1

Capitolo B12
c++, lessico e sintassi [1]

Contenuti delle sezioni

59 pagine

B12 0.01 Questo capitolo è dedicato a una presentazione piuttosto sistematica della parte del linguaggio di programmazione C++ che abbiamo chiamato C++ ridotto, in sigla c++,.

Ricordiamo che con c++ intendiamo un sottoinsieme del linguaggio C++ che in gran parte è contenuto nel linguaggio C; più precisamente la sua portata è un sottoinsieme della portata di C, ma segue alcune regole di C++ assenti dal linguaggio C.

In c++ si trovano tutti gli elementi che consentono di implementare, in modo prettamente procedurale gli algoritmi presentati nell'*esposizione*, algoritmi riguardanti soprattutto 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 sulla gran parte dei computers.

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

B12 0.02 Questo capitolo, il secondo sulla introduzione al linguaggio c++, inizia [:a] introducendo i numeri spesso chiamati "numeri reali", ma che in realtà costituiscono un sottoinsieme finito di $\mathbb{Q}$ e che qui chiamiamo **[numeri] reali-fp**.

Questi consentono di servirsi delle espressioni matematiche, ossia degli strumenti in grado di controllare nel modo più diretto la maggior parte delle costruzioni computazionali della geometria, del calcolo infinitesimale, della fisica elementare e di tutte le loro applicazioni quantitative.

Queste costruzioni per la maggior parte portano a risultati che al livello della matematica formale sono espressi da numeri reali costruibili, ma al livello delle operazioni effettive di interesse pratico si devono servire dei numeri reali-fp o di altri insiemi finiti di numeri razionali.

I problemi che conducono a risultati per i quali bastano i numeri interi sono piuttosto circoscritti, ma essenziali per affrontare problemi con finalità strettamente organizzative come molti di quelli che si incontrano nella ricerca operativa e nella logistica.

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 che richiedono modelli basati su geometria cartesiana, fisica di base e statistica.

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 che è stata verificata nel corso di decenni di calcoli automatici.

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 che ci si sforza di presentare in termini matematici atti a favorire la generalizzazione e la adattabilità a parecchi contesti.
Per tutti questi programmi si cercano motivazioni in applicazioni significative che possano anche avere rilevanza propedeutica.

B12 a. sintassi del linguaggio [1]

B12 a.01 Il testo (sorgente) di un programma nel linguaggio C++ è costituito da uno o più moduli, testi che sono rivolti alla svolgimento di compiti tendenzialmente ben definiti e distinguibili.

Ogni modulo risiede su un files sequenziale simbolico scritto in caratteri ASCII; un programma sorgente si trova su uno o più files, ciascuno contenente uno o più moduliogni perativo e sono estratti da una delle sue librerie di sottoprogrammi.

I programmi del livello più semplice hanno il testo sorgente costituito da un solo modulo main; un programma che qualifichiqmo di taglia ridotta risiede su un solo file che contiene il main e altri moduli dell'utente; i programmi più impegnativi richiedono molti moduli, anche centinaia o migliaia, registrati in più files.
Come vedremo tra i moduli vi sono collegamenti di dipendenza che nei programmi più compositi possono essere parecchio elaborati.

Nei testi di moduli di un programma si distinguono le righe successive e diverse **unità sintattiche** formalmente delimitabili e dotate di loro ruoli ben definiti;
quete unità si possono chiamare anche "unità sintattico-lessicali" e **sintagmi**; spesso in una riga di un modulo si trovano più unità sintattiche, ma si possono avere unità lessicali collocate in più righe successive.
In ogni unità lessicale si distinguono formalmente più unità elementari con ruoli semplici, non riducibili, che chiamiamo **tokens**; si trovano anche, poche, unità lessicali costituite da un unico token; vi sono invece unità lessicali composite UL, nelle quali si riconosce una struttura ad albero (sarebbe meglio dire di arborescenza distesa) nella quale si riconosce un processo di costruzione graduale della stessa UL, oppure l'opposto processo della sua decomposizione progressiva.
Nella struttura ad albero si individuano nodi riguardanti sottounità che risultano collocate su più livelli attribuibili al processo della sua riduzione o al processo della sua costruzione.
I tokens si trovano nei nodi indecomponibili, senza successori dell'arborescenza.
Viceversa si individuano unità sintattiche che non fanno pare di altre e che chiamiamo **frasi** o **enunciati** del linguaggio.

Quindi un programma sorgente si può considerare un complesso di moduli, ciascuno formato da una sequenza di righe, oppure da una sequenza di frasi, oppure, più dettagliatamente, , da una sequenza di tokens. Tutte queste componenti dei progrmmi devono soddisfare precise regole formali e nel corso di ogni esecuzione del programma hanno effetti ben precisi.

B12 a.02 Le unità sintattiche appartengono a ben definite categorie sintattiche; cominciamo con introdurne alcunele seguenti.

parole riservate o "parole chiave": brevi parole precisate nel linguaggio che fanno da demarcatori del testo e contribuiscono a organizzarlo nel suo complesso o nelle sue porzioni alle quali sono assegnati compiti specifici; queste porzioni le chiameremo spesso "porzioni di codice"';

termini riservati o "termini chiave": piccoli gruppi di parole riservate con compiti simili, in genere più specifici, di quelli delle parole riservate;

identificatori o "momi": stringhe scelte dal programmatore per rappresentare nel testo sorgente i dati da elaborare o altre informazioni ausiliarie, ossia in grado di influenzare le elaborazioni;

commenti: scritture che non hanno effetti interpretativi o operativi in quanto sono ignorate dal compilatore, ma servono a rendere più comprensibile la lettura e i controlli del testo e a rendere più agevoli i suoi cambiamenti;

scritture di costanti: scritture che rappresentano dati fissi, non modificabili nel corso di una elaborazione; in precedenza abbiamo visto le scritture di interi, di reali-fp, di valori booleani di caratteri e di stringhe;

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 porzioni di codice e a dotarlo di una struttura sintattica, ovvero servono a definire unità sintattiche.

B12 a.03 Le parole riservate o **keywords** di C++ sono stringhe formate da lettere, soprttutto minuscole, che nel linguaggio hanno ruoli ben definiti.
Per evitare ambiguità si impone che le altre stringhe a disposizione del programmatore per individuare gli identificatori dei dati che controlla, non possono coincidere con alcuna delle parole riservate.
Il loro elenco si trova in J18 e i rispettivi ruoli vengono spiegati nel corso della esposizione della sintassi.
Qui ci limitiamo a presentare quelle che troveremo nelle prossime pagine.

`int`, serve a definire variabili a valori interi;
`char`, consente di definire variabili aventi come valori caratteri leggibili o meno;
`bool`, consente di definire variabili che possono assumere i due valori booleani vero e falso;
`float`, consente di definire variabili aventi come valori reali-fp;
`if`, precede una clausola che nel corso di una esecuzione del programma può essere rispettata o meno e determina i successivi sviluppi della esecuzione stessa;
`do`, inizio della richiesta di attuazione di una certa manovra;
`goto`, comporta il passaggio del cosidetto controllo da una frase del codice sorgente a un'altra;
`new`, inizia una richiesta di nuova memoria.

B12 a.04 I commenti in C++ sono scritture che il programmatore può inserire con notevole libertà nel testo sorgente e che hanno lo scopo rendere il contesto più leggibile fornendo informazioni sulle porzioni di codice presso le quali sono collocate.

Sono diponibili 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 testo sorgente presentati.

B12 a.05 Nelle varie fasi del ciclo di vita di un programma articolato il programmatore può avvalersi dei commenti in vari modi.

Quando il compilatore ha segnalato errori sintattici che si possono trovare in più punti del testo non facili da individuare il programmatore può effettuare prove parziali in ciascuna delle quali sono resi invisibili al compilatore alcuni frammenti del testo, per potersi concentrare su altri; dopo avere trovati legali, dopo eventuali correzioni, i frammenti testati il programmatore si può concentrare sui passi provvisoriamente nascosti.

Molti errori emersi in fase esecutiva possono essere corretti inserendo richieste di emissione di dati intermedi che si pensa possano chiarire dettagli dell'esecuzione; queste richieste sono da commetare opportunamente in modo che, risolto il problema, possono essere eliminati con pochi rischi di errori che potrebbero essere stati commessi con le modifiche motivate dalla indagine.

I commenti possono essere molto utili per documentare un lavoro di programmazione, soprattutto se di lunga durata e in risposta di vari interessi.
Commenti servono per ricordare criteri adottati in fasi di lavoro precedenti, fasi che potrebbero riguardare tentativi o varianti effettuati per esaminare esecuzioni semplificate in attesa di disporre di un testo più completo, più consolidato o da considerare definitivo.

La documentazione serve particolarmente:
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 molte circostanze molte migliorie e molti consolidamenti del codice conviene siano effettuate dopo opportune sperimentazioni che si servono di dati di prima prova giudicati poco rischiosi e/o in grado di comportare percorsi esecutivi facili da monitorare.

B12 a.06 I dati in un programma sono le entità informative a partire dalle quali vengono effettuate le manipolazioni, soprattutto composizioni e trasformazioni, che le cosiddette frasi esecutive del programma richiedono.

Lr frasi esecutive possono richiedere elaborazioni di dati singoli ed elaborazioni di raggruppamenti di dati variamente strutturati; i raggruppamenti di dati che sono più chiaramente definiti secondo la loro organizzazione interna e per quanto riguarda le loro finalità li chiamiamo “strutture di dati”.

Le prime strutture di dati da considerare sono gli schieramenti di dati omogenei chiamati “arrays” e quelli chiamati “enumerazioni”.
Tra gli arrays si distinguoni i monodimensionali e quelli a più dimensioni, due, tre e a più.

Sappiamo che ogni dato trattabile con C++ deve appartenere a un tipo che determina l’estensione della cella di memoria che si usa per registrare ogni dato singolo, quali valori, costanti o variabili può assumere e in quali modi può essere utilizzato o prodotto dalle operazioni, ossia dai circuiti operativi, che possono agire su di esso.

Si distinguono i dati costanti dai dati variabili; nel corso di ogni elaborazione il valore dei primi non può essere modificato, mentre il valore dei secondi può essere cambiato.
In un programma sia le costanti che le variabili devono essere dichiarate, ossia rese individuabili dal compilatore; le dichiarazioni sono richieste mediante frasi apposite.
Le dichiarazioni dei tipi possono riguardare dati singoli o schieramenti di dati; in conseguenza di una dichiarazione il compilatore assegna a una costante, a una variabile o a ogni componente di un array una porzione di memoria che corrisponde al suo tipo.

C++ supporta, ossia consente di utilizzare, molti tipi di dati e il programmatore per ciascuno dei dati da trattare può scegliere il tipo più appropriato.

B12 a,07 Il seguente programma serve a ricordare alcune costanti matematiche.

```
#include <iostream>
using namespace std;
int main() {// fornisce i valori di costanti matematiche
    // introduce quattro variabili cui assegna precisi valori
    float pi = 3.1415926;
    float sqrtofpi = 1.7724538, pipw2 = 9.8696044;
```

```
float goldenratio = 1.61803399;
cout << "pi = " << pi << endl;
cout << "sqrtofpi = " << sqrtofpi << "pipwr2 = " << pipwr2 << endl;
cout << "sezione aurea = " goldenratio, endl;
return 0; }
```

B12 a.08 Nel C++ si distinguono tre categorie di tipi di dati: tipi basici, tipi derivati e tipi definiti dall'utente.

I tipi di dati basici sono fissati nel linguaggio e riguardano dati singoli, ossia identificatori che rendono disponibili valori singoli.

Si hanno sei sottocategorie di tipi basici caratterizzate rispettivamente dalle keywords

`int, float, double, char, bool, void.`

Anche i tipi di dati derivati sono fissati nel linguaggio e ciascuno di essi è una variante di un tipo di dati basico.

Si hanno quattro sottocategorie di tipi di dati derivati caratterizzate, rispettivamente, dalle keywords

`array, pointer, reference, function.`

I tipi di dati definiti dall'utente vengono decisi dal programmatore il quale cerca di avere nel testo sorgente una rappresentazione delle sue richieste operative che gli risulti il più possibile facile da interpretate nei momenti nei quali deve effettuare modifiche o ampliamenti del testo sorgente.

Sono previste cinque sottocategorie di dati definiti dall'utente caratterizzate rispettivamente dalle keywords

`class, struct, union, typedef, using.`

B12 a.09 Il tipo di dati character viene usato per registrere in una cella-8b, ossia in 8 bits, un carattere ASCII che nel sorgente viene presentato tra due segni " ' ", ossia tra due accenti acuti.
Esempi: `char tomo = 'G', sezione = 'd', bkslsh = '\\';`

Questa frase dichiarativa, oltre ad introdurre i tre identificatori `tomo`, `sezione` e `bkslsh`, assegna alle corrispondenti variabili i valori espressi, risp., dalle scritture di costante `'G'` , `'d'` e `'\\'` .

B12 a.10 Il tipo di dati `integer` riguarda le variabili che possono registrare numeri interi in celle-4B, ossia in sequenze di 32 bits; quindi possono essere trattati gli interi da $-2\,147\,483\,648 = -2^{31}$ a $2\,147\,483\,647 = 2^{31} - 1$.
I valori possono essere forniti da scritture binarie, ottali, decimali ed esadecimali.
Esempi `int ini = 12; trm = 0x4d;` `int length = o326; altz = b1011011;`

B12 a.11 Il tipo di dati `boolean` viene usato per trattare i valori di verità `true` e `false`, corrispondenti a 1 e 0; ciascuno di questi viene registrato in una cella-8b, cioè in 1 byte.

Esempi: `bool accepted = 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 intero o floating point che porta a false se vale zero e porta a true in ogni altro caso; si può assegnare anche una variabile booleana anche una scrittura di carattere la quale fornisce false solo se le viene assegnato il carattere `NUL = b00000000`.

```
// esempio di programma su dati del tipo boolean
#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; }
```

Fornisce in uscita

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

B12 a.12 Il tipo di dati `float` (parola chiave che sta per floating point, numero in virgola mobile o reale-fp) riguarda i numeri razionali della collezione dei valori registrabili in celle-4B, in 4 bytes, secondo una ben determinata modalità.

A grandi linee diciamo che si possono trattare come reali-fp valori appartenenti all'intevallo tra `1.2E-38` e `3.E+38`, con una precisione vicina alle 8 cifre decimali.
Occorre però precisare che questo vale per i sistemi che operano prevalentemente su celle da 64 bits, ossia con parallelismo interno da 64 bits.

Esempi di inizializzazione di variabili float: `float pesoinkg = 63.5;` `float costoInEuro = 29.99;`

Il tipo di dati `double` riguarda numeri (razionali) della collezione dei valori registrabili in celle-8B, in 128 bits, secondo una modalità simile a quella dei dati di tipo `float`.
Occorre avvertire che questo tipo di dati è sicuramente disponibile nei sistemi con parallelismo interno a 64 bits, ma potrebbe non esserlo in qualche sistema con parallelismo a 32 bits.

Il tipo di dati `double`, evidentemente, consente di lavorare con numeri molto più precisi e con valori assoluti che possono essere molto maggiori e molto minori di quelli consentiti dal tipo di dati `float`: grosso modo possono essere trattati valori assoluti che vanno da `1.7e-308` a `1.7e+308` con una precisione di 19 cifre decimali.

Esempio `double cstntAvogadro = 6.02214076e23;`

B12 a.13 Il tipo di dati`void` serve solo per introdurre functions che non forniscono un dato di ritorno primario.

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;
    }
```

B12 a.14 Gli elementi sintattici chiamati "data type modifiers", qui modificatori, sono parole riservate da apporre a parole chiave di dati basici al fine di introdurre varianti di tali tipi di dati quando si devono trattare dati che possono assumere valori che si possono muovere in intervalli di valori diversi e/o possono raggiungere precisioni diverse.

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

C++ mette a disposizione 4 data type modifiers associati, rispettivamente, alle parole riservte

`short, long, signed, unsigned .`

I modificatori `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 rendere più chiare alcune situazioni particolari.
Il modificatore `unsigned` è invece necessario quando si vogliono trattare interi naturali, ossia non negativi, che possono assumere valori fino a $2^{32} - 1$, all'incirca doppio di $2^{31} - 1$, il massimo dei valori integer signed.
I dati integer, sia signed che unsigned, riguardano contenuti 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 contenuti di celle-8b; anche `signed char` non è indispensabile, ma può essere utile per dare maggiore leggibilità a porzioni di codice.

B12 a.15 I modificatori `short` e `long` hanno l'effetto, rispettivamente, di dimezzare e di raddoppiare l'ampiezza delle celle dedicate a dati numerici.

La paola riserata `short` si applica solo ai dati di tipo `integer` e consente di trattare numeri interi, signed 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 `long long int` (enfatica, ma in grado di fornire elevata evidenza).

Con dichiarazioni che iniziano con `long double` si rendono disponibili numeri floating point che occupano 16 bytes.

B12 a.16 Introduciamo ora function di libreria piuttosto particolare di nome `sizeof()`, che riguarda i tipi di dati effettivamente disponibili nel sistema in uso.

Introduciamola in modo pratico considerando il programma che segue.

```
#include <iostream>
using namespace std;
// programma che serve a ricordare le estensioni dei diversi tipi di dati
int main() {
    cout << "Estensione di int:  " << sizeof(int) << " bytes" << endl;
    cout << "Estensione di char:  " << sizeof(char) << " byte" << endl;
    cout << "Estensione di float:  " << sizeof(float) << " bytes" << endl;
    cout << "Estensione di double:  " << sizeof(double) << " bytes";
    return 0;
}
```

Il suo richiamo provoca la emissione delle linee seguenti

```
Estensione di int:  4 bytes
Estensione di char:  1 byte
Estensione di float:  4 bytes
Estensione di double:  8 bytes
```

Questa function appare un po' superflua, in quanto fornisce informazioni che dovrebbero essere ben note al programmatore.
Essa può servire per avere programmi portatili, ossia in grado di funzionare correttamente su sistemi che assegnano estensioni diverse a certi tipi di dati.
Stabilita in fase esecutiva una di queste estensioni, il programma ha la possibilità di richiedere le azioni che meglio si adattano ai tipi di dati effettivamente disponibili.

B12 a.17 In ogni linguaggio di programmazione procedurale sono disponibili richieste che introducono informazioni che non sia possibile modificare nel corso delle esecuzioni.

Accade che un tipo di errore di programmazione piuttosto comune consiste nel provocare l'esecuzione di modifiche dannose.
Se si introducono informazioni che il sistema non consente di modificare risulta garantito che il programma non corre il rischio di effettuare loro modifiche.

Il linguaggio C++ rende disponibile la parola chiave `literal`, che apre la possibilità di introdurre informazioni immodificabili associate a variabili che possono avere identificatori atti a chiarire i significati delle informazioni rese gestibili attraverso gli identificatori stessi.
Il termine literal si può consideriamo sinonimo di "scrittura di costante".

B12 a.18 Sono disponibili quattro tipi di literals per i dati integer:

decimal-literal, octal-literal, hex-literal, binary-literal .

Il primo consente le scritture decimali di numeri interi positivi, scritture che devono iniziare con una cifra decimale positiva seguita da nessuna o più cifre decimali.

Il tipo octal-literal riguarda le scritture ottali di interi non negativi, scritture che devono iniziare con la lettera "O" seguita da una o più cifre ottali; esempi: `O36, O343, O1703`.

La richiesta hex-literal consente di esprimere numeri nonnegativi mediante scritture esadecimali: ciascuna di queste inizia con "0x" o con l'equivalente "0X" seguito da una o più cifre esadecimali; queste oltre alle cifre decimali sono a = A che rappresenta il decimale 10, b = B che fornisce 11, c = C che esprime 12, mentre d = D , e = E ed f = F forniscono, rispettivamente 13, 14 e 15.

esempi: `0x23d, 0XFF, 0X4fa56`.

Il tipo binary-literal riguarda scritture binarie di interi nonnegativi, scritture che iniziano con il digramma "0b" o l'equivalente "0B" e proseguono con una o più cifre binarie: esempi `0b1001, 0B00111100`.

Le scritture dei numeri interi possono essere seguite da suffissi che precisano il tipo che si intende attribuire all'intero introdotto.
Per il tipo `int` non è richiesto alcun suffisso, in quanto riguarda l'estensione per default.
Per il tipo `unsigned int` serve il suffisso è "utt" oppure "`U`.
Per il tipo `long int` il suffisso è "`l`" o l'equivalente "`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`".

B12 a.19 Sono disponibili due forme di floating-literals, la forma decimale e la esponenziale, detta anche forma scientifica.

La forma decimale può consistere nella parte intera seguita da punto, nel punto seguito dalla parte decimale e nella parte intera seguita da punto e dalla parte decimale.

Esempi `255., .0034, 39.3565656`.

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 opportunamente limitato.

A una tale scrittura si aggiunge il suffisso `l` o `L` per chiedere la assegnazione al tipo `double float`; se questo manca si ha l'assegnazione per default al tipo `float`.

Esempi: `1.215e-10L, 404.4E-12, 1.423E+23`.

B12 a.20 I character-literals riguardano sia singoli caratteri che sequenze di caratteri.

Nel caso di caratteri visualizzabili la scrittura si riduce a un trigramma formato dal carattere preceduto e seguito da apice singolo, "`'`" ;

Esempi: `'W'`, `'w'`.
Per altri caratteri si deve ricorrre a una scrittura del tipo **escape sequence** [J11???, J16c].

B12 a.21 Gli string-literals sono simili ai character-literals, ma invece che fornire singoli caratteri esprimono sequenze di caratteri e si possono assimilare agli arrays monodimensionali di caratteri.

Le scritture di stringhe sono sequenze di indicatori di caratteri; questi possono essere caratteri visualizzabili oppure escape sequences, e le relative sequenze sono 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 riga di file sorgente o possa essere presentata su una linea dello schermo video.

B12 b. espressioni che forniscono numeri reali

B12 b.01 Un importante tipo di dati è quello che fornisce la possibilità di servirsi di una gamma, necessariamente finita ma per molte applicazioni sufficientemente ampia, di **numeri reali-fp**.

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

Occorre anche dire che spesso questi numeri vengono chiamati sbrigativamente "numeri reali" e vengono proposti senza vere definizioni, approfittando del fatto che consentono di approssimare in modo soddisfacente gran parte dei numeri reali che si devono calcolare nei programmi che forniscono soluzioni accettabili di problemi di interesse pratico.

Osserviamo anche che L'utilizzo dei reali-fp, soddisfacente per gran parte delle attività computazionali mediamente impegnative, in situazioni particolarmente impegnative richiede di esaminare con cura la sua attendibilità.

B12 b.02 Per definire i numeri reali-fp conviene partire dai literals che consentono di esprimerli nei programmi sorgente e dalle configurazioni di bits che costituiscono le loro implementazioni nelle celle di memoria nelle quali vengono registrati.

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 sta tra il miliardesimo e il miliardo.

Tuttavia in molti calcoli di interesse scientifico, tecnologico e finanziario servono anche numeri ben inferiori al miliardesimo e ben superiori al miliardo.
Ad esempio la massa dell'elettrone viene valutata in $9,109\,383\,701\,5 \cdot 10^{-31}$ kg, mentre la costante di Avogadro, ossia il numero delle particelle che costituiscono una mole di sostanza, è definita come $N_A := 6,022\,140\,76 \cdot 10^{23}$ mol^{-1}

Per questi si rende necessaria 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.

B12 c. arrays e geometria cartesiana discreta

B12 c.01 arrays monodimensionali

B12 c.02

B12 c.03 Per rappresentare una grande varietà di situazioni servono gli **arrays a due indici**, le strutture di dati che consentono di implementare le matrici, cioè le entità matematiche che posseggono una grande quantità e varietà di applicazioni e che in particolare consentono di esprimere trasformazioni geometriche importanti come traslazioni e rotazioni.

Nel linguaggio C++, come 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, tutti costituiti da uno stesso numero di componenti elementari.
Segnaliamo che in molti contesti i componenti elementari degli arrays sono detti "scalari", gli arrays monodimensionali sono detti "vettori" e gli arrays di due e più dimensioni sono detti "tensori".

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;
```

In un programma nel quale sono richiesti solo i coefficienti binomiali simmetrici con indici inferiori a 4 basta la seguente frase di dichiarazione e 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 del tipo `int`, ossia celle-4B, ossia si impegnano complessivamente 64 bytes.

In questa sequenza di celle è opportuno distinguere 4 sottosequenze, ciascuna costituita da 4 celle-4B: la prima sottosequenza è dedicata alla prima riga della matrice, accessibile per intero scrivendo `cbs[0]`, la seconda alla seconda riga accessibile mediante l'espressione `cbs[1]`; similmente per la terza e la quarta.

B12 c.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.

B12 d. puntatori, operatori ed espressioni

B12 d.01 I puntatori del linguaggio C++, i **pointers**, sono variabili che hanno il compito di rendere possibile la manipolazione degli indirizzi associati alle variabili nel programma.

In particolare i pointers consentono di organizzare le chiamate per riferimento delle functions, prestazione molto importante che vedremo più oltre, e facilitano le manipolazioni dei componenti degli arrays e di altre strutture di dati, tipicamente gli scorrimenti regolari delle loro componenti.

Le manovre con puntatori consentono di definire e manipolare con comandi concisi le strutture di dati soggette a modifiche di rilievo, entità chiamate "strutture dinamiche".

Ogni variabile del genere pointer deve essere attribuita a un tipo, e può essere utilizzata solo per intervenire su dati di tale tipo.
I puntatori più utilizzati sono dei tipi `int` e `string`.

Per operare con un pointer occorre servirsi dell'operatore unario `'`; esso anteposto a una variabile fornisce il suo indirizzo di memoria, una informazione su una posizione nei dispositivi fisici del computer che il programmatore non deve conoscere con precisione in quanto gli basta sapere che riguarda una ben determinata variabile, ossia la variabile alla quale il puntatore "punta".

Un esempio: si definisce la variabile puntatore `pnt` e le si assegna come valore attuale l'indirizzo della variabile intera `vi` con la frase

`def pointer pnt = 'vi;` di conseguenza il programmatore sa che `pnt` fornisce l'indirizzo della variabile `vi`.

Per ottenere il valore memorizzato nella cella il cui indirizzo è il valore attuale di un puntatore si utilizza l'operatore unario " `*` "anteponendolo all'identificatore del puntatore.
Dunque l'espressione `*pnt-` fornisce il valore attuale della variabile `vi`.

Per utilizzare un puntatore su una sequenza di variabili omogenee occorre aver chiara l'ampiezza in bytes di queste variabili.
Quando incrementiamo di 1 un puntatore che punta alla variabile `v` che occupa k bytes otteniamo l'indirizzo in bytes della cella avente la stessa taglia della `v` e che occupa la posizione in memoria successiva.
Se sommiamo l'intero d (positivo o negativo) a un puntatore a celle di taglia k il cui contenuto è i, indirizzo di un byte della memoria si ottiene l'indirizzo del byte $i + k\,d$ che consente di accedere al valore contenuto nella cella della taglia k che si trova $|d|$ posizioni prima o dopo quella della `v`.

B12 d.02

```
// Programma con esempio di uso di puntatori
#include <bits/stdc++.h>
using namespace std;
void exe_pointer_1()
    {
    int var = 20;
    // dichiara una variabile del tipo pointer a interi
    int* ptr;
    // assegna al puntatore l'indirizzo della variabile var del tipo int
    ptr = &var;
```

```
    // si tenga presente che ptr e var devono appartenere allo stesso tipo
    cout << "valore in ptr = " << ptr << "\n";
    cout << "valore in var = " << var << "\n";
    cout << "valore fornito da *ptr = " << *ptr << "\n";
    }
// modulo main che serve solo a richiamare la function precedente
int main()
{
    exe_pointer_1();
    return 0;
}
```

L'esecuzione cmporta una emissione come la seguente

valore in ptr = 0x7ffe454c08cc
valore in var = 20
valore fornito da *ptr = 20

B12 e. functions e loro richiami

B12 e.01 Per affrontare problemi di interesse pratico si devono scrivere programmi di molte migliaia e anche di molti milioni di linee.

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

Le attività concernenti la produzione di programmi sono ormai sviluppate con criteri e metodi industriali e in esse si rende necessaria una sistematica e razionale suddivisione dei compiti; questa suddivisione riguarda sia le azioni svolte dai programmi, sia le attività delle persone che i programmi progettano, redigono, collaudano, sottopongono a revisione e fanno evolvere in relazione al mutare delle esigenze e dei contesti.

Per la suddivisione delle azioni che ruotano intorno ai programmi, quali che siano i linguaggi di programmazione utilizzati, riveste primaria importanza la organizzazione dei sottoprogrammi.

B12 e.02 Con C++ un sottoprogramma si può definire come una porzione di un programma che al livello lessicale e sintattico del testo sorgente del quale fa parte è organizzata seguendo precise modalità come modulo di programma del genere function e che al livello semantico, ossia al livello del suo significato operativo ha il compito di effettuare un servizio tendenzialmente ben definito in risposta di ciascuna richiesta che in ciascuna esecuzione gli viene rivolta attraverso manovre specifiche che costituiscono un suo "richiamo".

Consideriamo un sottoprogramma ***S*** e un programma ***P*** che richiede le sue prestazioni; il testo sorgente del programma è costituito da un modulo main e da moduli del genere function registrati su un insieme di files preparati dal programmatore o ottenuti da una libreria di sottoprogrammi di interesse generale o specialistico; ogni libreria deve essere esplicitamente richiesta nel modulo main del programma.

In una esecuzione di ***P***, in dipendenza del contesto attuale e in dipendenza, più o meno diretta, dei dati forniti in ingresso, si possono avere più esecuzioni di ***S***, anche nessuna.

Ogni esecuzione di ***S*** viene avviata da un suo richiamo da parte del modulo main o di un altro modulo function, ossia da un altro sottoprogramma a sua volta richiamato dal main direttamente o attraverso altri sottoprogrammi che possono essere considerati degli intermediari.

Ciascun richiamo di ***S*** deve fornirgli tutte le informazioni che determinano precisamente il servizio che gli viene richiesto.
Evidentemente le prestazioni richieste a un sottoprogramma devono essere adeguate alle capacità operative delle quali il sottoprogramma è stato dotato.
Stante le potenzialità complessive della programmazione degli odierni computers e grazie all'efficienza e alla versatilità delle risorse digitali, i sottoprogrammi scritti in un linguaggio come C++ sono in grado di svolgere una vastissima varietà di servizi e possono essere resi altamente efficaci e affidabilmente versatili.
Questo trova riscontro nella gamma delle prestazioni che possono essere assicurate dalle librerie di sottoprogrammi oggi disponibili.
Questo implica che tutti i programmi che risolvono problemi applicativi di interesse concreto demandano la gran parte delle loro prestazioni a una popolazione molto estesa di sottoprogrammi.

Molti programmi oggi valutati come mediamente articolati negli obiettivi e nelle prestazioni richiedono l'intervento di molte migliaia di sottoprogrammi con finalità, compiti e caratteristiche che possono essere grandemente differenziate.

B12 e.03 Procediamo ora ad una rapida rassegna i tipi di sottoprogrammi che incontreremo più spesso nel seguito per esaminarne i contenuti o solo per segnalarne le prestazioni.

I programmi richiesti dalle attività tecnico-scientifiche devono fare ricorso a una grande varietà di sottoprogrammi incaricati di eseguire calcoli specifici. Passiamo in rapida rassegna i maggiori tipi di questi sottoprogrammi.

Calcoli delle funzioni matematiche che si incontrano più spesso: radici quadrate, funzioni esponenziali, logaritmiche e trigonometriche, ...
Trasformazioni di coordinate, conversioni, manipolazioni di figure geometriche, calcolo di parametri statistici.
Procedimenti per risolvere equazioni di ogni genere: soluzioni di sistemi di equazioni lineari, manipolazioni di matrici, determinazione di zeri di polinomi e di altre funzioni di variabile reale; elaborazioni di processi meccanici, termodinamici, elettromagnetici, chimici; previsioni 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, ...
Generazione 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 risultati di elaborazioni statistiche riguardanti dati relativi a popolazioni di migliaia o milioni di esemplari; possono anche essere molto efficaci le animazioni riguardanti le evoluzioni di processi materiali, di comportamenti umani, di evolizioni di popolazioni animali e vegetali, previsioni meteorologiche, monitoraggio di sistemi ecologici.

B12 e.04 Alle operazioni di ingresso e uscita vengono dedicati tanti tipi di sottoprogrammi.
Molti di questi si incaricano del controllo dei dispositivi hardware di molteplici tipologie che si possono connettere a una apparecchiatura 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 asportabili.

A queste in tempi più recenti si sono aggiunte apparecchiature per l'immissione e l'emissione dei suoni, macchine fotografiche, videocamere, monitors televisivi e i collegamenti con reti della telefonia, con reti di computers e con sistemi cloud.

Tutte queste apparecchiature richiedono routines agenti sull'hardware che evidentemente dipendono da una varietàdi caratteristiche tecniche, devono essere controllate da programmatori con competenze specialistiche approfondite e possono anche richiedere il ricorso a linguaggi di progranmmazione con orientamenti diversi.

Inoltre negli ultimi anni si è iniziato a ricorrere a piattaforme di intelligenza artificiale il cui funzionamento richiede potenze di calcolo estremamente elevate.

B12 e.05 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, le codifiche con tre lettere degli stati del mondo, le codifiche dei corpi celesti, le codifiche delle successioni di numeri interi nella collezione OEIS, ...).

Ancora più variegate sono le prestazioni di scrittura: operazioni di decodifica in chiaro 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, mappe con colori aventi significati convenzionali.

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 incaricano 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, elementi estremali, configurazioni 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.

B12 e.06 Esaminiamo ora le caratteristiche delle **functions**, i moduli di programma con i quali nel linguggio C++ sono resi disponibili i sottoprogrammi.

Queste entità della programmazione presentano alcune caratteristiche delle funzioni della matematica e in genere in italiano vengono chiamate funzioni; qui invece per esse usiamo il termine inglese per evidenziare che si tratta di entità di natura differente.

Una function è una sezione sintatticamente ben definita del testo sorgente di un programma che può essere resa operativa attraverso uno o più suoi "richiami" in frasi esecutive i quali comportano le manovre specifiche che essa esprime.

Il testo di una function prevede una intestazione e un corpo che a grandi linee si possono schematizzare come segue.

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

Ogni esecuzione di una function dipende dai valori attuali degli argomenti che compaiono nel suo richiamo; tra le conseguenze di ogni richiamo si può o meno avere un **risultato primario** che viene reso disponibile come operando dell'espressione che contiene il richiamo stesso.

Nelle pagine che seguono incontreremo prevalentemente functions che producono un risultato primario intero; possono essere utili 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 risultato primario, il tipo che va dichiarato esplicitamente nella sua intestazione.
Tuttavia si possono avere functions che non producono un risultato primario, ma solo effetti di tipo collaterale; ad esse va attribuite lo specifico tipo corrispondente alla parola riservata `void` .

B12 e.07 Gli identificatori delle functions seguono le stesse regole lessicali degli identificatori delle variabili e come per questi si pone il problema di evitare le omonimie.

Naturalmente è opportuno che l'identificatore di ogni function sia scelto in modo da rendere evidente il suo compito; questa scelta, più delicata di quanto si possa pensare, va fatta cercando di avere chiari i possibili utilizzi della function e le circostanze operative nelle quali può essere richiamata.

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 l'identificatore che lo rappresenta all'interno del corpo; il tipo di un argomento può essere accompagnato da un modificatore.

Il corpo di una function ha la forma di 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 frasi e di blocchi riguardanti selezioni e iterazioni; è importante che queste composizioni siano ben strutturate.

Nel corpo di una function deve comparire almeno una frase `return` in genere accompagnata da un argomento costituito da un'espressione che può essere delimitato da parentesi tonde.

L'esecuzione di un comando `return` richiede innnzi tutto il calcolo del valore del suo argomento che viene a costituire il risultato primario del richiamo attuale della function.
Effettuato tale calcolo si ha il passaggio del controllo alla frase nella quale è comparso il richiamo attuale della function e l'attribuzione del risultato primario all'operando costituente il richiamo attuale della function.
La corretta esecuzione del richiamo della function richiede che il tipo del risultato primario coincida o sia compatibile con il tipo della function.

B12 e.08 Ogni richiamo a una function è costituito dal suo identificatore seguito dalla cosiddetta **sequenza degli argomenti attuali** delimitata da parentesi tonde; gli argomenti della function sono espressioni dei vari tipi riguardanti operandi che siano valutabili all'atto di ogni esecuzione del richiamo; quindi ad ogni esecuzione del richiamo ogni argomento può fornire un valore a una corrispondente variabile che compare nella intestazione e nel corpo della function richiamata, per costituire uno dei dati sui quali basare la sua elaborazione attuale.

Una function per essere utilizzabile in un modulo di programma deve essere dichiarata nel testo di tale modulo prima di ciascuno dei suoi richiami.
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 compilato che successivamente, dopo la manovra di linkaggio dei vari moduli che costituiscono il programma (che vedremo più oltre) viene reso parte integrante del programma eseguibile in grado di governare ciascuna delle esecuzioni che potranno essere richieste.

La dichiarazione di una function esprime le sue caratteristiche essenziali, cioè i tipi dei suoi argomenti e del suo risultato primario e ad essa devono adeguarsi l'intestazione della function e tutti i suoi richiami.

In una dichiarazione di function la specificazione di un argomento può presentare, oltre al suo tipo, un identificatore fittizio, dummy, che conviene scegliere in modo di fornire a chi legge il suo testo un'indicazione del ruolo dell'argomento stesso, ovvero del suo significato operativo.
Occorre anche segnalare che la dichiarazione di una function contribuisce a documentare in forma concisa il suo significato e per questo va redatta preoccupandosi che possa essere facile da interpretare.

B12 e.09 Anche il modulo main di ogni programma viene strutturato come una function, con la differenza che l'identificatore di tale function è prefissato, non può che 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 spesso 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; il nome del file del programma da eseguire, se il valore dell'argomento `argc` è positivo, deve essere seguito da di stringhe che hanno il compito di specializzare la esecuzione attualmente richiesta.
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` (nome che sta per "argument values").
Questo meccanismo e il ruolo di "`**`" verrà precisato più oltre.

B12 e.10 Analizziamo gli effetti che può avere l'esecuzione di un richiamo di function.

Il richiamo di una function presenta la sequenza dei suoi argomenti attuali che devono essere in precisa corrispondenza biunivoca con gli argomenti fittizi che compiono nella intestazione della function.
Occorre distinguere tra gli argomenti di un richiamo di function quali vengono "richiamati per valore" e quali sono "richiamati per indirizzo", in quanto riguardano due diversi procedimenti per scambiare informazioni tra modulo richiamante e function richiamata.

Con l'esecuzione di un richiamo di function ciascuno dei suoi argomenti fornisce un valore attuale che viene passato al corrispondente argomento fittizio, chiamato anche **dummy variable**.

Occorre distinguere se un valore passato è un indirizzo di memoria fornito da un puntatore o da una espressione di puntatori, oppure un risultato di operazioni sopra i dati disponibili nel modulo richiamante; distinguiamo quindi tra le informazioni passate da un richiamo di function, rispettivamente, i "valori indirizzo" dai "valori calcolati".

B12 e.11 Ciascuno dei valori calcolati viene fornito da una variabile o, più in generale, dal risultato fornito di una espressione valutabile e viene riprodotto nella corrispondente variabile fittizia; un tale argomento ottenuto dalla function viene detto **argomento passato per valore**.

Il fatto che che il valore fornito da una semplice variabile del modulo chiamante viene riprodotto in una variabile della function garantisce che tale informazione non può essere modificatoe all'interno della fuction, ossia in un ambiente che il programmatore del modulo chiamente può permettersi di ignorare.

Una modifica di un dato con un suo ruolo nel modulo chiamante non leggibile direttamente dal suo programmatore costituirebbe un cambiamento opaco portatore di un rischio di fraintendimento del controllo delle operzioni programmate.

Evidentemente un argomento nella chiamata di una function costituito da una espressione non comporta alcun rischio di modifica non palese di una variabile del modulo chiamente.
Il richiamo per valore quindi contribuisce a rafforzare la controllabilità dei programmi.

B12 e.12 Il passaggio degli indirizzi a una function lascia invece aperta la possibilità di modificare i contenuti delle celle di memoria individuate dagli indirizzi stessi, in quanto queste celle sono accessibili sia al modulo richiamante che alla function.
Dato che queste celle sono accessibili nel modulo chiamato solo attraverso i loro indirizzi, le modifiche che la function può effettuare su di esse sono piuttosto evidenti e controllabili.

Il rischio di incongruenza tra il controllo del modulo chiamate e il controllo della function è elevato soprattutto quando il modulo richiamante e la function sono curati da programmatori diversi o da un solo programmatore ma in momenti diversi,

Le due diverse modalità di passare gli argomenti delle functions costituiscono una certa complicazione, ma riescono effettivamente a rendere meno frequenti le modifiche di dati del modulo chiamante da parte di istruzioni formulate nel testo sorgente della function che non hanno riscontro evidente nel testo del chiamante.

B12 e.13 Una function può avere argomenti costituiti da arrays. Uno di questi potrebbe essere passato al sottoprogramma per valori assegnando i valori delle componenti dell'array del modulo chiamante acomponenti di un array manipolabile nella function chiamata; una tale manovra di copiatura però può essere costosa in termini di consumo di memoria e di tempo esecutivo.

Molto più frequentememte un array viene passato a una function chiamata attraverso l'indirizzo della sua prima cella; in tal modo i contenuti dell'array possono essere utilizzati e modificati, ma solo attraverso un indirizzo e quindi mediante una modalità che il programmatore è invitato a controllare con molta attenzione.

Le manovre implicate da un richiamo di function forniscono il servizio che costituisce la ragion d'essere della function stessa.
Per ogni esecuzione di un richiamo di una function dopo la riproduzione dei valori degli argomenti, vengono eseguite 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 la organizzazione delle functions di un programma richiede che siano collocate a diversi livelli a formare un digrafo ordinato il quale può essere assai intricato.

Solo in pochi casi si ha un digrafo dei sottoprogrammi con la struttura di una una arborescenza, quando ogni sottoprogramma viene richiamato da un solo modulo di livello superiore.
Ancor più rara la possibilità di un digrafo totalmente ordinato di sottoprogrammi, ossia di una sequenza (gerarchica) di sotttoprogrammi.
È piuttosto rara anche la possibilità di un digrafo graduato con sottoprogrammi in successivi livelli e con richiami possibili solo da un livello di maggiore responsabilità a uno dei livelli immediatamente inferiori.

B12 e.14 Consideriamo il lavoro di due programmatori: P_M si occupa di un modulo **M** che richiama una function **F** curata dal secondo programmatore P_F; non escludiamo il caso nel quale uno stesso programmatore $P_M = P_F$ dopo aver scritto **F** si deve curare di **M**, ma solo dopo parecchio tempo.

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 per spiegarli con maggiore completezza.
In genere è anche opportuno che i dettagli di **F** non compaiano nel testo di **M**, in quanto potrebbero rendere più faticosa la comprensione.
In effetti ogni function potenzialmente 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 dei richiami della function, mentre gli può essere conveniente evitare di addentrarsi nei dettagli delle operazioni che vengono effettuate nel corso delle possibili esecuzioni della function.

B12 e.15 In altre oarole, è opportuno favorire una divisione dei compiti tra il programmatore della function e il programmatore di 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 fruibile, soprattutto quando si tratta di una function molto articolata della quale si vuole utilizzare solo una parte ben circoscritta delle sue numerose prestazioni.

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 delle cosiddette function libraries alle quali si deve ricorrere, alla documentazione delle prestazioni dei meccanismi che può fornire e ai risultati delle prove di collaudo, in modo da raggiungere in poco tempo la garanzia della adeguatezza e della correttezza delle function che saranno richiamate dal programma sorgente che si sta per redigere.

B12 e.16 Aggiungiamo alcune prestazioni perticolari dei richiami di function.

Si possono usare anche frasi esecutive della forma

espressione contenente un richiamo di function ;

L'effetto di questa frase è la valutazione dell'espressione, processo che può comportare alcuni cosiddetti **effetti collaterali** (*side effects*), cioè modifiche di variabili che possono avere un ruolo importante nelle elaborazioni successive e delle quali il programmatore dei suoi richiami rischia di non avere consapevolezza.

Nel caso particolare di richiamo di function isolato e seguito dal solo "`;`" il risultato primario ottenuto da ogni sua esecuzione resta inutilizzato; avranno seguito solo gli effetti collaterali che nel modulo chiamante non compaiono, ma che devono essere ben presenti a chi si interessa delle prestazioni del modulo richiamente.

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, scritture di complessi di risultati.
Possono essere eseguite manovre rilevanti sopra le cosiddette "variabili globali" nella memoria centrale o disponibili sullo spazio chiamato "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 del programma in esame; in particolare si possono avere interazioni con il Web, ossia con un ecosistema informativo in continua crescita.

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 addetti interessati ai risultati che possono essere influenzati e ad altri programmatori che fossero incaricati di aggiornare o di riutilizzare il modulo contenente i richiami in esame.

B12 f. files e manovre di immissione ed emissione

B12 f.01 Un computer odierno può collegarsi a una grande varietà di dispositivi esterni, ovvero di **unità periferiche**‘.

Tra queste apparecchiature le più semplici da descrivere e da motivare sono tastiere, schermi video, stampanti, lettori/registratori di nastri e di dischi fisici o simulati su memorie a stato solido.

Un computer inoltre può proficuamente scambiare dati con sensori e attuatori collegati ad ambienti domestici, urbani, e naturaiu, con droni, con apparecchiature e con interi impianti industriali, con reti di comunicazione, con sistemi di logistica distribuiti su grandi aree, con l'intero Web globale, con piattaforme cloud, con computing farms, con costellazioni satellitari e con quant'altro.

C++, come ogni altro linguaggio di programmazione di portata generale, deve consentire una ampia varietà di manovre di entrata e uscita.

Questo si realizza soprattutto facilitando la utilizzazione di una ampia gamma dei sottoprogrammi di I/O, per l'immissione e l'emissione di dati, resi disponibili in librerie adeguatamente organizzate, in particolare per quanto riguarda il lessico e la sintassi dei comandi l'accesso alle unità periferiche.

Tra le varie librerie I/O disponibili ai programmi in C++ si deve distinguere, innanzi tutto, tra quelle introdotte con il linguaggio C, tendenzialmente più elementari, e quelle adottate nelle successive versioni di C++; queste in parte sono di uso più impegnativo, ma forniscono prestazioni più complete e danno maggiori garanzie di sicurezza.

B12 f.02 Come molti sottoprogrammi anche quelli dedicati alle operazioni di I/O fanno parte di librerie richiamabili attraverso i rispettivi cosiddetti **headers**; un modulo che fa uso di questi sottoprogrammi deve presentare frasi iniziali che consentono l'accesso a una di queste librerie attraverso il suo specifico header.

Queste frasi hanno la forma

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

L'header per i classici sottoprogrammi di I/O del linguaggio C ha come specifica `stdio.h` .

Nell'ambito dei sistemi di sviluppo per C++ sono disponibili vari altri headers.

`iostream.h` header che fa accedere a sottoprogrammi per la lettura e la scrittura dei cosiddetti **streams** o "data streams", termini che traduciamo con "flussi di dati [sequenziali]".
Un data stream realizzato o potenziale è costituito da una sequenze di righe di bytes prevalentemente visualizzabili che possono essere gestite da apparecchiature periferiche che hanno la capacità di controllare la lettura e/o la scrittura di tali sequenze di caratteri.

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

`fstream.h` riguarda operazioni per l'emissione su e l'immissione da files registrati su memorie di massa (nastri o dischi).

B12 f.03 Le più semplici operazioni di I/O 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') {
// ha stabilito di avere una lettera nel byte c
    if('0'<=d && d<='9') {   // in d cifra:
// segnala che c e d sono accettati
    putchar(c); putchar(' '); putchar('e'); putchar(d);
        putchar(' '); putchar('o'); putchar('k'); putchar(EOF);
        return(1);
        }
    }
// caratteri rifiutati.
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 dopo la lettura dei singoli caratteri 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.

B12 g. operazioni di lettura e scrittura [1]

B12 g.01 La massima parte dei programmi prevede la possibilità di leggere da un'apparecchiatura di ingresso i dati iniziali della esecuzione, dati che determinano la istanza del problema che ci si accinge a risolvere; inoltre tutti i programmi prevedono la possibilità di scrivere i risultati ottenuti con l'esecuzione corrente 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 risultano opportune anche le emissioni di segnalazioni che chiariscano il contesto nel quale si sono ottenuti i risultati.

A tale chiarimento possono servire tutti i dati di ingresso, alcuni dati critici contenuti nel programma, alcuni dati intermedi (dati ottenuti nel corso della esecuzione) in grado di far capire lo svolgimento delle manovre eseguite o in corso di svolgimento e talora anche 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 di dati che potranno essere utilizzati in un secondo tempo su nastri, dischi o memorie flash.
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 in lettura e più tecnicamente di **input stream** e flusso di bytes in scrittura ovvero di **output stream**.

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 ciascuna di queste operazioni può essere effettuata con una sola manovra oppure può essere eseguita in fasi successive.

B12 g.02 In molti programmi semplici si prevede una sola lettura iniziale dei dati, un complesso di elaborazioni di informazioni registrate in celle della memoria e una sola scrittura finale dei risultati.
In generale invece in un programma letture, elaborazioni e scritture possono essere alternate e possono svolgersi in modi diversi determinati dai dati letti e dai risultati che si vanno costruendo.

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 di un programma potrebbe essere utilizzato per ospitare grandi quantità di dati intermedi, oppure potrebbe essere letto tutto o in parte una o più volte.

Queste evenienze si verificano nei casi di penuria di spazio di memoria; questi erano comuni nei sistemi del passato, mentre con l'attuale disponibilità di memorie molto più estese e con l'attuale possibilità di ricorrere a sistemi remoti come depositi di dati, l'uso di memorie esterne si risontra solo in 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 ci occuperemo primariamente di elaborazioni che non incontrano vincoli fisici rilevanti, ma riguardano solo manovre che richiedono operazioni ben definite a priori.

B12 g.03 Nelle prossime pagine limitiamo l'attenzione alle apparecchiature periferiche costituite da tastiera e schermo video che consideriamo in quanto strumenti per immettere ed emettere flussi di dati leggibili, ossia stringhe ASCII anche molto lunghe.
Prendiamo in considerazione, ma solo per lo svolgimento di compiti ausiliari, dispositivi in grado di registrare e rendere disponibili flussi di dati come nastri, dischi ed equivqlenti memorie a stato solido e asportabili.

Tastiera e video in genere vengono utilizzate insieme e spesso in modo interattivo per scambi bidirezionali di stringhe leggibili, di solito di lunghezza contenuta, tra il programma in esecuzione e l'operatore che ha il ruolo del suo utente.
In queste circostanze si dice che il programma opera in **modalità assistita**.

Va segnalato che in tale modalità lo schermo del video serve anche per mostrare tutto quello che l'utente procede a digitare sulla tastiera.

In questa conversazione tra uomo e macchina si incontrano molte immissioni di dati sollecitate da richieste del programma.
Queste richieste è opportuno siano espresse con la massima chiarezza e in assenza di ambiguità.

In particolare risultano comode e prive di ambiguità le richieste che iniziano presentando il genere dell'oggetto richiesto e proseguono con la lista degli specifici oggetti tra i quali l'utente deve scegliere; queste richieste le chiamiamo "richieste con alternative prefissate".

Simili sono le richieste che invitano a digitare un nome e che nel corso della digitazione da parte dell'utente si preoccupano di rifiutare caratteri che non portano a indicazioni accettabili e a completare nomi individuabili dai primi caratteri inseriti; in questi casi si potrebbe anche avere in risposta a un nome nonprevisto la proposta di confermare uno dei nomi simili previsti.

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

B12 g.04 Per le operazioni di entrata/uscita non entreremo in molti dettagli tecnici e cominceremo con le situazioni più semplici.
Iniziamo con la descrizione del funzionamento e dell'utilizzo di sottoprogrammi ai quali demandiamo tutte le manovre di immissione ed emissione.

Occorre precisare che nelle stringhe elaborate possono essere presenti anche caratteri ASCII non leggibili incaricati in lettura di presentare le stringhe come costituite da righe successive e in uscita incaricati di organizzare le stringhe come linee successive costituenti pagine successive.

B12 g.05 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 gruppetto di elaborazioni su informazioni numeriche e simboliche e infine richieste di scrittura dei risultati ottenuti.

Le informazioni lette saranno costituite da stringhe ASCII che risulta necessario trasformare in corrispondenti dati interni che saranno prevalentemente numeri interi da registrare canonicamente in celle-32b.

Per ottenere scritture finali saranno necessarie trasformazioni di numeri interi e stringhe ASCII in stringhe che possono essere visualizzate come linee e pagine nelle quali compaiono i risultati corredati di qualche commento esplicativo che in genere si preferisce conciso.

Le trasformazioni dei dati letti nei dati memorizzati da elaborare, dal punto di vista del programma le diciamo **operazioni di codifica dei dati**;, le operazioni per la presentazione dei risultati le chiamiamo invece **operazioni di decodifica dei risultati**.

B12 g.06 Ci proponiamo inoltre di fornire ai programmi files simbolici sequenziali che possono essere preparati agevolmente con uno dei comuni source editors, e di chiedere ai programmi l'emissione di files che possano essere visionati come pagine stampate oppure che possono essere rielaborate con un source editor interattivo tramite tastiera e video per riuscire con poca fatica a ritoccare i files emessi prima di una loro archiviazione o per adattarli, eventualmente portandoli in nuovi files, nella prospettiva 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 si pensa a 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 che vengono utilizzati in successione si usa il termine **programmi da utilizzare in cascata**.

B12 g.07 I programmi più semplici non fanno altro che emettere un messaggio su un dispositivo di uscita scelto come strumento standard, ossia come strumento di uso molto facile per compiti poco impegnativi e ricorrenti.
Con il susccennato 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 del 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.

B12 g.08 Programmi lievemente più articolati prevedono la lettura di alcuni valori e la loro successiva riemissione.
Il programma che segue prevede la lettura e la scrittura di tre numeri interi; più precisamente si aspetta una prima immissione di un numero, con un secondo intervento sulla tastiera l'immissione di due numeri e infine la loro emissione su due righe.

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

Le due frasi che iniziano con `cin` prevedonoe la immissione da parte dell'utente dal dispositivo di lettura standard, in genere una tastiera, della scrittura e l'invio di un intero e la successiva scrittura seguita da invio di due interi; gli interi sono da esprimere con notazioni decimali.

Inoltre si ha la codifica dei tre numeri in tre sequenze di 32 bits e la registrazione di queste nelle tre celle assegnate, rispettivamente, alle variabili `k`, `m` ed `n`.

Le frasi successive provocano l'emissione in quello che è stato scelto come dispositivo di scrittura standard (stampante, video o entrambi) scelto per i tre valori suddetti, due su una riga e uno sulla successiva, ciascuno preceduto dall'identificatore della variabile che lo rappresenta nel nel programma; questi identificatori hanno lo scopo di chiarire al lettore il significato di ciascuno dei numeri.

Queste scritture esplicative qui appaiono piuttosto inutili, ma nei programmi che producono numerosi risultati che nelle diverse esecuzioni possono essere presentati diversamente hanno grande utilità in quabto contribuiscono a rendere il programma facile da usare e non vanno sottovalutate.

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 quanto vuole le scritture emesse, (cosa che in un programma con molti risultati non è scontata); quando l'utente ha battuto un carattere qualsivoglia il programma constata la sua immissione e pone fine all'esecuzione con la esecuzione della frase conclusiva `return(0);` .

B12 g.09 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 scegliere l'istante nel quale battere una risposta elementare.

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 concernente i dati immessi comporti la esecuzione di elaborazioni con dati intermedi e risultati finali non voluti, in genere completamente initili.

Dei dati erronei, ossia incoerenti con gli obiettivi del programmatore, potrebbero avviare elaborazioni del tutto inutili o, peggio, elaborazioni che portano a interpretazioni erronee 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 J12h e in J12i.

Inoltre la lettura di dati articolati e soggetti a vincoli di coerenza richiede validazioni che seguono logiche che vanno precisate con attenzione, che tengono conto delle caratteristiche dei significati applicativi dei dati; in genere conviene che questi non semplici controlli siano organizzati attraverso il richiamo di appositi sottoprogrammi.
Queste questioni saranno riprese nella sezione J12j.

B12 g.10 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 come lettere, cifre, spazi bianchi, delimitatori, separatori e segni di interpubzione.

Vi sono però vari caratteri ASCII, visualizzabili e non, che devono essere rappresentati da sequenze di altri caratteri chiamate **sequenze 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 alla successiva posizione multipla di 4
`\b` backspace, arretramento di una posizione
`\r` carriage return, passaggio a inizio linea successiva
`\f` form feed, inizia nuova pagina
`\a` alert, suono di avvertimento
`\\` backslash, carattere di inizio di ogni escape sequence
`\?` 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 bytes

B12 g.11 Il linguaggio C++ consente di formulare una ampia gamma di espressioni servendosi di operandi di diversi generi (costanti, variabili, componenti di arrays, puntatori), di diversi tipi di operatori e di parentesi tonde aventi come scopo primario quello 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-fp, 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 dà inizio a una scrittura di numero negativo o esprime 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].

B12 g.12 Ogni espressione matematica costituisce la richiesta di un complesso di operazioni (grosso modo di una sequenza di operazioni), ciascuna delle quali richiesta da una occorrenza di operatore il quale può agire sopra uno o due operandi oppure richiesta da una function con un certo numero di argomenti.

Il significato operativo di una espressione matematica consiste nell'insieme delle sequenze dei calcoli che essa può richiedere, ciascuna sequenza determinata da una possibile assegnazione di valori attuali ai suoi operandi.
Ciascuna sequenza di calcoli si identifica con la sequenza delle occorrenze di operatori che vengono successivamente eseguiti.

Per semplicità nel seguito invece di "esecuzione di una occorrenza di operatore" parleremo di "esecuzione di un operatore", come se gli operatori in una espressione fossero distinguibili indipendentemente dalla posizione che occupano nell'espressione.

L'esecuzione del primo operatore di una espressione orrisponde alla riduzione dell'espressione dovuta alla sostituzione dell'operatore e dei suoi operandi (oppure del richiamo di function) con il valore risultato del calcolo eseguito (oppure con il risultato primario del richiamo della function).
Con le esecuzioni dei successivi operatori si hanno analoghe riduzioni dell'espressione corrente; in ciascuna riduzione compre un nuovo operando che svolge il ruolo di operando intermedio.
L'esecuzione di una espressione con n operatori (o functions) corrisponde alla sequenza di $n+1$ espressioni, la prima essendo l'espressione in esame e l'ultima essendo il risultato dell'intero calcolo.

Occorre aggiungere che una espressione è da considerare corretta sse il suddetto procedimento di riduzione di può portare fino alla fine e quindi è effettivamente in grado di dornire un valore risultato; in caso contrario va considerata illegale o errata.

B12 g.13 Per una espressione nel linguaggio C++ conta quindi l'ordine di esecuzione degli operatori: questo viene determinato dalle cosiddette "regole di precedenza" (rispetto alla esecuzione).

La prima di queste regole richiede di considerare le coppie di parentesi coniugate che delimitano le sottoespressioni e i richiami di function nella espressione in esame e che consentono di individuare dei nidi all'interno dell'espressione; conviene anche supporre che l'intera espressione in esame sia delimitata da du parentesi coniugate.

Innanzi tutto questi nidi devono essere tali da determinare una loro organizzazione ad arborescenza: ogni nido è contenuto in un unico nido immediatamente più comprensivo e le parentesi che si incontrano nell'espressione costituiscono una stringa di Dick [].

Inoltre è stabilito che ogni operatore esterno a un nido N ed interno al solo nido immediatamente più comprensivo di N può essere eseguito solo dopo l'esecuzione di tutti gli operatori interni ad N.

Resta allora da stabilire quale eseguire per primo tra due cosiddetti **operatori contigui** operatori che nella espressione tokens consecutivi oppure sono separati solo da operandi, non da parentesi o da altri operatori.

Per questo intervengono le regole fissate dal linguaggio C++ e che chiamiamo **regole di precedenza tra operatori contigui**.

B12 g.14 Cominciamo ad esaminare le sole espressioni numeriche e le regole di precedenza degli operatori contigui aritmetici.
La precedenza maggiore riguarda `++`, `--` e `-` unario;
seguono gli operatori `*`, `/` e `%`;
infine la precedenza minore è quella degli opertori `+` e `-`.

Tra due operatori successivi della stessa precedenza viene eseguito per primo il più a sinistra.
Come si è detto, 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.

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 il risultato intermedio 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.

B12 g.15 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 18, numero chiamato quoziente; `144/8/2` vale 9, come `(144/8)/2`, mentre `144/(8/2)` rende disponibile 36, come `(144/8)*2`;

`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 possibilità di '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++ 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 $(\texttt{k/h}) * \texttt{h} + (\texttt{k\%h})$.

L'operatore resto si può usare liberamente nelle espressioni numeriche del linguaggio, ma richiede attenzione.

Ad esempio $\texttt{12\%5} + \texttt{40\%7} * \texttt{23\%8}$ si riduce alla $\texttt{2} + \texttt{5} * \texttt{7}$ e successivamente al risultato 37.

B12 g.16 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
```

B12 g.17 **(1) Eserc.** Esprimere le richieste per la individuazione e l'emissione, uno per riga, di tutti i prefissi di un numero di 4 cifre precedentemente letto.
(2) Eserc. Esprimere le richieste per il calcolo dell'area di un particolare rettangolo con i vertici aventi coordinate intere.

(3) Eserc. Esprimere le richieste per il calcolo del volume (positivo) di un parallelepipedo retto rettangolo i cui vertici sono esprimibili con coordinate intere; si assuma che il solido sia individuato da un vertice a dalle lunghezze dei suoi lati.

(4) Eserc. Esprimere le richieste per la emissione delle 3 permutazioni circolari della parola `tre` , delle permutazioni della parola `cancan` e delle permutazioni di una parola qualsiasi formata da 4 lettere diverse.

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

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

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

B12 g.18 Come si è detto, nel linguaggio C++ si trattano valori booleani costanti o variabili, cioè valori interpretabili come true e false che si possono combinare con valori interi.
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.

L'importanza dei dati booleani discende dalla possibilità di utilizzarli, come vedremo in :h e in :i, per decidere le scelte esecutive sulle quali si possono ottenere versatilità, adattabilità e 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 numeriche e, più in generale, su valutazioni quantitative riguardanti dati che possono essere numerosi ed etereogeneri.

B12 g.19 C++ e tutti i linguaggi procedurali dispongono di operatori relazionali su valori numerici, operatori binari che prevedono due operandi numerici interi o reali-fp 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 scambia 0 e 1.

Per quanto riguarda le precedenze esecutive, prevale l'operatore unario !, seguono gli operatori relazionali su numeri e l'operatore `&&`, mentre `||` ha la minima precedenza.
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 aritmetici, relazionale e logici di far uso di coppie di parentesi anche non strettamente necessarie.

B12 g.20 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, ossia una cella da 32 bits.

L'espressione `i < -2 || 10 <= i` vale `true` sse il valore attuale della `i` è inferiore a -2 oppure è maggiore o uguale a 10; essa quindi implementa la funzione indicatrice $\mathcal{I}_{\mathbb{Z}}[\mathbf{(} : -2\mathbf{)} \cup \mathbf{[}10 : \mathbf{)}]$ (consideriamo accettabile la limitatezza dei valori rattabili con le 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 da intendere come sottinsieme di $\mathbb{Z} \times \mathbb{Z}$.

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, per ogni coppia $\langle i, j\rangle$, il numero delle sottoespressioni relazionali delimitate tra parentesi tonde le quali risultano `true`.

B12 g.21 Nel linguaggio 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)

˜	···	0	1	···	9	···	A	···	Z	···	a	···	z	···
32	···	48	49	···	57	···	65	···	90	···	97	···	122	···

.

Presentazioni più complete si trovano in `J12c03` 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 ; endl // legge un carattere che supponiamo minuscolo
carmai = carmin - 'a' + 'A' ; // ottiene il corrispondente carattere maiuscolo
cout << carmai endl ; // lo emette
```

B12 h. strutture di controllo selettive

B12 h.01 Nelle linee di programma considerate finora sono intervenuti solo pochi e ben definiti componenti elementari del linguaggio: definizioni di costanti, dichiarazioni di variabili, assegnazioni che si servono di espressioni, frasi di I/O piuttosto semplici, richiami di sottoprogrammi generici o con un solo compito circoscritto, sequenze di pochi comandi.

Ora dobbiamo occuparci della redazione di programmi un po' più elaborati e con compiti un po' più articolati.

Cominciamo con il precisare che con il termine **controllo del programma**, in breve **controllo-p**, intendiamo una sorta di agente computazionale che è comodo descrivere in termini antropomorfi come persona che durante una esecuzione di un programma dirige e controlla l'esecuzione di ogni comando espresso nel programma "muovendosi sulle sue frasi" per far eseguire dai meccanismi del sistema computer costituito da hardware e software le operazioni che sono richieste dalle frasi che va toccando nei suoi 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; quindi non è stato necessario farlo intervenire.

Dato che le operazioni da eseguire per risolvere ogni istanza di problema risolvibile in tempi finiti sono in numero finito, ingenuamente si potrebbe pensare che ogni problema possa essere risolto con un controllo che si muove solo avanzando.
A questo si obietta che un programma di questo genere può affrontare solo problemi con istanze molto simili e quindi sarebbe un programma con una portata molto ridotta; l'esperienza dice invece che tutti i problemi di qualche importanza applicativa riguardano insiemi di situazioni che possono presentare forti differenze.

La maggior parte dei problemi di una certa consistenza riguarda insiemi di istanze che richiedono esecuzioni che possono essere molto differenti e che in genere sono difficilmente prevedibili a priori, ma si rivelano solo con una adeguata sperimentazione.

Ogni automatismo che intende essere efficace nella pratica deve essere in grado di prevedere tutte le situazioni nelle quali si può venire a trovare.

Procedimenti e programmi che intendono essere ampiamente applicabili devono essere capaci di distinguere le istanze dissimili e devono essere in grado di applicare ad esse manovre diverse.
Questa capacità di cogliere le caratteristiche delle situazioni da affrontare diversamente e di scegliere di conseguenza tra successivi diversi percorsi operativi costituisce un requisito che devono avere tutti i procedimenti e i programmi che aspirano a a potersi adattare senza grandi cambiamenti a istanze nuove rispetto a quelle sperimentate.
Questo conta soprattutto in tutti gli ambienti che si prevede possono essere toccati dalle innovazioni.
Nel tempo attuale si rende necessario disporre di programmi versatili, adattabili e in grado di evolversi; questo ha forti conseguenze sulle attività della programmazione e sulla evoluzione degli stessi linguaggi di programmazione.

B12 h.02 La distinzione delle situazioni che un programma può incontrare deve essere ricavata da parametri facenti parte dei dati iniziali o ricavabili 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 di 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 relativamente omogenei: questo è del tutto evidente per le problematiche che richiedono analisi statistiche su popolazioni numerose e sulle problematiche che richiedono valutazioni numeriche ottenibili procedendo servendosi di modelli via via più impegnativi e realistici e in particolare procedendo per approssimazioni numeriche 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 "blocchi del programma".

Facendo riferimento al controllo diciamo che è necessario che esso possa muoversi, oltre che con passi su frasi successive, anche effettuando scelte tra percorsi computazionali diversi e anche effettuando salti all'indietro verso frasi precedentemente toccate o appartenenti a percorsi precedentemente trascurati in seguito a scelte precedenti.

Si vuole dunque che il controllo possa effettuare vari tipi di salti sul testo sorgente, in particolare salti all'indietro per richiedere la ripetizione di certe azioni servendosi di nuovi valori per variabili importanti, valori ottenuti con azioni più recenti e in linea di massima "migliori" dei precedenti.

Si vuole in particolare la possibilità di organizzare iterazioni di sequenze di frasi esecutive il cui scopo è descrivibile come la **visita** di oggetti collocati in successivi punti di una figura geometrica dotata di regolarità al fine di esaminarli ed eventualmente di modificarli.

Va subito segnalato che si possono avere iterazioni con numeri di ripetizioni definiti in partenza e iterazioni le cui ripetizioni sono decise in conseguenza di dati raccolti nel corso di ciascuna ripetizione; questo accade ad esempio per molte costruzioni di risultati per approssimazioni successive. JR

B12 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 un'espressione logica, in particolare un'espressione relazionale o 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 precisa frase esecutiva del modulo di programma in esame che viene ad assumere il ruolo di possibile meta per la istruzione in esame e per altre frasi che comportano spostamenti del controllo-p.
Ciascuna di queste frasi 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 che nel testo segue l'attuale.

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, a sua volta conclusa da un segno "`:`".

Conviene anche avvertire che la possibilità del controllo di effettuare salti consente movimenti sul programma sorgente tanto articolati da portare a rischiare di avere programmi poco dominabili che possono aver comportamenti imprevisti.

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 i salti a frasi bersaglio.

B12 h.04 La possibilità di realizzare automatismi in grado di effettuare le varie manovre sopra accennate è stata presa in considerazione da tempo e meccanismi in grado di effettuare sequenze di operazioni, di scegliere tra successivi percorsi computazionali e di reiterare manovre sono stati concepiti fin dall'antichità.

Le prime realizzazioni di queste prestazioni sono state orrenute mediante strumenti meccanici, pneumatici, elettromeccanici e dell'elettronica analogica; tra queste realizzazioni ci limitiamo ad citare gli automi a controllo idrico e pneumatico dell'epoca ellenistica, il meccanismo di Anticitera, le calcolatrici di Pascal e Leibniz e i giocatori di scacchi del secolo XVIII.

Dal 1945 si sono imposti i dispositivi elettronici digitali che con la crescita esponenziale delle loro prestazioni (legge di Moore) hanno sostenuto la crescita delle attività informatiche e in particolare la crescita della programmazione.

Quando si potevano programmare gli elaboratori elettronici disponendo solo dei linguaggi di macchina (all'incirca dal 1945 al 1955) e dei primi linguaggi procedurali (il Fortran associato al nome di Backus e il COBOL dovuto a Grace Hopper) avvalendosi dei sopra accennati dispositivi di salto sono stati scritti parecchi programmi efficaci e di ragguardevoli dimensioni che sono riusciti a diffondere la convinzione dell'importanza che stavano assumendo il calcolo automatico e i gli strumenti per la elaborazione automatica delle informazioni.

B12 h.05 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 segnalato che l'adozione di programmi a esecuzione non sequenziale, oltre a estendere grandemente 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 ha comportato 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 intera classe di automatismi e in questo secondo caso viene detto "problema dell'arresto degli automatismi di uno specifico genere".

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 le applicazioni chiedono di controllare attività concrete, e quindi definite finitamente e che si basano su elaborazioni automatiche che dovrebbero concludersi in tempi finiti con l'impiego di risorse finite.

Tuttavia per soddisfare le richieste di applicazioni reali molto sentite si è giunti a proporre l'uso di strumenti che rischino di portare avanti le loro operazioni illimitatamente in condizioni di forte incertezza.

In altre parole si è spinti ad adottare strumenti con capacità operative potenzialmente infinite e questo comporta nuovi problemi, a cominciare da quello dell'arresto.
Evidentemente sarebbe stato ingenuo, e ora sappiamo illusorio, sperare di sfruttare appieno gli automatismi e la loro autonomia senza dover incontrare nuovi seri problemi.

B12 h.06 Negli anno dal 1955 al 1965 si sono sviluppati i primi computers (allora si chiamavano calcolatori elettronici o anche, suggestivamente, ordinatori); parallelamente e sinergeticamente si sono avuti dispositivi tecnologicamente innovativi e con prestazioni maggiori, ampliamenti delle applicazioni, crescita del numero delle macchine disponibili, nascita degli studi sulla programmazione e sulla matematica computazionale, crescita dell'importanza sociale e culturale del settore dell'elaborazione automatica diventato rapidmente un ecosistema.

Inoltre si è avuta una progressiva crescita dgli studi e dei dibattiti sopra i cambiamenti indotti; in particolare si è riscontrato il prevedibile aumento dei neologismi associati al computer (software, hardware, ...) e non è mancata l'indignazione di molti puristi delle lingue con forti tradizioni, come l'italiano.

La crescita della programmazione ha riguardato sia il numero dei programmi richiesti, messi a punto e fatti crescere, 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, dagli industriali, dai militari, dal settore della logistica e dal mondo 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.

B12 h.07 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, che poco si interrogavano sui criteri di organizzazione dei loro programmi e poco si confrontavano con i colleghi che avevano il compito di rielaborare i relativi testi sorgente.

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

In effetti il crescere della domanda di eleborazioni automatiche comportava spesso la richiesta, in genere accompagnata da urgenza, di ritoccare e ampliare le prestazioni di programmi in uso.
In molti casi si richiedeva di aggiungere a un programma porzioni di altri programmi, richiesta che spesso si soddisfaceva con l'inserimento di alcuni `goto` verso blocchi di frasi individuati affrettatamente.

Molti di questi `goto` avevano bersagli in posizioni poco facili da individuare e spesso le conseguenze delle loro aggiunte non venivano analizzate con sufficiente completezza.

Inoltre quando venivano attuate successive modifiche il controllo e le linee guida della organizzazione complessiva delle elaborazioni possibili tendevano a a deteriorarsi progressivamente.

B12 h.08 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 le modalità per la loro organizzazione secondo una certa rigidità ma che, se adottate come criteri finalizzati alla programmazione disciplinata e consapevole, portano alla disponibilità di programmi più leggibili, più controllabili, piĕstendibili, più adattabili 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 **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 e individuare abbastanza chiaramente consentono soluzioni chiare e poco rischiose; in questi casi qualche deroga dalla programmazione strutturata può essere conveniente.

B12 h.09 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 articolata che si rende necessaria solo se si riscontrano determinate condizioni.
Con le prestazioni 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.

B12 h.10 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 organizza 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 ciascuno dei quali chiaramente condizionato da una clausola che lo precede.

B12 h.11 Veniamo ora alla preannunciata nozione di blocco, che risulta centrale per la programmazione strutturata.

Per **blocco di istruzioni** si intende un segmento di programma delimitato con chiarezza, in particolare delimitato tra due parentesi graffe coniugate che richiede una manovra alla quale si può dare un chiaro significato.

Conviene distinguere vari tipi di blocchi.
Chiamiamo "blocchi di livello 0"' i moduli di programma caratterizzati da una premessa e da un corpo delimitato da una coppia di parentesi graffe.
Definiamo poi come blocchi primari o di livello 1 i blocchi interamente contenuti in un modulo.

Abbiamo infine i blocchi di livello superiore a 1 che sono porzioni di programma interamente contenuti in blocchi loro volta contenuti nel 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 chiamare "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.

B12 h.12 Aggiungiamo 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 di 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 un cosiddetto **scope**, cioè un ambito di validità, ossia di visibilità e operatività, che coincide con l'insieme delle frasi contenute nello stesso blocco.

Va detto anche che per un blocco semplice di 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, elementi che sono invece necessarie per le frasi che fanno da bersaglio per ogni frase `goto`.

Le caratteristiche strutturali dei blocchi contribuiscono a renderli delle unità operative dotate di rilevante autonomia, simile a quella dei sottoprogrammi; questi a loro volta si possono considerare blocchi particolari.

Come vedremo questo porta notevoli vantaggi alle 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 procedendo con blocchi successivi, da adattare al mutare delle esigenze, evento che in molti campi applicativi si verifica di frequente e talora è facilmente 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 pianificati e attentamente monitorati.

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.

B12 h.13 Nel corso di una elaborazione accade spesso che al controllo si pone la scelta tra tre o più successive diverse manovre alternative.
Per esempio si deve procedere con l'esecuzione di una prima manovra 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 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 1, né clausola 2
    }
```

Soluzioni analoghe facilmente precisabili vengono adottate per programmare 4 o più possibili scelte.

Per affrontare queste scelte tra diversi percorsi computazionali occorre esaminare l'insieme dei possibili percorsi successivi per 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 la corrispondente strada; 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`, ossia di "altrimenti".
Si noti che anche `else` è una **parola riservata** .

B12 h.14 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 nel testo sorgente.

Gli schemi di questi costrutti riguardanti 2 e 3 possibilità (in J12h08 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
    }
```

B12 h.15 Presentiamo, ancora sotto forma di frammenti di programma ridotti all'essenziale, alcuni esempi di costrutti con qualche commento che intende suggerire contesti applicativi concreti.

```
// Associa al progressivo del mese il numero dei suoi giorni
```

```
in un anno nonbisestile
if(prMese == 2) giorniN = 28 ;
else if(prMese==4 || prMese==6 || prMese==9 || prMese==11) giorniN = 30 ;
else giorniN = 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 ;
    }
```

B12 h.16 Come si è detto, in un blocco selettivo possono essere inseriti altri costrutti di selezione e in questi altri ancora; in questi casi si dice che vengono organizzate selezioni a più livelli.

Un primo esempio è dato dalla generalizzazione di una selezione in J12g11 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 ;
}
```

B12 h.17 (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 i casi di tre punti allineati e i casi di punti coincidenti.

(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 coincidenze.

B12 i. strutture di controllo iterative

B12 i.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.

B12 i.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 di volta in volta 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.

Questi gruppi di frasi, come quelli che sono oggetti di scelta nei costrutti selettivi, sono detti blocchi di frasi e sono caratterizzati formalmente dall'essere delimitati da coppie di parentesi graffe coniugate; va detto anche che questi delimitatori possono essere evitati per i blocchi che si riducono a una singola frase.

I blocchi di frasi che possono essere ripetuti più specificamente li chiamiamo **blocchi iterativi** e chiamiamo **ciclo [esecutivo]** ogni loro esecuzione.

il complesso delle azioni eseguite in un ciclo iterativo si può chiamare manovra reiterata.

Nei blocchi iterativi è possibile (e utile) richiedere azioni diverse nelle loro diverse esecuzioni, cioè nei diversi cicli iterativi.

In effetti servono blocchi iterativi dotati di rilevante versatilità che non devono essere soltanto pedissequamente ripetitivi; si deve avere la possibilità di organizzare cicli che si possono avvalere di tutti i meccanismi disponibili e quindi possano contenere tutte le scelte tra diverse alternative che possono servire.

B12 i.03 Per realizzare la versatilità dei cicli si possono adottare vari accorgimenti.

Innanzi tutto nei blocchi possono essere elaborate variabili con valori che cambiano nelle successive esecuzioni.

Un primo modo di procedere consiste nel servirsi di variabili globali o più in generale di variabili disponibili sia prima che all'interno del blocco iterativo e di prevedere di cambiare i loro valori prima di eseguire ogni ciclo.

Un secondo modo consiste nel porre nel blocco operazioni di lettura o richiami di functions in grado di fornire nuovi valori di variabili a ogni nuova esecuzione del blocco stesso.

Manovre interne al blocco che possono essere molto utili si ottengono con costrutti selettivi che dipendono da qualche dato variabile e quindi nei diversi cicli consentono di scegliere comportamenti diversi. Un elemento di differenziazione che può essere semplice ed efficace è il ricorso a richiami di functions accuratamente diversificati.

B12 i.04 Ogni linguaggio di programmazione che vuole essere versatile deve rendere disponibili dei **costrutti iterativi**;

Per avere programmi ben controllabili servono dei costrutti specifici per l'organizzazione dei blocchi iterativi, costrutti che si servono di parole chiave ed espressioni con il compito di controllare la sequenza delle esecuzioni delle frasi dal blocco.

Una esecuzione di tutti i cicli attualmente richiesti da un costrutto iterativo la chiamiamo **iterazione**.

I blocchi iterativi strutturalmente più semplici sono costituiti da una o più frasi esecutive, come al solito delimitate da parentesi graffe coniugate che possono essere trascurate nel caso di blocco 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.

Di fronte a questo tipo di costrutto il programmatore ha il problema di scegliere al meglio le azioni da prevedere nel ciclo; un consiglio spesso conveniente dice di avere cicli che prevedono richieste visibili con differenze contenute e con le eventuali differenze maggiori delegate a diverse functions richiamate o a sottoblocchi ben distinti.

B12 i.05 Nel linguaggio C++, come in gran parte 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 operazione di modifica, (che spesso 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 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 che viene esaminata alla fine di ogni

manovra reiterata e quelle rette da una clausola esaminata in un certa frase del blocco iterativo, ossia in qualche fase dell'iterazione.

B12 i.06 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 dati mensili di un qualche genere realizzati in un anno procedendo all'indietro, in una sorta di ritorno graduale 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 nonbisestili che 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 richiede attenzione e cautela, in quanto di comprensione non immediata e quindi possibile portatrice di 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 di numeri interi, 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.

B12 i.07 In generale un costrutto `for` presenta una assegnazione iniziale ad una variabile che di solito è intera, ma potrebbe anche essere una variabile reale-fp che svolge il ruolo di **indice del costrutto**.
Seguono una clausola da valutare 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 blocco iterativo; viceversa essa può mancare: questo accade quando si è provveduto a una inizializzazione dell'indice prima della comparsa di `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 si può assumere come indice dell'iterazione, in quanto tale ruolo lo vogliamo univoco.

Una clausola di iterazione può essere molto complessa e/o essere "incapsulata" in una function [J12e] richiamata nel blocco iterativo; in quest'ultimo caso la clausola può essere difficile da riconoscere nel testo del programma e questa poca visibilità 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 governate da complessi gruppi di frasi esecutive, in particolare possono venire incapsulate in una function.

B12 i.08 Presentiamo altri frammenti con costrutti `for`.

(1)
```
// dato un array int d[100] trovare i suoi valori minimo e massimo
// e le posizioni delle rispettive prime occorrenze
    int dMIN, dMAX, imin, imax; dmin = dmax = d[0] ; imin = imax = 0;
    for(int i = 1 ; i++ ; i<100) {
        if(d[i] < dMIN)  dMIN = d[i]; imin = i;
        if(d[i] > dMAX)  dMAX = d[i]; imax = 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
    int dTHMI, dTHMA; intdaccL=0;
    for(int i = 0 ; i++ ; i<1000) {
        if(dTHMi <= d[i] && d[i] <= dTHMA) {
            dacc[daccL] = d[i]; daccpos[daccL++] = i;
            }
```

B12 i.09 (3)
```
// 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 dei decili
    int id, datt;
    for(id=0 ; id++ ; id <10) decil[id] = 0;
    for(int i = 0 ; i++ ; i < dL) {
        datt=d[i];
        for(id = 0 ; id++ ; id < 10) {
            if(datt < 10*(id+1) {
                decil[id]++; break;
                }
            }
        }
```

In questo frammento compare la frase `break;` formata dalla parola chiave break 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 iterativo del quale la frase `break;` fa parte.

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 dalla parola chiave `switch`.

B12 i.10 (1)
```
// Lettura delle prime 365 o 366 componenti dell'array monodimensionale di interi
vi[] di 366 componenti
    int vi[366]; boolean bisest;
    for(int ivi = 0 ; ivi++ ; ivi < 365) {
        cin >> vi[ivi];
    }
    cin >> vi[365]; ]pqq bisest = OK
```

(2)
```
// esame di una stringa di caratteri diversi da ';' registrata in code[], stringa
        // che dovrebbe avere al più 40 caratteri e precisazione del loro numero nuc
    int nuc; char car_letto, code[40];
    for(nuc = 0 ; nuc++ ; nuc < 40) {
        cin >> car_letto ;
        if(car_letto == ';') break;
        code[nuc] = car_letto; }
        }
    if(nuc >= 41) {cout << "stringa troppo lunga" << endl ; }
```

B12 i.11 (1)
```
// Lettura da successive righe della tastiera standard 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 35 sono seguite da pseudosigla blank;
// determinazione del numero delle sigle siglN
// e loro collocazione nei primi siglN intervalli di 10 posizioni ciascuno
// facenti parte dell'array sigl[]
    char sigl[350], lett; int siglN, c;
    for(c = 0; c++; c < 350) sigl[c] = ' '; // predispone blanks in sigl[]
    for(siglN = 0 ; siglN++ ; siglN < 35) {// corsa sulle sigle
        forc = siglN*10; c < (siglN*10+10 c++;  {// corsa sui caratteri da leggere
            cin >> lett ;
            sigl[c] = lett;
            if(lett == ' ') break;
            }
        if(c == siglN*10) break;
        }
// 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
```

```
int isigl;
// scrittura per righe
for(siglN = 0 ; siglN++ ; siglN < 35) {// corsa sulle sigle
    if(sigl(siglN*10 == ' ') break;
    for(c=0; c++; c<10) {
        cout << sigl[siglN*10+c] ;
        cout << endl;
    }
}
// stampa per colonne
for(c=0; c++; c<10) {
    for (isigl = 0 ; isigl = isigl+10 ; isigl < siglN) {
        cout << sigl[isigl*10+c] ; cout << ' ';
    }
cout << endl;
}
```

B12 i.12 **(1) Eserc.** Scrivere un frammento di programma che genera e presenta le prime 30 potenze di 2.

(2) Eserc. Scrivere un frammento di programma che genera i primi 10 numeri fattoriali a partire dalla formula $n! = n \cdot (n-1)$.

(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.

(5) Eserc. Scrivere un frammento di programma che genera le permutazioni circolari dei 7 trigrammi dei giorni della settimana.

B12 i.13 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 che si basa sulla parola riservata `while` ha la forma che segue.

azioni di inizializzazione
`while(`*clausola per l'esecuzione di 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 ha il valore false.

B12 i.14 Esaminiamo un paio di programmi incentrati sul costrutto `while`.

(1)
```
// si abbia una sequenza di valori interi vi[i] per i < viN;
```

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

```
(2) // si abbia una sequenza di valori interi pos[i] per i < Npos;
    // raccogliamo in vic1[vic1N] i primi al più 10 valori che distano da pos1
    // meno di 20 in vic2[vic2N] i primi al più 5 valori che distano da pos2
    // meno di 25; vic1N e vic2N potrebbero essere inferiore a 10;
    // pos1 e pos2 si assume siano molto distanti.
    int Npos, ipos=0, pos[200], Nvic1=0, vic1[10], Nvic2=0, vic2[10];
    whileipos < posN {
        posatt = pos[ipos];
        if(vic1N < 10) {
            if(pos1-20 <= posatt &&posatt < pos1+20) {
                vic1[vic1N++] = posatt; ipos++;
                if(ipos <= posN) posatt = pos[ipos];
            }
        if(vic2N < 15) {
            if(pos2-25 <= posatt &&posatt < pos2+25) {
                vic2[vic2N++] = posatt; ipos++;
            }
        if(vic1N + vic2N >= 25) break;
    }
```

B12 i.15 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 di quella del costrutto `while`.

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

Esso provoca l'esecuzione di una prima manovra ciclica e successivamente, dopo l'esecuzione di ogni possibile successiva 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 < NviN
// sicuramente inferiori a 2000;
// poniamo in vinmadime il primo valore minore di 60, oppure,
```

```
// in mancanza di meglio, il minimo dei valori in vi[<viN]
    int vi[300], viN;
    lettura o costruzione di vi[0 ...  < viN]
    int vinmadime = 2000; posivi = -1; ivi=0;
    do {
        if(vi[ivi] < vinmadime) {
            vinmadime = vi[ivi];
            posivi = ivi;
            }
        }
    while(vinmadime >= 60) ;
    utilizzo di vinmadime
```

B12 i.16 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 nel blocco che segue 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 sia seguita dalla valutazione della clausola di iterazione successiva, in modo di evitare 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 corrente
    }
```

B12 i.17 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 J12i21, 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 di 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.

La conclusione della iterazione ottenuta da 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`

B12 i.18 **(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];
    }
```

B12 i.19 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, la quale ovviamente deve essere interrotta.

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 di una manovra ciclica non ultima
    }
```

L'argomento del `while` è sempre vero e vengono avviate sempre nuove esecuzioni della manovra ciclica fino a che si verificano le condizioni che implicano una esecuzione del comando `break`; il programma deve garantire che questo `break` si verifichi.

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 pesantemente 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.

B12 i.20 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 dall'argomento che segue `switch`.

Nelle soluzioni più semplici da leggere ciascuno dei successivi sottoblocchi esprime azioni da eseguire se per l'argomento dello `switch` si trova uno dei valori presentati all'inizio e si conclude con un comando `break` che implica l'uscita dal blocco del 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 dell'argomento dello 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 :  {    // dodecaedro
            numspig = 30; numfacce = 20; n1Schl = 3; n2Schl = 5; break; }
        case 20 :  {    // icosaedro
            numspig = 30; numfacce = 12; n1Schl = 5; n2Schl = 3; break; }
        }
```

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

B12 i.21 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 da un sottoblocco si può avere l'esecuzione delle operazioni previste da vari sottoblocchi che lo seguono, fino a quelle del primo blocco nel quale risulta 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 però in molti contesti consente di risparmiare molte ripetizioni di comandi.

B12 i.22 Nei programmi con costrutti nei quali vengono combinati più schemi dei tipi precedentemente introdotti si possono avere blocchi di programma che esprimono 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, vuoi per valutarne il valore tecnico o economico, vuoi 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 (velocità, versatilità, leggibilità, rapida adattabilità, ...).

Dalle diverse possibili esigenze applicative discendono i diversi pesi e le diverse desiderabili.

Questi parametri vanno riferiti alle qualità desiderabili per il prodotto programma quali garanzia di adeguatezza e di correttezza per le istanze prevedibili, velocità, efficienza esecutiva, facilità d'uso, versatilità e riutilizzabilità.

B12 i.23 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 lunghezze 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.
Si possono avere blocchi costituiti da uno solo dei costrutti selettivi o iterativi visti in precedenza e costrutti nei quali si individuano più 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 essere conveniente o meno risparmiare) si possono individuare strutture formali che vengono chiamate arborescenze distese; questi tipi di digrafi sono esaminati 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 sono strutture che è opportuno tenere ben controllate, tuttavia non sono le più generali; infatti si possono incontrare strutture contenenti anche composizioni con costrutti iterativi, anch'essi delimitati da proprie coppie di parentesi graffe evitabili per i blocchi di una sola frase.

B12 i.24 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 su sequenze di valori, 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.

Abbiamo quindi visto 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 sostanziale 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 successivamente 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 ancora più convenientemente blocchi già studiati e formalizzati in precedenza riutilizzabili con facilità; in caso contrario si incontrano sottoproblemi che non si sanno esprimere in termini di soluzione operativa.

Nel primo caso si sta ottenendo un programma ben strutturato in grado di risolvere il problema dato.

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 degli elementi che determinano il problema stesso e dei fattori che lo caratterizzano.

B12 j. organizzazione modulare dei programmi

B12 j.01 Come già segnalato, per sviluppare la maggior parte deiprogrammi applicativi incisivi servono ampie librerie di functions ed 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 più ampiamente diffusi e consolidati, costituisce un elemento di grande importanza per le attività di sviluppo del software e in particolare per la scelta del linguaggio da adottare per un progetto impegnativo.

La disponibilità di librerie di sottoprogrammi versatili e affidabili si può considerare come un essenziale 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 membri di estese comunità di programmatori.

Inoltre negli ultimi anni si sono resi disponibili piattaforme di intelligenza artificiale generativa in grado di fornire valido aiuto ai programmatori.

Le considerazioni che seguono 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 occupiamo solo di 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 nella *esposizione*.

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 di interesse generale, ossia riguardanti manovre che sono richieste da varie categorie di prodotti software.

Questa vicinanza si evidenzia soprattutto quando si considerino i sottoprogrammi per la soluzione di problemi computazionali come implementazioni di risultati matematici.

B12 j.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.

Molte librerie di sottoprogrammi vengono messe a punto dai programmatori che affrontano problemi specifici operando singolarmente o in gruppi di lavoro; altre possono essere acquisite da fornitori specializzati che rendono disponibili prodotti software di largo interesse o su misura; altre ancora sono reperibili nei siti del Web finalizzati alla messa a disposizione di software libero secondo gli ideali dell'open content.

Nel linguaggio C++ vengono rese disponibili effettivamente molte librerie di functions concretamente agganciabili mediante frasi `#include` .

Questi sono enunciati di un preciso tipo detto tipo dei **comandi per il preprocessore**; enunciati caratterizzate dal fatto di iniziare con il carattere `#`, e dal fatto che ciascuno di essi occupa una linea del sorgente del modulo nel quale viene usato, prima dell'intestazione del modulo che inizia con la parola riservata `main` o con la dichiarazione di function.

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

B12 j.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 i caratteri visualizzabili, direttamente o meno, da un carattere `null` che segnala la conclusione della stringa stessa.

Una tale stringa la chiamiamo **stringa-nt** e conviene segnalare esplicitamente che una stringa-nt 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 solo 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 lettera minuscola nella corrispondente maiiuscola.

`sprintf(...)` Costruisce una stringa visualizzabile a partire da argomenti che nei diversi richiami possono avere diverse forme e riguardare dati in numero diverso e di diversi tipi.

B12 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 semplicemente nella precisazione e 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`.