativa per le nozioni matematiche e computazionali che esamineremo.
Va comunque segnalata la vicinanza concettuale tra la disponibilità dei risultati della matematica attraverso terminologie e notazioni ampiamente condivisibili e le linee generali seguite per la organizzazione delle librerie di sottoprogrammi riutilizzabili per lo sviluppo di prodotti software.
Questa vicinanza si evidenzia soprattutto quando si considerino i sottoprogrammi per la soluzione di problemi computazionali come implementazioni di risultati matematici.

B82j.02 Gli odierni ambienti per lo sviluppo dei programmi scritti in un linguaggio di programmazione di largo uso mettono a disposizione ampie librerie di sottoprogrammi predisposti per risolvere problemi che si presume si debbano affrontare spesso.
Le librerie di sottoprogrammi possono anche essere messe a punto dai programmatori che affrontano problemi specifici operando singolarmente o in gruppi di lavoro, oppure essere acquisite da fornitori specializzati che rendono disponibili prodotti software di largo interesse o su misura, oppure possono essere reperite nei siti del Web finalizzati alla messa a disposizione di software libero.

Nel linguaggio C++ vengono rese disponibili effettivamente molte librerie di functions mediante i comandi `#include` .
Questi sono enunciati di un preciso tipo detto tipo dei **comandi per il preprocessore**; sono frasi ccaratterizzate dal fatto di iniziaro con il carattere `#`, che ciascuno di assi occupa una linea del sorgente del modulo nel quale compare, prima dell'enunciato che inizia con la parola riservata `main`.

Tra le librerie di base per C++ segnaliamo la `iostream`, la tt conio e la `stdio` riguardanti prestazioni di ingresso e uscita e la `math` che raccoglie functions dedicate a operazioni matematiche.

B82j.03 Presentiamo alcune routines per la manipolazione di stringhe.

Preliminarmente va detto che in C/C++ vengono trattate facilmente le stringhe ASCII organizzate in arrays del tipo `char` monodimensionali che fanno seguire i successivi da un carattere `null` che segnala la conclusione della stringa stessa. Una stringa di n caratteri deve avere a disposizione un array di almeno $n + 1$ bytes.

`strcat(str1,str2)` Concatena, ossia giustappone, due stringhe.

`strcmp(str1,str2)` Confronta le due stringhe argomento e fornisce 1 sse coincidono, 0 in caso contrario.

`strcmpi(str1,str2)` Confronta le due stringhe argomento e fornisce 1 sse coincidono oppure presentano differenze maiuscola/minuscola, 0 in caso contrario.

`strcpy(str1,str2)` Riproduce la stringa assegnata a `str2` nell'array `str1`.

`strstr(str1,str2)` Scandisce la stringa in `str1` alla ricerca della prima occorrenza come sua sottostringa della stringa in `str2`.

`strlen(str1)` Fornisce l'intero esprimente la lunghezza della stringa argomento.

`strupr(str1,str2)` Pone in `str1` la stringa fornita da `str2` dopo aver modificato ogni eventuale carattere minuscolo nel corrispondente minuscolo.

`sprintf()` Costruisce una stringa a partire da dati in numero variabile, cioè che in diverse frasi di richiamo possono presentare argomenti in diversi numeri.

B82**j.04** Tra le functions, e in particolare tra le precedenti, occorre distinguere tra quelle che non presentano e quelle che presentano i cosiddetti **effetti collaterali**. L'effetto di un richiamo di una function del primo tipo consiste solo nella fornitura di un valore del tipo assegnato alla function nella sua dichiarazione. Viceversa una function presenta effetti collaterali se un suo richiamo comporta modifiche alle variabili e agli arrays che del richiamo sono gli argomenti.

Tra le functions del paragrafo precedente non presentano effetti collaterali `strcmp`, `strcmpi`, `strstr` e `strlen`.
Ne comportano invece `strcat`, `strcpy`, `strupr` e `sprintf`.

L'esposizione in https://www.mi.imati.cnr.it/alberto/ e https://arm.mi.imati.cnr.it/Matexp/matexp_main.php