Dispense di Laboratorio di Informatica
II Parte
A.A. 2025/26

M. Baioletti

November 21, 2025

Chapter 6

Linguaggio C++ e istruzioni elementari

6.1 Introduzione a C++

Il linguaggio C++ è un linguaggio di programmazione imperativo orientato agli oggetti, derivato dal linguaggio C (che contiene come sottoinsieme) ed esteso in vari modi.

Inizieremo questa introduzione con un esempio di programma completo: un programma che calcola l'area di un triangolo, leggendo da tastiera la base e l'altezza.

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

int main() {
    double base, altezza, area;
    cout << "inserisci la base ";
    cin >> base;
    cout << "inserisci l'altezza ";
    cin >> altezza;
    area = base * altezza / 2;
    cout << "l'area e' " << area << endl;
}
```

Commenteremo ora il programma linea per linea.

1. L'istruzione `#include` serve ad includere nel programma la definizione di funzioni e di altri oggetti utilizzati nel programma. Nell'esempio,

`iostream` è la libreria che contiene la definizione delle operazioni di input/output. Senza includere `iostream`, un programma non può leggere dati da tastiera o scrivere sullo schermo. In un programma possono esservi altre `#include`.

2. L'istruzione `using namespace std` semplifica l'uso di cout, cin e endl. Senza di essa, si dovrebbe scrivere `std::cout`, `std::cin` e `std::endl`.

3. Il programma vero e proprio è all'interno della funzione "main", racchiuso tra parentesi graffe.

4. La prima istruzione `double base, altezza, area` dichiara tre variabili di nome *base*, *altezza* e *area*, di tipo `double`, ovvero contengono numeri reali.

5. La seconda istruzione `cout << "inserisci la base "` scrive sullo schermo la frase in questione.

6. La terza istruzione `cin >> base` legge dalla tastiera il valore della variabile base.

7. La quarta e la quinta istruzione sono analoghe alla seconda e alla terza.

8. La sesta istruzione calcola il prodotto tra la base e l'altezza, lo divide per 2 e ne assegna il risultato alla variabile *area*.

9. La settima istruzione infine scrive sullo schermo la frase `l'area e'` , poi il valore della variabile *area* e infine va a capo (`endl` sta per endline).

Le istruzioni di un programma sono eseguite di solito dall'alto verso il basso.

6.2 Tipi di dati

Il linguaggio C++, al pari degli altri linguaggi di programmazione, lavora con dati che sono classificati per tipo di dato. Un **tipo di dato** è definito mediante un insieme di possibili valori (detto **dominio** del tipo) e un insieme di operazioni elementari. I tipi di dato si dividono in **tipi elementari** e **tipi strutturati**. Ci occuperemo dei dati strutturati in un capitolo successivo. I tipi elementari di C++ sono

1. tipi numerici interi: `int`, `long`, `long long`, `short`, `char` e loro versioni `unsigned`

2. tipi numerici reali: `float`, `double`, `long double`
3. tipo booleano: `bool`
4. tipo stringa: `string`

6.2.1 Numerici interi

I tipi numerici interi sono rappresentati in complemento a due. I più utilizzati sono gli `int`, che normalmente sono a 32 bit, mentre gli `short` hanno un numero minore bit, e i `long` ne hanno un numero maggiore. Infine i `char` sono a 8 bit. Di ognuno di questi, esiste anche la versione `unsigned` che non ammette numeri negativi (e quindi può contenere numeri più grandi). I dati possono essere inseriti non solo in base 10, ma, con particolari accorgimenti, anche in base 8 o in base 16. Esistono 5 operazioni aritmetiche elementari

`+` addizione

`-` sottrazione

`*` moltiplicazione

`/` quoziente della divisione

`%` resto della divisione

Ad esempio `17 / 4` fa 4, mentre `17 % 4` fa 1.

C++ supporta anche operazioni sui bit (`& | ~ ^ << >>`) che qui non tratteremo.

Un dato di tipo char può contenere un singolo carattere ASCII. Nei programmi i caratteri singoli sono indicati tra apici, ad esempio `'A'`.

6.2.2 Tipi numerici reali

Esistono tre tipi di dato reali: `float`, `double` e `long double`, che sono rispettivamente a 4, 8 e 16 byte.

La rappresentazione usata è quella a virgola mobile, quindi i `float` hanno una grandezza della mantissa e dell'esponente minore rispetto ai `double`, che a loro volta hanno mantisse e esponenti più corti dei `long double`.

I numeri reali si possono introdurre in notazione decimale (con il punto al posto della virgola). Ma si può anche usare la notazione scientifica, comoda per esprimere numeri molto grandi, come $3.7 \cdot 10^{56}$ che si scrive `3.7e+56`, o numeri molto piccoli, come $2 \cdot 10^{-17}$, che scrive `2e-17`. I numeri reali ammettono le quattro operazioni `+`, `-`, `*` e `/`.

Le principali funzioni con numeri reali in C++ sono

- `exp`, esponenziale
- `log`, logaritmo naturale
- `log10`, logaritmo in base 10
- `sin`, `cos`, `tan`, con argomento in radianti
- `sqrt`, radice quadrata
- `pow`, elevamento a potenza con due argomenti, ad esempio 7.2^5 si calcola con `pow(7.2, 5)`
- `abs`, valore assoluto
- `floor` e `ceil`, l'intero più vicino all'argomento, rispettivamente, per difetto e per eccesso.

Queste funzioni richiedono obbligatoriamente le parentesi che racchiudono gli argomenti, ad esempio `sqrt(3.2)`.

Inoltre, per utilizzare le funzioni bisogna includere la libreria `cmath` mediante l'istruzione

```
#include <cmath>
```

In tale libreria, il valore di π è disponibile con `M_PI`, mentre la costante di Nepero e si ottiene con `M_E`.

6.2.3 Tipo booleano

Il tipo di dato `bool` ha come possibili valori `true` e `false`, rappresentati internamente come 1 e 0, rispettivamente.

Una **condizione** è un'espressione è di tipo `bool`, ovvero il cui risultato è `true` o `false`.

Le condizioni elementari si possono formare mediante gli operatori di confronto, i quali si possono applicare sia a numeri, sia a stringhe:

- `>`
- `<`
- `>=`, che corrisponde a $\geq$,
- `<=`, che corrisponde a $\leq$

- `==`, che è il simbolo dell'uguaglianza.
- `!=`, che corrisponde a $\neq$

Ad esempio se a vale 3 e b vale 7, allora la condizione `a < b` vale `true`, mentre la condizione `a * b == 4` vale `false`. Le condizioni possono essere composte con gli operatori booleani `&&` (che corrisponde a AND), `||` (che corrisponde a OR) e `!` (che corrisponde a NOT).

Riportiamo qui le tabelle di verità corrispondenti, che sono perfettamente analoghe a quelle dell'algebra di Boole

C1	C2	`C1 && C2`	`C1 || C2`	`! C1`
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Quindi

- `C1 && C2` è vera solo se entrambe le condizioni C1 e C2 sono vere (congiunzione), ad esempio `(a > b) && (b > 3)` è falsa, mentre `(a < b) && (b == 7)` è vera
- `C1 || C2` è vera se almeno una delle due condizioni C1 e C2 è vera (disgiunzione), ad esempio `(a > b) || (b > 3)` è vera, mentre `(a + b < 5) || (b < 7)` è falsa.
- `! C1` è vera solo se C1 è falsa (negazione), ad esempio `! (a > 5)` è vera, mentre `! (b - a == 4)` è falsa.

Dato che `&&` e `||` sono associative, è possibile usare congiunzioni e disgiunzioni multiple:

- `C1 && C2 && ... Cn` è vera se solo se tutte le condizioni C_i sono vere
- `C1 || C2 || ...|| Cn` è vera se almeno una delle condizioni C_i è vera.

Ad esempio, per controllare se un anno a è bisestile, è necessario controllare che

1. a sia divisibile per 4 e
2. a non sia divisibile per 100 oppure
3. a sia divisibile per 400

Infatti la regola dice che gli anni non divisibile per 4 non sono bisestili, ad esempio 2011, e invece generalmente quelli divisibili per 4 lo sono, ad esempio 2012. Ma gli anni che iniziano il secolo non lo sono, ad esempio 1900, tranne quelli divisibili per 400, come 2000, che lo sono.

La condizione che controlla se a è bisestile è quindi

```
a % 4 == 0 && (a % 100 != 0 || a % 400 == 0)
```

Per controllare se un numero x appartiene ad un intervallo chiuso $[a, b]$, con $a < b$, si può usare la seguente condizione

```
(x >= a) && (x <= b)
```

Si noti che `a <= x <= b` non funziona correttamente perché verrebbe interpretata come `(a <= x) <= b`, cioè controlla se il risultato del confronto `a <= x` (che in ultima analisi corrisponde a 0 o 1) è minore di b. Ad esempio per $a = 1$, $b = 4$ e $x = 0$ darebbe come risultato `true`, anziché `false`.

Negli ultimi standard del linguaggio è anche possibile usare `and` al posto di `&&`, `or` al posto di `||` e `not` al posto di `!`.

6.2.4 Tipo stringa

Le stringhe sono sequenze finite di caratteri. Si tratta di un tipo di dato essenziale per l'elaborazione di dati non numerici. In un programma un valore di tipo stringa si scrive tra virgolette, ad esempio `"scienza"`.

Tra le principali operazioni delle stringhe è doveroso citare la concatenazione. Tale operazione, che si indica con il simbolo $+$, "attacca" tra di loro due stringhe. Ad esempio `"matem" + "atica"` fa `"matematica"`.

Si noti che la concatenazione non è commutativa e che la stringa vuota `""` ne è l'elemento neutro. Vedremo maggiori dettagli sulle stringhe nel capitolo relativo ai tipi di dati strutturati.

6.2.5 Tipi e sottotipi

Tra i tipi elementari numerici esiste la relazione di sottotipo: T1 è sottotipo di T2 se è possibile convertire direttamente un valore di T1 ad un valore di T2. Matematicamente, il dominio di T1 è un sottoinsieme di quello di T2.

Ad esempio, gli short sono un sottotipo degli int, dato che ogni valore di tipo short può essere convertito in un valore di tipo int. Infatti la differenza tra short e int è solo nel numero di bit disponibili. La relazione di sottotipo (indicata con $\prec$) crea la seguente gerarchia

$$\text{char} \prec \text{short} \prec \text{int} \prec \text{long} \prec \text{float} \prec \text{double} \prec \text{long double}$$

6.3 Variabili, costanti ed espressioni

6.3.1 Variabili

Le variabili sono piccole parti di memoria centrale in cui il programma è in grado di memorizzare un valore per poi riutilizzarlo in seguito. Le variabili sono identificate da un nome, che deve iniziare con una lettera alfabetica e deve essere composto solo da lettere, cifre e il simbolo di sottolineatura ("_"). Ad esempio sono nomi validi di variabili "z", "x1", "a2b", "tasso_di_interesse".

I nomi possono essere usati per identificare anche altri oggetti all'interno di un programma: in generale si chiamano **identificatori**.

Ogni variabile X è associata ad un tipo di dato T: solo valori di tipo T possono essere memorizzati in X. In C++ è obbligatorio dichiarare le variabili utilizzate in un programma. La **dichiarazione di una variabile** consiste nell'indicare il tipo, il nome ed un eventuale valore iniziale. Ad esempio con

```
int a;
double x1;
bool k;
```

si dichiarano tre variabili: a di tipo `int`, $x1$ di tipo `double` e k di tipo `bool`. Con

```
int b = 11;
```

si dichiara una variabile b di tipo `int`, il cui valore iniziale è 11.

Si possono accorpare le dichiarazioni di variabili dello stesso tipo, ad esempio con

```
int u,v,z;
```

si dichiarano tre variabili di tipo `int`.

In C++ una variabile può essere utilizzata in un programma solo dopo essere stata dichiarata. Ad esempio

```
int main() {
    // prima parte
    ...
    int a;
    // seconda parte
    ...
}
```

nella prima parte del programma la variabile a non può essere usata, mentre nella seconda parte sì.

6.3.2 Costanti

E' possibile definire una costante simbolica attribuendole un nome e un valore che non può essere alterato. Un modo rudimentale, ma ancora molto diffuso, è usare l'istruzione `#define`, la cui sintassi è

```
#define NOME_COSTANTE valore
```

Ad esempio ecco come definire la costante $\hbar$ (che è la costante di Planck divisa per 2π)

```
#define H_BAR 1.0545718e-34
```

Quindi, si può usare `H_BAR` per ottenere il valore della costante $\hbar$ in tutto il resto del programma.

Si noti che essendo un'istruzione del preprocessore (come `#include`), non ha bisogno del punto e virgola finale.

E' prassi che i nomi delle costanti siano scritti in maiuscolo, per distinguerli dalle variabili e dagli altri elementi del programma.

Un modo più moderno per dichiarare è usare constexpr. La sintassi è

```
constexpr tipo_costante NOME_COSTANTE = valore
```

La costante $\hbar$ (che è la costante di Planck divisa per 2π) è dichiarata con

```
constexpr double H_BAR = 1.0545718e-34;
```

Le due modalità sono simili, ma la seconda è più moderna. Se un compilatore non supporta la versione C++11, al posto di `constexpr` si può usare `const`.

6.3.3 Espressioni

Le espressioni possono comparire in varie parti di un programma. Esse denotano un valore che deve essere calcolato mediante il processo di valutazione. In un'espressione possono comparire

- costanti letterali, ad esempio 5, -7.1, "ciao", false
- variabili
- operatori, ad esempio +, *
- funzioni, ad esempio `sqrt`, `abs`.

Nelle espressioni si possono usare anche le parentesi tonde per alterare l'ordine di valutazione. Ad esempio mentre `3 + 4 * 5` fa 23, `(3 + 4) * 5` fa 35. Si noti che si possono solo usare le parentesi tonde, anche in forma ripetuta, ciò infatti non crea problemi di ambiguità.

Il valore di un'espressione dipende dal valore delle variabili che vi compaiono, quindi la stessa espressione può assumere valori diversi in momenti diversi del programma.

Ad esempio se la variabile a vale 5 e la variabile b 17, allora l'espressione `a * (b - 10)` vale 35. Mentre nel caso in cui a vale 6 e b vale 13, `a * (b - 10)` vale 18. In virtù del fatto che le variabili, le costanti e i risultati delle funzioni e degli operatori hanno un tipo prestabilito, si può stabilire in modo univoco qual è il tipo T del valore di un'espressione. Si dice in questo caso che l'espressione è **di tipo T**.

Ad esempio se

```
int a,b,c;
double x,y,z;
string s,t;
```

- le espressioni `a + b`, `2 * c`, `a % 4 + a / 4` sono di tipo `int`,
- le espressioni `x / y`, `3.1 * z - x * y` e `sqrt(x) * sin(y)` sono di tipo `double`,
- le espressioni `s + t` e `t + "ciao"` sono di tipo `string`.

Se in un'espressione compaiono numeri di tipo diverso, ad esempio `3 * z + a * y`, allora la valutazione procede promuovendo tutti i numeri interi a numeri reali e svolgendo le operazioni come se gli operandi fossero di tipo `double`. Ovviamente, il risultato sarà di tipo `double`.

In generale, la valutazione di un'espressione del tipo

x OP y

in cui OP è un operatore binario mentre x e y sono numeri di tipo diverso, di cui uno è un sottotipo dell'altro, avviene portando gli operando al tipo più generale. Ad esempio se x è di tipo `short` e y è di tipo `long`, x è convertito in `long`. Le conversioni da sottotipo a tipo sono quindi fatte automaticamente dal C++.

La conversione inversa da un tipo numerico ad un suo sottotipo è invece un'operazione che potrebbe essere impossibile o far perdere informazione.

Ad esempio è impossibile convertire esattamente il numero 123456789 al tipo `short` (a 16 bit), in quanto il massimo intero rappresentabile in quest'ultimo tipo di dato è 32767.

Invece la conversione da `double` a `float` fa perdere le ultime cifre decimali: convertendo il numero 4.25167536214367 in `float` si produce il numero 4.25168.

Perciò, in C++ tali conversioni devono essere richieste esplicitamente nel programma mediante gli operatori di cast. Il modo più semplice, valido anche per il linguaggio C, è quello di anteporre al dato il tipo (scritto tra parentesi) a cui lo si vuole convertire. Ad esempio per convertire il valore di una variabile x di tipo double in int si scrive `(int) x` Se x vale 2.0, il risultato di tale operazione sarà il numero intero 2.

Un altro metodo alternativo, più moderno, è scrivere `int(x)`. Un modo più sofisticato e ancora più moderno è `static_cast<int>(x)`.

6.4 Assegnamento

L'assegnamento ha come forma elementare

$$\mathrm{x} = \mathrm{e}$$

in cui x è l'identificatore di una variabile ed e è un'espressione. Un assegnamento è valido se l'espressione e è dello stesso tipo di x o di un sottotipo di quello di x. Ad esempio sono assegnamenti validi

```
int a,b,c;
double x,y,z;
string s,t;

a = 4;
b = a + 3;
c = 7 * (a + b);
x = 1.0;
y = sqrt(x) - 7;
z = 4 * x;
```

Si noti che nell'espressione e può comparire anche x stesso. Ad esempio, partendo da

```
a = 5;
```

se si esegue l'istruzione

```
a = a +1 ;
```

a diventa 6, dato che l'espressione $a + 1$ ha proprio tale valore.

Una caratteristica importante da capire è che l'assegnamento non fa altro che assegnare alla variabile x il valore che in quel momento possiede l'espressione e. Qualunque successiva operazione che altera il valore di e non avrà alcun effetto su x. Ad esempio, partendo da

```
a = 7;
```

se si assegna a b il valore di $a + 1$

```
b = a + 1;
```

quando si richiede il valore di b si ottiene 8. Anche quando si cambia il valore di a con

```
a = 5;
```

il valore di b rimane 8.

Un'altra proprietà interessante è che l'assegnamento non è commutativo. Sia infatti l'espressione assegnata composta da una singola variabile. Ad esempio si prendano due variabili a e b, allora gli assegnamenti `a = b` e `b = a` hanno effetti diversi. Infatti, se

```
a = 7;
b = 4;
```

l'istruzione

```
a = b;
```

modifica a facendola diventare uguale a b: ora a e b hanno entrambe il valore 4. Mentre partendo dalla stessa situazione precedente (a=7 e b=4) con l'istruzione

```
b = a;
```

si ottiene che sia a che b assumono il valore 7.

Si noti che l'operatore di uguaglianza si scrive con il doppio uguale per distinguerlo dall'operazione di assegnamento, che viene indicata con l'uguale singolo. Infatti `a = b` significa che la variabile a diventa uguale alla variabile b (assume il valore corrente di b), mentre `a == b` si chiede se a abbia lo stesso valore di b.

6.5 Istruzioni di input/output

In questa sezione vedremo le istruzioni di input e di output su terminale, ovvero lettura da tastiera e scrittura sullo schermo. Queste istruzioni possono essere generalizzate alla lettura e scrittura su file, argomento che tratteremo in seguito.

6.5.1 Istruzione di input

Per leggere dei dati da tastiera si usa l'istruzione

```
cin >> variabile
```

o con più variabili

```
cin >> variabile1 >> ... >> variabileN
```

`cin` sta per input da console (terminale). Si noti che l'operatore di lettura `>>` deve essere ripetuto per ogni variabile che si intende leggere.

L'operazione di lettura ferma il programma e aspetta che l'utente inserisca almeno N dati (ove N è il numero di variabili coinvolte nella lettura) separati da uno o più spazi e prema invio.

Se i dati non sono sufficienti, ad esempio se si vogliono leggere quattro variabili ma l'utente inserisce solo tre valori, il programma aspetta fino a che l'utente non inserisce almeno un quarto dato e preme nuovamente invio. Se i dati sono troppi, ad esempio con quattro variabili coinvolte l'utente inserisce cinque valori, i valori in eccesso saranno usati per le successive istruzioni di lettura.

In conclusione, quando l'utente ha inserito un numero sufficiente di dati il programma riparte memorizzando in ciascuna variabile il corrispondente valore introdotto da tastiera. Ad esempio in

```
int a;
string t;
double x;
cin >> a >> t >> x;
```

se l'utente digita

```
5 ciao 3.1 8
```

il programma riparte assegnando 5 alla variabile a, "ciao" a t, 3.1 a x e conservando il valore 8 per la successiva istruzione di lettura.

6.5.2 Istruzione di output

Per scrivere dati sullo schermo si usa l'istruzione

```
cout << dato1 << dato2 << ... << datoN
```

I dati possono essere una qualsiasi espressione, di cui viene scritto il valore. Si noti che l'operatore di scrittura `<<` deve essere ripetuto per ogni dato da scrivere. `cout` sta per output da console (terminale).

I dati sono sempre scritti sullo schermo da sinistra verso destra. Ad esempio, se n è una variabile int di valore 4, allora

```
cout << "il doppio di " << n << " e' " << 2 * n;
```

scrive sullo schermo

```
il doppio di 4 e' 8
```

E' possibile andare a capo usando `endl` come dato da scrivere. Ad esempio

```
cout << n << endl << 2 * n << endl << 3 * n;
```

scrive sullo schermo

```
4
8
12
```

In alternativa è possibile usare il carattere `\n` all'interno di una stringa. Ad esempio

```
cout << "coefficiente\n\n" << n << "x";
```

scrive

```
coefficiente

4x
```

I dati booleani sono normalmente scritti come numeri interi (0 per false e 1 per true), ma con `boolalpha` il prossimo dato di tipo `bool` sarà scritto per esteso. Ad esempio

```
bool b=true;
cout << boolalpha << b << endl;
```

scrive sullo schermo

```
true
```

E' possibile scrivere un dato indicando esattamente quanti caratteri impiegare mediante `setw`. Gli eventuali caratteri mancanti saranno riempiti con degli spazi. Ad esempio

```
int a=4, b=11;
cout << a << setw(7) << b << endl;
```

scrive

```
4     11
```

con 5 spazi tra 4 e 11 (infatti b è stato scritto con esattamente 7 caratteri).

Normalmente C++ scrive i numeri usando al massimo 7 cifre dopo la virgola. Per aumentare o diminuire tale valore si può usare `setprecision`. Ad esempio

```
int main() {
    double a=1.0/3;

    cout << a << endl;
    cout << setprecision(4) << a << endl;
    cout << setprecision(6) << a << endl;
    cout << setprecision(8) << a << endl;
}
```

scriverà sullo schermo

```
0.333333
0.3333
0.333333
0.33333333
```

Per usare `setw` e `setprecision` è necessario includere la libreria `iomanip`.

6.6 Convenzioni lessicali

Quando si scrivono le istruzioni nei programmi in C++, occorre tenere in mente le seguenti convenzioni.

Innanzitutto, C++ è un linguaggio *case-sensitive*, ovvero la differenza tra lettere maiuscole e minuscole è significativa. Ad esempio le variabili a e A sono distinte, perciò se si assegna un valore ad a, si ottiene un messaggio di errore quando si tenta di valutare A. Anche le istruzioni e i nomi delle funzioni tengono conto di maiuscole e minuscole. Ad esempio la funzione coseno si chiama cos scritta in minuscolo, non esiste una funzione chiamata Cos, COS o coS. Allo stesso modo l'istruzione if, ad esempio, deve essere scritta solo in minuscolo, altrimenti si genera un errore di sintassi.

Tutte le istruzioni, tranne rare eccezioni, sono concluse dal punto e virgola. Tra le prime eccezioni segnaliamo le istruzioni del preprocessore che iniziano con il carattere #.

E' norma mettere una sola istruzione per riga. Se però accade che un'istruzione è troppo lunga per entrare in una riga dello schermo, la si può spezzare in qualsiasi punto

```
x=a * 2 + b * 2 + c * 2 + d * 2 + e * 2 +
   f * 2 + g * 2 + h * 2;
```

Inoltre è possibile scrivere più istruzioni sulla stessa linea, anche se ciò ne riduce la leggibilità.

```
a = 3; b = c + a - 7; d = (a + b) % 3;
```

La presenza di spazi, tabulazioni, andate a capo e linee vuote è consentita all'interno di un programma ed è anzi vivamente consigliata per aumentarne la sua leggibilità.

Infine è possibile inserire un commento all'interno di un programma C++, in due modalità distinte.

Il primo modo è usare `//`. In tal caso il commento finisce a fine linea. Il secondo modo è racchiudere il commento tra `/*` e `*/`, in tal caso il commento può essere arbitrariamente lungo. Ad esempio

```
int main() {

    // commento su una sola linea

    /* commento
    su
    piu' linee */

}
```

Un commento può contenere qualsiasi sequenza di caratteri e termina a fine riga. Di solito nei commenti sono inserite delle frasi di descrizione o di chiarimento del programma. I commenti possono essere scritti anche in italiano (o in qualsiasi lingua), dato che il compilatore ignora completamente il contenuto.

6.6.1 Spazi e indentazione

Il compilatore ignora gli spazi all'interno di un programma (per spazi si intendono i caratteri di spaziatura, tabulazione e andata a capo), tranne nel caso in cui si trovano consecutivamente due tra identificatori e parole chiavi. Ad esempio in `int x` lo spazio tra int e x è obbligatorio.

Gli spazi all'interno di un programma servono in gran parte per migliorare la leggibilità del codice. Infatti, anche se è perfettamente possibile scrivere

```
a=1;b=a+2;if(a<3)c=b-1;else c=x+y;z=4;
```

il codice risultante è completamente illeggibile.

Le istruzioni all'interno dei costrutti, come le istruzioni composte (blocchi, strutture condizionali e iterative) e i corpi delle funzioni, sono scritte tutte incolonnate, ma più a destra rispetto alle altre. Tale accorgimento si chiama indentazione e migliora la leggibilità del programma perché così si vede a colpo d'occhio che le istruzioni sono "interne" al costrutto in questione. L'indentazione deve crescere all'aumentare della profondità di annidamento, cioè quando ci sono costrutti dentro altri costrutti, come nel caso di

```
if(x == 3) {
    a = 4;
    while(c < 1) {
        c = c * 7;
        b++;
    }
}
```

6.7 Esercizi

1. Scrivere un programma che legge il raggio r di una circonferenza e ne calcola l'area e la lunghezza.

2. Scrivere un programma che calcola l'area e il perimetro di un triangolo a partire dalle misure dei tre lati a, b, c. Per l'area si usi la formula di

Erone

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

in cui p è il semiperimetro.

3. Scrivere un programma che legge i coefficienti a e b di un'equazione di primo grado $ax = b$ e ne scrive la soluzione (supporre che $a \neq 0$).

4. Scrivere un programma che calcola il determinante di una matrice 2x2 a partire dai suoi 4 elementi.

5. Scrivere un programma che calcola il determinante di una matrice 3x3 a partire dai suoi 9 elementi.

6. Scrivere un programma che calcola le soluzioni di un sistema lineare di due equazioni in due incognite a partire dai 4 coefficienti e dai 2 termini noti.

7. Scrivere un programma che calcola il prodotto $u + iv$ di due numeri complessi $a + ib$ e $c + id$ prendendo come input a, b, c, d e producendo come risultato u, v.

8. Scrivere un programma che legge da tastiera un numero naturale compreso tra 100 e 999 e calcolare le tre cifre di cui è composto.

9. Scrivere un programma che risolve un problema di geometria: leggere i dati del problema, calcolare e scrivere sullo schermo i risultati. Ad esempio, leggere i coefficienti e i termini noti di due rette non parallele nel piano e calcolare il loro punto di intersezione.

10. Scrivere un programma che risolve un problema di fisica: leggere i dati del problema, calcolare e scrivere sullo schermo i risultati.

11. Scrivere un programma che legge i coefficienti a, b, c di un'equazione di secondo grado $ax^2 + bx + c = 0$ e ne scrive le soluzioni (supporre che il delta sia maggiore o uguale a zero).

12. Scrivere un programma che legge il capitale C, il tasso di interesse i ed il tempo in anni t di un prestito e calcola il montante sia in regime di capitalizzazione semplice $M_s = C(1 + it)$, sia quello in regime di capitalizzazione composta $M_c = C(1 + i)^t$.

13. Scrivere un programma che legge da tastiera due variabili intere a e b e mediante una serie di assegnamenti, scambia i due valori. Suggerimento: usare una terza variabile di "appoggio".

14. Scrivere un programma che legge tre variabili intere a, b, c e "ruota" i valori, in modo che b assuma il valore di a, c quello di b e a quello di c.

Chapter 7

Istruzioni condizionali

Le **istruzioni condizionali** permettono ad un programma di prendere decisioni e di eseguire istruzioni diverse in condizioni diverse. In C++ sono presenti due istruzioni condizionali: `if` e `switch`.

L'istruzione condizionale di base, presente in tutti i linguaggi di programmazione, è l'istruzione `if`. In C++, come del resto in molti linguaggi, assume due forme, la prima che comprende anche la parte `else` e la seconda forma senza la parte `else`.

7.1 If-else

In C++ l'istruzione *if-else* ha la seguente sintassi

`if`(C) S1 `else` S2

ove C è una condizione e S1, S2 sono due singole istruzioni o blocchi.

Un **blocco** è una sequenza arbitraria di istruzioni racchiuse tra parentesi graffe. Un blocco è considerato come se fosse un'istruzione singola. Ad esempio un blocco è

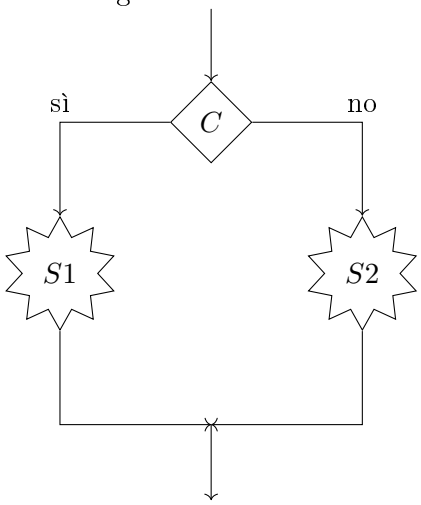
```
{
    a = 3;
    b = 6;
    cout << "messaggio\n";
}
```

Si noti che non si mettono i punti e virgola dopo le parentesi graffe (anche se il loro uso non costituisce errore).

L'esecuzione dell'istruzione if consiste nei seguenti passi

1. viene valutata la condizione C
2. se il risultato è true, è eseguita l'istruzione o il blocco S1
3. se il risultato è false, è eseguita l'istruzione o il blocco S2.

E' possibile capire il funzionamento dell'istruzione if-else mediante il seguente diagramma di flusso



In pratica l'istruzione if-else è una sorta di bivio, da cui partono due percorsi di esecuzione, chiamati anche "rami", che poi si ricongiungono in seguito. La condizione C permette al programma di scegliere quale dei due percorsi seguire.

7.1.1 Esempi di istruzione if-else

Un primo esempio che forniamo è quello di calcolare il massimo m di due numeri reali a, b. Per risolvere questo problema si può usare il seguente schema

$$m = \begin{cases} a & \text{se } a \geq b \\ b & \text{se } a < b \end{cases}$$

che corrisponde al programma

```
int main() {
    int a,b;
    cout << "inserisci due numeri ";
    cin >> a >> b;
    int massimo;
    if(a > b)
```

```
        massimo = a;
    else
        massimo = b;
    cout << "il massimo è " << massimo << endl;
}
```

Un altro esempio è calcolare il valore assoluto di un numero reale x (senza usare la funzione *abs*) tramite il seguente schema

$$|x| = \begin{cases} x & \text{se } x \geq 0 \\ -x & \text{se } x < 0 \end{cases}$$

che corrisponde alla seguente parte di programma

```
if(x>=0)
    valore_assoluto = x;
else
    valore_assoluto = -x;
```

Nell'istruzione if-else S1 e S2 possono essere blocchi, ovvero essere composti da più istruzioni. Ad esempio il seguente codice calcola contemporaneamente il massimo e il minimo di due numeri reali a e b

```
if (a>b) {
    massimo = a;
    minimo = b;
}
else {
    massimo = b;
    minimo = a;
}
```

Se si omettono le parentesi graffe nel primo "ramo" dell'if si ottiene un errore di sintassi

```
if (a>b)
    massimo = a;
    minimo = b;
else {
    massimo = b;
    minimo = a;
}
```

in quanto l'istruzione `minimo=b` è considerata fuori dall'istruzione if: in questo caso l'istruzione if è ritenuta conclusa e la successiva parte else non è attribuita a nessun if.

Invece l'omissione delle graffe nel secondo ramo di `if`, cioè quello relativo ad else, produce invece un programma sintatticamente corretto ma errato

```
if (a>b) {
    massimo = a;
    minimo = b;
}
else
    massimo = b;
    minimo = a;
```

in quanto l'istruzione `minimo = a` è considerata esterna all'`if` ed è eseguita anche nel caso in cui $a > b$. Per essere maggiormente sicuri ed evitare errori, è consigliato usare in ogni caso le parentesi graffe (nei due rami dell'if, ma anche in altre istruzioni composte), anche quando i blocchi sono composti da singole istruzioni.

7.2 If senza else

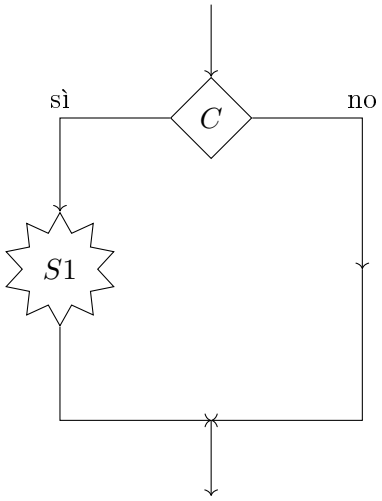
L'istruzione `if` ha anche una versione senza la parte else. Tale istruzione assume in C++ la seguente sintassi

`if`(C) S1

ove C è una condizione e S1 è un'istruzione o un blocco.
L'esecuzione di tale istruzione corrisponde ai seguenti passi

1. è valutata la condizione C
2. se il risultato è true, è eseguita l'istruzione o il blocco S1
3. se il risultato è false, non si fa nulla.

Questa forma si distingue sintatticamente dalla forma completa (con else) perché alla fine della prima sequenza non compare else, ma un'altra istruzione. Il diagramma di flusso equivalente è



Ad esempio un modo alternativo per calcolare il massimo tra a e b è

```
massimo = b;
if (a > b) {
    massimo = a;
}
```

Questa soluzione si può facilmente generalizzare al calcolo del massimo di tre variabili reali a, b, c (o anche un numero maggiore):

```
massimo = a;
if (b > massimo) {
    massimo = b;
}
if (c > massimo) {
    massimo = c;
}
```

In tal caso, infatti, la variabile massimo agisce da massimo parziale: prima solo su a, poi contiene il massimo tra a e b, infine il massimo di tutti e tre i numeri.

Un altro esempio interessante è il seguente: dati tre numeri reali a, b, c, contare quanti sono gli elementi uguali tra di loro.

```
conta = 0;
if (a == b) {
    conta = conta + 1;
}
```

```
if (a == c) {
    conta = conta + 1;
}
if (b == c) {
    conta = conta + 1;
}
```

Ad esempio se $a = 4, b = 4$ e $c = 4$, *conta* vale 3, mentre se $a = 4, b = 2, c = 1$, conta vale 0. In questo esempio la variabile *conta* agisce da contatore: è inizializzata a zero e per ogni coppia di numeri è incrementata se i due numeri sono uguali, mediante l'istruzione `conta = conta + 1`.

Si noti che un'istruzione *if-else* si può sempre ricondurre a due istruzioni `if` relative a condizioni complementari. Ad esempio il calcolo del massimo si potrebbe anche scrivere

```
if (a > b) {
    massimo = a;
}
if (a <= b) {
    massimo = b;
}
```

Tale soluzione è però da ritenersi inferiore rispetto alle altre, dato che richiede di valutare le due condizioni $a > b$ e $a \leq b$, pur sapendo che la seconda è l'opposto della prima.

7.3 If in cascata

Chiaramente si può sempre inserire un'istruzione `if` dentro un'altro `if`. Ad esempio

```
if(a > 5) {
    if(b != 3) {
        c = c + 1;
    }
    d = d - 2;
}
```

in cui quando a è maggiore di 5, d viene decrementato di 2, e se anche b è diverso da 3, c è incrementato di 1.

Una situazione frequente nella programmazione è quando si deve prendere una decisione tra più di due alternative. Mentre una decisione binaria (cioè tra due alternative) è facilmente riconducibile ad un *if-else*, una decisione multivoca si può ricondurre ad una serie di *if-else* in cascata, cioè in cui ogni `if` (tranne il primo) è dentro la parte `else` di quello precedente. La struttura è quindi del tipo

```
if(C1)
    S1
else
    if(C2)
        S2
    else
        if(C3)
            S3
        else
            S4
```

in cui $C1, \ldots, Cn$ sono condizioni e $S1, \ldots, S_{n+1}$ sono istruzioni singole o blocchi.

Ad esempio, per classificare un triangolo in equilatero, isoscele o scaleno a partire dalle lunghezze dei tre lati a, b, c si può usare la seguente parte di programma

```
string tipo;
if(a == b && b == c) {
    tipo = "equilatero";
}
else {
    if(a != b && a != c && b != c) {
        tipo = "scaleno";
}
    else {
        tipo = "isoscele";
}
}
```

Di solito nell'*if in cascata*, gli `if` all'interno degli `else` si scrivono incolonnati:

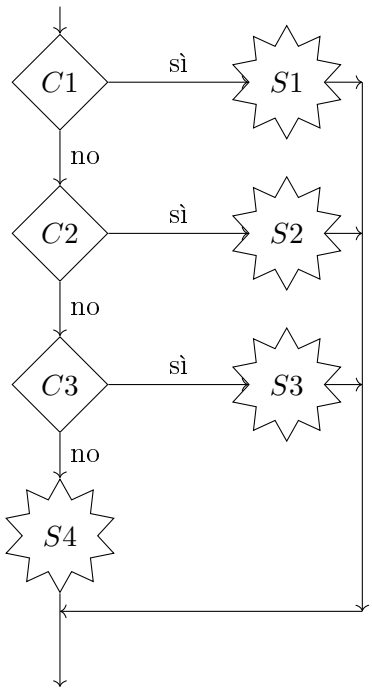
```
string tipo;
```

```
if(a == b && b == c) {
    tipo = "equilatero";
}
else if(a!=b && a!=c && b!=c) {
    tipo = "scaleno";
}
else {
    tipo = "isoscele";
}
```

La semantica dell'*if in cascata* è la seguente

1. viene valutata la conduzione C1, se è vera, è eseguita l'istruzione o il blocco S1
2. in caso contrario, viene valutata la conduzione C2, se è vera, è eseguita l'istruzione o il blocco S2
3. in caso contrario, viene valutata la conduzione C3, se è vera, è eseguita l'istruzione o il blocco S3
4. ecc.
5. se nessuna delle condizioni è vera, viene eseguita la sequenza S_{n+1}.

Questo andamento può anche essere spiegato attraverso il seguente diagramma di flusso



Un altro esempio semplice è fornito dal seguente problema: si vuole scrivere una parte di programma che, dato il numero intero m compreso tra 1 e 12 e corrispondente ad un mese dell'anno, deve calcolare il numero di giorni di tale mese (nell'ipotesi che l'anno non sia bisestile) memorizzandolo nella variabile g. Per risolvere tale problema si può tradurre in C++ un famoso proverbio ottenendo

```
int g;
if(m == 4 || m == 6 || m == 9 || m == 11) {
    g = 30;
}
else if(m == 2) {
    g = 28;
}
else {
    g = 31;
}
```

7.4 Istruzione switch

L'istruzione `switch` è un'istruzione condizionale "a più vie". La sua sintassi è un po' particolare e un suo uso corretto è il seguente

```
switch(e) {
case c1:
    S1
    break;
case c2:
    S2
    break;
...
case cN:
    SN
    break;
default:
    S0
}
```

L'espressione e associata allo `switch` deve essere di tipo intero, compreso `char`. Non sono ammessi valori di altri tipi (reali, stringhe, ecc.). $c1, c2, \ldots, cN$ sono costanti tutte diverse tra di loro e dello stesso tipo dell'espressione e (non sono ammesse variabili o espressioni). $S1, S2, \ldots, SN$ e $S0$ sono sequenze di una o più istruzioni concluse dall'istruzione `break` (ad eccezione di $S0$ in cui break non serve).

Si possono accorpare più case insieme, cioè

```
case c1: case c1_1: case c1_2:
    istruzioni_1
    break;
```

La parte `default` è facoltativa e, se presente, deve essere l'ultima. Il funzionamento di switch è il seguente

1. L'espressione e è valutata, sia v il suo valore
2. Se esiste una clausola case in cui è presente il valore v come costante, sono eseguite tutte le istruzioni in essa fino al break
3. In caso contrario, se esiste la clausola default, sono eseguite le istruzioni associate Sd

4. Se nessuna delle precedenti situazioni si verifica, non accade nulla.

Un esempio di `switch` può essere dato dal seguente codice che scrive sullo schermo "uno", "due" o "tre" se n è compreso tra 1 e 3, altrimenti un messaggio di errore:

```
switch(n) {
case 1:
    cout << "uno";
    break;
case 2:
    cout << "due";
    break;
case 3:
    cout << "tre";
    break;
default:
    cout << "errore: numero non compreso tra 1 e 3\n";
}
```

Se si toglie per errore uno dei `break`, sono eseguite tutte le istruzioni delle clausole seguenti, fino ad incontrare `break` o la fine dello `switch`. Ad esempio in

```
switch(n) {
case 1:
    cout << "uno";
    // break eliminato
case 2:
    cout << "due";
    break;
case 3:
    cout << "tre";
    break;
default:
    cout << "numero non compreso tra 1 e 3\n";
}
```

nel caso in cui $n = 1$, sarà scritto sullo schermo "unodue".

Un altro esempio di `switch` è dato dal codice che stabilisce quanti giorni ci sono in un mese:

```
switch(mese) {
case 4: case 6: case 9: case 11:
// aprile, giugno, settembre o novembre
    g = 30;
    break;
case 2: // febbraio
    g = 28;
    break;
default: // gli altri mesi
    g = 31;
}
```

Si noti comunque che un'istruzione `switch` può essere sempre tradotta in un *if in cascata*, come può essere facilmente intuito.

7.5 Esercizi

1. Scrivere un programma che legge da tastiera due numeri interi e indica se sono uguali, se è più grande il primo o più grande il secondo.

2. Scrivere un programma che calcola il massimo di 3 numeri reali a, b, c.

3. Scrivere un programma che calcola il massimo di 4 numeri reali a, b, c, d.

4. Scrivere un programma che calcola la mediana tra tre numeri reali letti da tastiera, nell'ipotesi che siano tutti diversi (la mediana è l'elemento intermedio tra il minimo e il massimo).

5. Scrivere un programma che legge da tastiera tre numeri reali e li scrive in ordine crescente.

6. Scrivere un programma che dati i coefficienti a, b, c di un'equazione di secondo grado $ax^2 + bx + c = 0$ trova le radici reali, o nel caso in cui $\Delta < 0$, scrive che non ci sono soluzioni reali.

7. Scrivere un programma che dati i coefficienti a, b, c, d e i termini noti e, f di un sistema di due equazioni in due incognite

$$\begin{cases} ax + by &= e \\ cx + dy &= f \end{cases}$$

   trova le soluzioni o, quando il sistema non è compatibile, scrive che non c'è un'unica soluzione

8. Scrivere un programma che dati tre coppie di punti $P1 = (x1, y1)$, $P2 = (x2, y2)$ e $P3 = (x3, y3)$ decide se P1 è all'interno del rettangolo i cui vertici opposti sono P2 e P3.

9. Scrivere un programma che dato un numero naturale n compreso tra 1 e 999999 restituisce come risultato il numero delle cifre decimali di n (senza usare il logaritmo).

10. Scrivere un programma che legge da tastiera tre numeri reali a, b, c, verifica che possano essere i lati di un triangolo (deve essere soddisfatta la diseguaglianza triangolare $a \leq b + c$ per tutti e tre i lati) ed infine scrive sullo schermo il tipo di triangolo: equilatero (tre lati uguali), isoscele (due lati uguali) o scaleno (tre lati diversi).

11. Scrivere un programma che legge da tastiera un primo numero reale x, un carattere *op* (che può essere solo '+', '-', '*', '/') ed un secondo numero reale y e calcola il risultato dell'espressione corrispondente. Ad esempio leggendo x=3, op=+ e y=4 deve scrivere 7 (3+4), mentre con x=3, op=* e y=7 il risultato è 21.

12. Scrivere un programma che legge da tastiera un numero naturale n compreso tra 1 e 99 e lo scrive sullo schermo a parole. Ad esempio se $n = 75$ deve scrivere la stringa *settantacinque*. (Non usare uno switch con 99 case...)

13. Scrivere un programma che legge da tastiera una data, suddivisa in giorno, numero del mese e anno, e verifica se è una data esistente.

14. Scrivere un programma che legge da tastiera un numero n compreso tra 1 e 99 e lo scrive sullo schermo in numerazione romana. Ad esempio se $n = 78$ deve scrivere la stringa LXXVIII.

15. Scrivere un programma che applica la legge di Ohm in uno dei tre modi possibili: a partire da due dei tre possibili valori tra R, I, V calcola il terzo. Il programma può ad esempio chiedere quali delle tre problemi si vuole risolvere e in base a tale scelta legge i due dati e scrive il risultato corrispondente.

Chapter 8

Strutture iterative limitate

8.1 Iterazione limitata e illimitata

I **costrutti iterativi** consentono di eseguire delle istruzioni per più volte ed hanno un'importanza fondamentale nella programmazione: in pratica ogni programma significativo ha almeno un costrutto iterativo. I costrutti iterativi sono comunemente chiamati **ciclo**.

Esistono due motivi principali per usare i cicli, oltre alla semplice ripetizione *tout court* di istruzioni:

- algoritmi di tipo iterativo: tali algoritmi hanno bisogno di più iterazioni per convergere alla soluzione cercata, ad esempio l'algoritmo di Euclide per calcolare il M.C.D. di due numeri naturali;
- operazioni su collezioni di elementi: si vuole svolgere la stessa operazione su tutti gli elementi di un vettore, un insieme o altra collezione di dati, ad esempio calcolare la somma di due vettori o la media aritmetica di un insieme di dati numerici.

Un ciclo valido deve ripetere le istruzioni per un numero finito di volte (**numero di iterazioni**). Esistono a tal scopo due situazioni:

1. Il numero di iterazioni è noto all'inizio del ciclo. Si tratta dell'**iterazione limitata**, per definire correttamente un ciclo di questo tipo è necessario rispondere alla domanda "quante volte ripetere ?".
2. Il numero di iterazioni non è noto all'inizio del ciclo. In tal caso si tratta dell'**iterazione non limitata**. Il ciclo va avanti, controllando ad ogni iterazione se è il caso di continuare. Pertanto la domanda da porsi è "fino a quando ripetere ?".

In questo capitolo vedremo il principale costrutto di iterazione limitata, il ciclo `for`, invece studieremo nel prossimo capitolo i costrutti di iterazione non limitata.

8.2 Il ciclo for

Il ciclo `for` nella sua forma più utilizzata in C++ ha la seguente sintassi

`for(` I=A `;` I<=B `;` I++`)` S

in cui I è una variabile chiamata **indice** del ciclo, solitamente di tipo intero, A e B sono gli estremi di un intervallo di numeri, solitamente interi, e S è un'istruzione singola o un blocco.

Dopo la parola chiave `for`, tra parentesi troviamo le tre componenti del ciclo.

- La prima parte `I=A` è l'**inizializzazione** dell'indice e indica "da che punto" parte il ciclo.
- La seconda parte `I<=B` è la **condizione di continuazione** del ciclo for, se è vera il ciclo continua, se è falsa il ciclo termina.
- La terza parte `I++` è l'**aggiornamento dell'indice**, denota come ottenere il successivo valore dell'indice.

La speciale notazione `I++` è un'abbreviazione classica del linguaggio C che equivale ad `I = I + 1` Si può usare con qualsiasi variabile numerica, anche al di fuori del ciclo for. L'operatore `++` si può usare anche in forma prefissa, cioè `++i`. La differenza tra `I++` e `i++` è sottile e per gli usi illustrati in queste dispense queste due forme possono essere ritenute equivalenti.

Esiste anche l'altra abbreviazione `I--`, che corrisponde a `I = I - 1`. Anche tale operatore ha la sua variante prefissa.

E' molto frequente, sia in matematica, che nella programmazione, usare come indici i, j, k ed infatti in queste dispense useremo quasi sempre tali nomi per le variabili usate come indici.

L'esecuzione del ciclo for consiste nell'eseguire S per ognuno dei numeri interi nell'intervallo $[A, B]$. L'indice I viene fatto variare in $[A, B]$, ovvero assume ad ogni iterazione un valore crescente di tale intervallo: $A, A+1, A+2, \ldots, B-1, B$.

S viene pertanto eseguita $B - A + 1$ volte nel caso in cui $A \leq B$. Se invece $A > B$, S non viene eseguita nemmeno una volta, dato che I è subito maggiore di B.

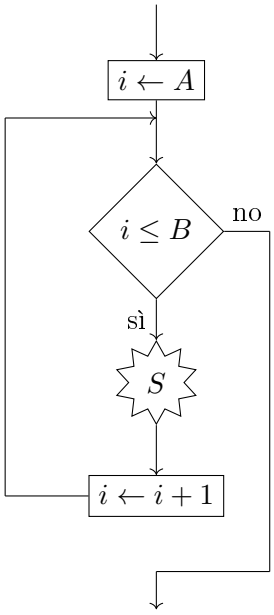
La semantica intuitiva del ciclo for è

1. valuta A e B, siano a e b i valori ottenuti
2. inizializza l'indice I al valore a
3. se I è maggiore di b, termina il ciclo
4. (altrimenti) esegui S
5. incrementa I di 1
6. torna al punto 3.

L'esecuzione di un ciclo corrisponde perciò ad eseguire le seguenti istruzioni in sequenza

```
I = A;
S
I = A + 1;
S
...
I = B - 1;
S
I = B;
S
```

con gli evidenti vantaggi di compattezza (S è scritto una sola volta) e generalità (A e B possono essere espressioni non costanti). Un diagramma di flusso che esemplifica il comportamento del ciclo for è il seguente:



Ad esempio, per calcolare la somma dei numeri naturali da 1 a n (con $n > 1$)

```
int somma=0,i;
for(i = 1; i <= n; i++) {
        somma = somma + i;
}
```

Con $n = 5$ le variabili *somma* e i assumerebbero progressivamente i seguenti valori

somma	i
0	indef.
0	1
1	1
1	2
3	2
3	3
6	3
6	4
10	4
10	5
15	5
15	6

E' possibile dichiarare l'indice del ciclo for direttamente nella prima parte (inizializzazione) Ad esempio

```
for(int i = 1; i <= 10; i++) {
        cout << i << endl;
}
```

Però in questo modo l'indice i si può usare solo all'interno del ciclo: al di fuori del ciclo i non esiste più.

```
for(int i = 1; i <= 10; i++) {
        cout << i << endl;
}
// ora la variabile i non esiste più
```

8.3 Esempi

L'esempio della somma dei numeri interi da 1 a n si può generalizzare per calcolare una sommatoria generica

$$\sum_{i=1}^{n} f(i)$$

in cui f è una funzione di i, anche espressa sotto forma di espressione. Ad esempio per calcolare la somma dei quadrati dei numeri interi da 1 a n si può scrivere

```
int somma = 0;
for(int i = 1; i <= n; i++) {
        somma = somma + i * i;
}
```

In maniera molto simile è possibile calcolare il fattoriale di un numero naturale n come prodotto di tutti i numeri naturali compresi tra 1 e n

```
int prodotto = 1;
for(int i = 1; i <=n ; i++) {
        prodotto = prodotto * i;
}
```

Tale funzione è corretta anche nel caso $n = 0$. Infatti un ciclo da 1 a 0 non ha effetto e il risultato è 1, che corrisponde a $0!$ per definizione.

Anche in questo caso una generalizzazione possibile è quella del calcolo della produttoria generica i cui termini sono una funzione di i.

Supponiamo come ulteriore esempio di voler calcolare il coefficiente binomiale di due numeri naturali n e k, con $0 \leq k \leq n$. Il coefficiente binomiale si può calcolare mediante la seguente formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

che porta alla seguente implementazione

```
int f1 = 1;
for(int i = 1; i <= n; i++) {
        f1 = f1 * i;
}
int f2 = 1;
for(int i = 1; i <= k; i++) {
        f2 = f2 * i;
}
int f3 = 1;
for(int i = 1; i <= n - k; i++) {
        f3 = f3 * i;
}
int cb = f1 / (f2 * f3);
```

Si noti che i tre cicli for usano tutti lo stesso indice i e ciò è perfettamente legale.

Un metodo di calcolo più efficiente si ottiene usando la formula alternativa

$$\binom{n}{k} = \frac{(n-k+1)\cdots n}{k!}$$

```
int f1 = 1;
for(int i= n - k + 1; i <= n; i++) {
        f1 = f1 * i;
}
int f2 = 1;
for(int i = 1; i <= k; i++) {
        f2 = f2 * i;
}
int cb = f1 / f2;
```

Poiché i due cicli for hanno lo stesso numero di iterazioni, si possono accorpare in un solo ciclo for. L'unica difficoltà è che bisogna aggiustare la formula in modo da usare lo stesso indice. Una possibilità è quindi

```
int f1 = 1, f2 = 1;
for(int i = 1; i <= k; i++) {
        f1 = f1 *(n + 1 - i);
        f2 = f2 * i;
}
int cb = f1 / f2;
```

Come nuovo esempio, facciamo vedere come calcolare in maniera approssimata il massimo di una funzione f definita su un intervallo $[a, b]$. Un'idea molto semplice è quella di calcolare f su un insieme finito di punti di $[a, b]$ e trovare il valore maggiore di f su tali punti. Per esempio si potrebbero prendere i punti

$$x_0 = a, x_1 = a + h, x_2 = a + 2h, \ldots, x_{n-1} = b - h, x_n = b$$

ove $h = \frac{b-a}{n}$

Supponiamo che la funzione sia $f(x) = \sin x \cdot \cos x$ e che i valori per a, b, n siano già stati impostati. Quindi avremo il seguente codice

```
double h = (b - a) / n, x = a;
double m = sin(x) * cos(x);
for(int i = 1; i <= n; i++) {
        x = x + h;
        double y = sin(x) * cos(x);
        if (y > m) {
              m = y;
        }
}
```

Il funzionamento dell'algoritmo è il seguente. La variabile m all'inizio è pari a $f(a)$, mentre x è uguale ad a.

Ad ogni passo x è spostato al punto successivo, tramite un incremento di h, e la funzione f è valutata sul punto x, ottenendo y.

Se y è maggiore di m, m è aggiornato con il valore di y. E' facile vedere che alla fine del ciclo

$$m = \max\{f(x_j) : \text{ per } j = 0, 1, \ldots, n\}$$

Con semplici modifiche è possibile ottenere, anziché il valore massimo di f, il (o uno dei) punto x_j in cui tale valore è ottenuto.

8.4 Cicli annidati

Supponiamo di voler calcolare un valore approssimato del numero di Nepero e mediante la somma parziale n-esima della serie

$$\sum_{k=0}^{n} \frac{1}{k!}$$

arrestandosi cioè all'elemento di ordine n. Un modo diretto per calcolare di tale sommatoria è di generare ad uno ad uno i fattoriali dei numeri da 0 a n e di sommare i loro inversi. Ciò può essere svolto dal seguente programma

```
double somma = 0;
for(int k = 0; k <= n; k++) {
        int fattoriale = 1;
        for(int i = 1; i <= k; i++) {
                fattoriale = fattoriale * i;
        }
        double termine = 1.0/ fattoriale;
        somma = somma + termine;
}
cout << "valore approssimato di e=" << somma << endl;
```

In questo esempio si nota una costruzione particolare con due cicli `for`, il primo dei quali contiene il secondo. Questa situazione è indicata mediante il termine di **cicli for annidati** ed è molto utilizzata nell'informatica.

Il ciclo `for` con indice k è chiamato **ciclo esterno**, mentre il ciclo `for` con indice i è il **ciclo interno**. Ovviamente l'annidamento può essere anche esteso a più di due cicli.

```
for(int j = 1; j <= 1000; j++)
        for(int k = 1; k <= 100; k++)
                for(int h = 1; h <= 200; h++)
...
```

In tal caso si introduce il concetto di profondità di annidamento: il ciclo esterno (indice j) ha profondità 0, quello mediano (indice k) ha profondità 1 e quello più interno (indice h) ha profondità 2.

Il corretto funzionamento dei cicli annidati richiede che i cicli for coinvolti usino indici diversi.

E' però ammesso che l'intervallo associato ad un ciclo dipenda dal valore dell'indice di un ciclo esterno ad esso. Ad esempio nella nell'esempio che

calcola un'approssimazione di e l'intervallo del ciclo su i dipende dal valore di k.

Per capire meglio cosa succede scriviamo i valori assunti da i e k nel caso $n = 4$:

k	i
0	1
1	1
2	1
2	1
3	1
3	2
3	3
4	1
4	2
4	3
4	4

Si noti che per $k = 0$, il ciclo for interno non ha effetto.

8.5 Varianti del ciclo for di base

La forma del ciclo `for` che abbiamo usato negli esempi precedenti è quella maggiormente usata nei programmi, ma non è assolutamente l'unica. In realtà il ciclo `for` è abbastanza libero e se ne possono creare varianti ad-hoc per particolari situazioni. Alcune varianti del ciclo for di utilizzo frequente sono le seguenti.

8.5.1 For con l'estremo superiore escluso

Nel ciclo `for` usato finora, l'ultimo valore assunto dall'indice all'interno del ciclo è proprio B. Si può però terminare il ciclo un passo prima, in modo che l'indice non arrivi ad assumere il valore B. E' sufficiente nella condizione di continuazione usare il $<$ stretto, anziché il `<=`. Ad esempio

```
for(int i = 0; i < 10; i++) {
        cout << i << endl;
}
```

scrive tutti i numeri naturali tra 0 e 9, senza arrivare al 10.

8.5.2 For all'indietro

Per creare un ciclo `for` che "va all'indietro" si usa la sintassi

for(I = A ; I >= B ; I") S

in cui $A \geq B$. L'indice varia nell'intervallo $[B, A]$, però partendo dal valore maggiore e andando verso quello minore. Ad esempio

```
for(int i = 10; i >= 1; i--) {
        cout << i << endl;
}
```

scrive sullo schermo i numeri da 1 a 10 in ordine decrescente.

8.5.3 For a passo non unitario

Si può ottenere facilmente un ciclo `for` in cui l'indice viene incrementato di una quantità diversa da +1 e -1. Nel ciclo

`for(` I = A ; I <=B ; I = I + D) S

l'indice ad ogni passo viene incrementato di D (che è un valore positivo) e l'iterazione termina quando I ha raggiunto o superato B. Ad esempio

```
int somma = 0;
for(int i = 1; i <= n; i = i + 2) {
        somma = somma + i;
}
```

calcola la somma dei numeri dispari compresi tra 1 e n, infatti i assume i valori 1,3,5,. . . ,. L'ultimo valore assunto da i è n, se n è dispari, se invece n è pari, il ciclo termina con $i = n - 1$.

Se invece $D < 0$, allora il ciclo va scritto come un ciclo all'indietro

```
for(i = 10; i>= 0; i = i - 2) {
        cout << i << endl;
}
```

scrive i numeri pari da 2 a 10 in ordine decrescente.

Le due istruzioni `i = i + 2` e `i = i - 2` si possono abbreviare in `i += 2` e `i -= 2`, rispettivamente.

8.5.4 For con indice reale

E' possibile definire un ciclo `for` che usa un indice di tipo `float` o `double`, anziché `int`. Dato che nei numeri reali, l'incremento $+1$ non ha lo stesso significato che ha nei numeri interi, di solito tali cicli for si usano con incremento non unitario. Ad esempio

```
for(double alpha = 0; alpha <= M_PI / 2; alpha = alpha + M_PI / 36) {
        cout << alpha << " " << sin(alpha) << endl;
}
```

stampa i valori del seno degli angoli compresi tra 0 e $\pi/2$, a passi di $\pi/36$ (che corrisponde a 5 gradi).

I cicli con indici reale non sono usati spesso, anche perché tale ciclo si può sempre riscrivere in un ciclo ad indice intero

```
for(int i = 0; i <= 18 ; i++) {
        double alpha = i * M_PI / 36;
        cout << alpha << " " << sin(alpha) << endl;
}
```

con il vantaggio che il ciclo è svolto usando l'aritmetica intera (le cui operazioni sono più veloci).

8.6 Esercizi

1. Scrivere un programma che dati due numeri naturali n e k, calcola il numero di disposizioni semplici di n elementi a k a k, che è pari a $n(n-1)(n-2)\dots(n-k+1)$.

2. Scrivere un programma che, dati un numero reale x e un numero naturale n, calcola x^n tramite moltiplicazioni successive.

3. Estendere la soluzione del precedente esercizio al caso in cui n è un numero relativo.

4. Scrivere un programma che, dati tre numeri naturali m, n, k, calcola $m^n \mod k$, tramite n moltiplicazioni successive modulo k.

5. Scrivere un programma che, dato un numero naturale n calcola la somma dei divisori propri di n.

6. Scrivere un programma che, dato un numero naturale n, stabilisce se n è primo, contando il numero dei divisori propri di n.

7. Scrivere un programma che controlla se un numero naturale n è perfetto, ovvero è pari alla somma dei suoi divisori (incluso 1, ma escluso n stesso).

8. Scrivere un programma che, dato un numero naturale n, calcola l'n-esimo numero di Fibonacci. I primi due numeri di Fibonacci sono 1, ogni altro si ottiene sommando i due numeri di Fibonacci precedenti. Perciò i primi numeri di Fibonacci sono $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$.

9. Scrivere un programma che, dati un numero reale x e un numero naturale n, calcola $\log(x + 1)$ mediante la formula di Taylor arrestata all'n-esimo termine.

10. Scrivere un programma che, dati un numero reale x e un numero naturale n, calcola e^x mediante la formula di Taylor arrestata all'n-esimo termine.

11. Scrivere un programma che, dati due numeri reali a, b e numero naturale n, calcola l'integrale approssimato di una funzione f, di una sola variabile x, in $[a, b]$ tramite il metodo dei trapezi. Si suddivida l'intervallo $[a, b]$ in n sottointervalli di uguale ampiezza $h = (b-a)/n$, si calcoli per ognuno di questi sottointervalli $[x_i, x_{i+1}]$ l'area del trapezio avente come basi $f(x_i)$ e $f(x_{i+1})$, ed infine si sommino le aree così ottenute.

Chapter 9

Strutture iterative non limitate

Come abbiamo visto nel capitolo precedente, esistono due tipi di iterazione: l'iterazione limitata e quella non limitata. In questo capitolo sarà descritta il secondo tipo di iterazione, verrà inoltre svolto un confronto tra il primo e il secondo tipo di iterazione.

9.1 Ciclo while

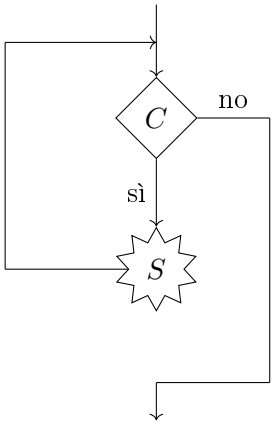
La forma più utilizzata di iterazione non limitata è il ciclo `while`, che è presente in molti altri linguaggi di programmazione. Esso ha la sintassi

`while(C) S`

in cui C è una condizione e S un'istruzione o un blocco. La semantica "intuitiva" del ciclo while è la seguente

1. valuta la condizione C
2. se è falsa, il ciclo termina
3. (altrimenti) esegui S
4. torna al punto 1

Il comportamento del ciclo `while` può anche essere illustrato dal seguente diagramma di flusso.



Il ciclo `while` può essere usato come mezzo per raggiungere un obbiettivo all'interno di un programma:

- la condizione è l'opposto dell'obbiettivo da raggiungere;
- le istruzioni eseguite ripetutamente fanno raggiungere (prima o poi) l'obbiettivo;
- appena la condizione diventa falsa, l'obbiettivo è raggiunto ed il ciclo termina.

E' comunque importante assicurarsi che le istruzioni interne al ciclo `while` contribuiscano alla terminazione del ciclo garantendo che prima o poi la condizione diventi falsa. Senza questa garanzia il ciclo non termina mai e quindi è pressoché inutile.

Uno dei casi in cui `while` è indispensabile si ha quando il valore della condizione è influenzato direttamente dall'utente: ad esempio si vuole sommare una serie di numeri interi letti da tastiera fino a che l'utente non inserisce 0. Senza memorizzare tutti i valori da sommare (cosa praticamente impossibile senza usare array, liste o simili) si può risolvere il problema tramite una variabile accumulatore che deve essere inizializzata a 0 e su cui si addizionano uno alla volta i dati letti da tastiera.

```
somma=0;
cout << "Inserisci i numeri da sommare (0 per terminare) ";
cin >> dato;
while(dato != 0) {
    somma = somma + dato;
    cin >> dato;
}
cout << "La somma e' " << somma << endl;
```

Un altro caso adatto all'uso del ciclo `while` è quello in cui si ricerca un elemento con determinate caratteristiche. Ad esempio, cerchiamo la parte intera r della radice quadrata di un numero intero n. Il numero r gode della proprietà che

$$r^2 \leq n < (r+1)^2$$

Per trovare r si usa un ciclo che va avanti fino a che non si arriva al più piccolo numero intero r tale che $r^2 > n$. Allora la parte intera della radice quadrata è proprio $r - 1$. Perciò

```
int r=0;
while(r * r < n) {
    r = r + 1;
}
r = r - 1;
```

Un altro esempio è quello per calcolare il massimo comun divisore di due numeri interi a e b mediante il metodo di Euclide, già descritto in un precedente capitolo. Per risolvere tale problema si può usare un ciclo `while` nel seguente modo

```
while(a != b) {
    if(a > b) {
        a = a - b;
    }
    else {
        b = b - a;
    }
}
cout << "MCD " << a << endl;
```

Una forma più veloce per risolvere tale problema è quello di considerare che se a è molto più grande di b, si può calcolare il resto della divisione di a per b, che corrisponde a più sottrazioni consecutive. Con alcune semplici considerazioni si perviene al seguente procedimento

```
while(b != 0) {
    int r = a % b;
    a = b;
    b = r;
}
cout << "MCD " << a << endl;
```

Come ulteriore esempio di ciclo `while`, vediamo un metodo (chiamato *metodo della bisezione*) per trovare uno zero di una funzione continua $f : [a, b] \to \mathbb{R}$ nell'ipotesi che f assuma agli estremi di $[a, b]$ valori di segno opposto. Per un noto teorema di analisi esiste almeno un elemento $z \in [a, b]$, detto zero di f, tale che $f(z) = 0$. Se m è il punto medio di $[a, b]$ e $f(a) < 0$ (e quindi $f(b) > 0$), allora i casi sono tre

1. $f(m) = 0$, allora m è uno zero di f;
2. $f(m) > 0$, allora f ha almeno uno zero in $[a, m]$;
3. $f(m) < 0$, allora f ha almeno uno zero in $[m, b]$.

In maniera analoga è il caso in cui $f(a) > 0$. In sintesi, basta controllare se $f(a)$ e $f(m)$ hanno lo stesso segno oppure no: un modo semplice è verificare se $f(a) \cdot f(m) > 0$.

Si può quindi impostare un ciclo `while` che continua fino a che non si trova uno zero oppure l'intervallo in cui f ammette uno zero non diventa sufficientemente piccolo, di modo che il suo punto medio sia un'approssimazione accettabile di uno zero di f.

L'eventualità che l'algoritmo trovi proprio uno zero di f è molto remota anche perché a causa degli errori di arrotondamento è improbabile che, pur essendo m il vero zero di f, $f(m)$ sia esattamente 0. Comunque la condizione di continuazione del ciclo while tiene conto di entrambe queste due possibilità.

Ad esempio se la funzione di cui si vuole trovare lo zero è $f(x) = cosx - x$, che assume valori opposti nell'intervallo $[0, \pi/2]$, con precisione eps=10^{-5}

```
double a = 0, b = M_PI/2, eps = 1e-5;
bool trovato = false;
double ya = cos(a)-a, yb = cos(b)-b;
while(b - a >= eps && trovato == false) {
    m = (a + b) / 2;
    double ym = cos(m) - m;
    if(ym == 0.0) {
        trovato = true;
    }
    else if(ym * ya<0) {
        b = m;
        yb = ym; }
    else {
        a = m;
```

```
        ya = ym;
    }
}
double z = (a + b) / 2;
```

Per trovare gli zeri di un'altra funzione, è sufficiente cambiare le espressioni che calcolano i valori di ya, yb e ym e i valori iniziali di a e b.

Si noti che l'obbiettivo del ciclo è disgiuntivo: trovare uno zero oppure avere un intervallo piccolo. Di conseguenza la condizione del while è congiuntiva: lo zero non è stato trovato e l'intervallo ha un ampiezza maggiore o uguale a epsilon. L'uso della variabile booleana *trovato* serve a ricordare che lo zero è stato effettivamente trovato.

9.2 Istruzione do-while

Nel ciclo while la condizione è controllata all'inizio di ogni iterazione: ciò comporta che il ciclo che potrebbe concludersi subito, quando la condizione è falsa già all'inizio, senza svolgere alcuna iterazione. In alcuni casi si ha la necessità che le istruzioni del ciclo siano eseguite comunque almeno una volta. A tal scopo esiste l'istruzione `do-while`, la quale ha la sintassi

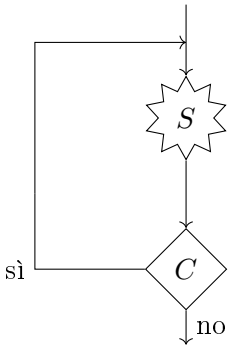
`do` S `while(`C`);`

in cui C è una condizione e S un'istruzione singola o un blocco. Si noti il punto e virgola dopo la parentesi chiusa.

La semantica dell'istruzione `do-while` è leggermente diversa dall'istruzione `while` ed è illustrata nel seguente schema

1. S è eseguita
2. C è valutata
3. se è falsa il ciclo termina
4. altrimenti il ciclo continua con il punto 1.

Il diagramma di flusso equivalente al ciclo *do-while* è



La differenza tra `while` e `do-while` è che nel primo la condizione è controllata prima di ogni iterazione, mentre nel secondo è controllata dopo. Quindi nel `do-while` vi è comunque la garanzia che le istruzioni siano eseguite almeno una volta. Per il resto, le due istruzioni si comportano allo stesso modo:

- il ciclo termina quando la condizione diventa falsa (quindi può essere la negazione dell'obbiettivo da raggiungere)
- il ciclo può non terminare mai
- non esiste un massimo numero di iterazioni.

Come primo esempio di `do-while` risolviamo il problema di leggere da tastiera un numero intero assicurandoci che sia positivo.

```
do {
    cout << "Inserisci un numero positivo ";
    cin >> numero;
} while(numero <= 0);
```

Tale problema può essere risolto anche con `while`, leggendo il numero anche prima del ciclo:

```
cout << "Inserisci un numero positivo ";
cin >> numero;
while(numero <= 0) {
    cout << "Non va bene, deve essere positivo, inseriscilo nuovamente ";
    cin >> numero;
}
```

Un esempio matematico consiste nel calcolo approssimato della radice quadrata di un numero reale $x \geq 0$ tramite il metodo delle tangenti (detto

anche di *Newton-Raphson*). Si parte con $r_0 = x$ e poi ad ogni passo si calcola il successivo valore della seguente successione di numeri reali

$$r_{n+1} = \frac{1}{2}\left(r_n + \frac{x}{r_n}\right)$$

Si può dimostrare che tale successione converge alla radice quadrata di x.

Il limite non può essere calcolato esattamente tramite programma, ma con vari metodi è possibile trovare un'approssimazione sufficientemente precisa con un numero finito di passaggi.

Un modo è quello di calcolare i valori della successione fino a che gli ultimi due valori r_n e r_{n+1} sono sufficientemente vicini, ad esempio quando $|r_n - r_{n+1}| < \epsilon$. Dato che il ciclo while continua quando la condizione è vera, è necessario usare come condizione l'opposto di quella scritta in precedenza, ovvero $|r_n - r_{n+1}| \geq \epsilon$. Non c'è bisogno di conservare tutti i valori della successione $\{r_n\}$, è sufficiente avere solo quello corrente (indicato con r) e quello precedente (indicato con *r_prec*). Il codice di programma è perciò

```
double r = x, r_prec;
do {
    r_prec = r;
    r=0.5 * (r_prec + x / r_prec);
} while(abs(r - r_prec) >= eps);
```

Questa parte di programma si può adattare al calcolo approssimato dello zero di una funzione f (sotto opportune ipotesi di regolarità). Si parte con un valore r_0 che è vicino allo zero di f e poi ad ogni passo si definisce la successione

$$r_{n+1} = r_n - \frac{f(r_n)}{f'(r_n)}$$

L'iterazione, anche in questo caso, può essere terminata quando gli ultimi due valori r_n e r_{n+1} sono sufficientemente vicini: $|r_n - r_{n+1}| < \epsilon$.

9.3 Istruzione break

L'istruzione `break`, già vista come chiusura dei "case" nell'istruzione `switch`, si può usare anche nei cicli for, while e do-while con l'effetto di terminare anticipatamente l'esecuzione del ciclo e passare all'istruzione successiva. Ogni

uso sensato di `break` deve essere condizionato tramite un'istruzione `if`, altrimenti il ciclo terminerebbe subito alla prima iterazione. Di solito si usa così

```
while(C) {
    ...
    if(C2) break;
    ...
}
```

o

```
for(I = A; I <= B; I++) {
    ...
    if(C2) break;
    ...
}
```

Quando si verifica la condizione C2, il ciclo viene interrotto e l'esecuzione prosegue dall'istruzione successiva al ciclo (come se il ciclo fosse stato eseguito per intero).

Si noti che quando si usa `break`, il ciclo può terminare normalmente (nel `while` quando la condizione C diventa falsa, nel `for` quando, ad esempio, $I > B$) oppure in maniera anticipata (quando la condizione C2 associata al break è vera). E' importante notare che nel prosieguo del programma sarà quasi sempre necessario distinguere quale dei due motivi ha causato l'uscita dal ciclo.

Un esempio utile di `break` si trova per verificare se un numero naturale n è primo. Si può facilmente dimostrare che n è primo se e solo se n non ammette divisori nell'intervallo $[2, \lceil\sqrt{n}\rceil]$. Si può usare un ciclo for che si interrompe appena trova un divisore.

```
bool primo = true;
int r = int(sqrt(n));
for(int d = 2; d <= r; d++) {
    if(n%d == 0) {
        primo = false;
        break; }
}
```

La variabile booleana *primo* serve a ricordare qual è il motivo della terminazione del ciclo. Se infatti è rimasta `true`, il ciclo è stato svolto per

intero e il divisore di n non esiste (quindi n è primo). Invece se il ciclo è stato interrotto anticipatamente, n non è primo.

L'uso di `break` è non è mai indispensabile, infatti lo stesso ciclo si potrebbe scrivere usando un ciclo `while`

```
bool primo = true;
int r = int(sqrt(n));
int d = 2;
while(d <= r && primo == true) {
    if(n % d == 0) {
        primo = false;
    }
    else {
        d++;
    }
}
```

Infatti il ciclo termina quando $d > r$ oppure se è stato trovato un divisore.

Con una tecnica simile, tutti gli esercizi in cui si chiede di controllare l'esistenza di un elemento all'interno di un insieme (ad esempio un intervallo) con determinate caratteristiche possono essere risolti mediante un ciclo `for` con un `break` o mediante un ciclo `while` con doppia condizione.

Si noti che, in C++, `break` interrompe solo il ciclo in cui si trova, pertanto se si trova all'interno di un ciclo annidato, come in

```
for(int i = 1; i <= 1000; i++) {
    for(int j = 1; j <= 1000; j++) {
        ...
        if(impossibile_continuare) break;
        ...
    }
}
```

in cui si interrompe solo il ciclo for sull'indice j, ma non quello sull'indice i. In tali situazioni si dovrebbe ripetere il controllo della condizione anche nel ciclo for esterno

```
for(int i=1; i<=1000; i++)
    for(int j=1; j<=1000; j++) {
        ...
        if(impossibile_continuare) break;
```

```
        ...
    }
    if(impossibile_continuare) break;
}
```

9.4 Istruzione continue

L'istruzione `continue`, di utilizzo abbastanza raro, consente di interrompere l'iterazione corrente e di passare a quella successiva. Ad esempio

```
for(int i = 1; i <= 10; i++) {
    // istruzioni A
    if(i == 6) continue;
    // istruzioni B
}
```

le istruzioni "B" non sono eseguite per $i = 6$. Anche l'istruzione `continue` è evitabile

```
for(int i = 1; i <= 10; i++) {
    // istruzioni A
    if(i != 6) {
    // istruzioni B
    }
}
```

9.5 Confronto tra while e for

Nel linguaggio C++ i cicli `for` e `while` sono strettamente collegati.

Infatti è facile vedere che un ciclo `for` del tipo

```
for(I = A; I <= B; I++)
    S
```

può essere sempre emulato con un ciclo `while` nel seguente modo

```
I = A;
while(I <= B) {
    S
    I++;
}
```

Ad esempio il ciclo `for` che calcola la somma dei primi n numeri naturali

```
int somma = 0;
for(int i = 1; i <= n; i++)
    somma = somma + i;
```

può essere riscritto come ciclo `while`

```
int somma = 0;
int i = 1;
while(i <= n) {
    somma = somma + i;
    i++;
}
```

D'altro canto, un ciclo `while` può essere sempre scritto come particolare ciclo `for`. Ad esempio

```
q = 0;
while(a > b) {
    a = a - b;
    q++;
}
```

si può scrivere come

```
q = 0;
for( ; a > b ;) {
    a = a - b;
    q++;
}
```

Questa particolare forma di `for` prevede solo la condizione di continuazione, ma non l'inizializzazione, né l'aggiornamento. Ovviamente un ciclo del genere non è più un costrutto di iterazione limitata, ma diventa a tutti gli effetti un'istruzione di iterazione non limitata. Nella pratica non si usa quasi mai un ciclo `for` scritto in questo modo.

Dato che in C++ ogni `for` si può sempre riscrivere come `while` e ogni `while` come particolare ciclo `for`, si può concludere che i due cicli sono equivalenti dal punto di vista semantico/teorico. Comunque, rimangono distinti dal punto di vista pratico.

Infine, in C++ si può usare una forma estremamente generale di ciclo `for`, che sostanzialmente è un `while` "potenziato". Infatti, la sua sintassi è

`for( e1 ; e2 ; e3 )` S

in cui e2 è una condizione, e1 e e3 sono istruzioni di assegnamento e S è un'istruzione o un blocco.

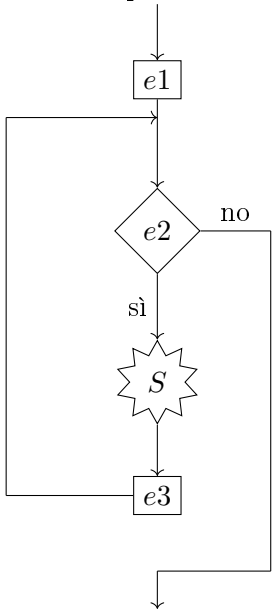
Tale forma di ciclo `for` è equivalente al frammento di codice

```
e1
while(e2) {
    S
    e3
}
```

L'istruzione ha il seguente comportamento

1. viene eseguita e1
2. viene valutata e2
3. se è falsa, il ciclo termina
4. altrimenti, viene eseguita S
5. viene eseguita e3
6. il ciclo ritorna al punto 2

che corrisponde al diagramma di flusso



Un caso molto particolare di tale istruzione è il ciclo `for` infinito, che si indica direttamente con `for(;;)`. Tale ciclo ripete per sempre le istruzioni ad esso associate. Ovviamente un ciclo infinito ha senso solo quando esiste almeno una via di uscita, ad esempio tramite `break`.

Comunque un ciclo simile può risultare utile in alcune situazioni, ad esempio definire un ciclo la cui condizione deve essere controllata in mezzo dell'iterazione (all'inizio sarebbe un `while`, alla fine sarebbe un `do-while`). Ad esempio, si vuole sommare una serie di numeri interi inseriti da tastiera e conclusi con 0.

```
int somma = 0;
for(;;) {
    cout << "Inserisci un numero, scrivi 0 per finire ";
    cin >> num;
    if(num == 0) break;
    somma = somma + num;
}
```

Un ciclo infinito si può anche ottenere con while con la condizione `true`

```
int somma = 0;
while(true) {
    cout << "Inserisci un numero, scrivi 0 per finire ";
    cin >> num;
    if(num == 0) break;
    somma = somma + num;
}
```

Per concludere questa sezione di confronto tra `while` e `for` chiediamoci se è possibile esprimere ogni ciclo `while` tramite un vero ciclo `for` ad iterazione limitata, cioè quello ad indice su un opportuno intervallo $[A, B]$, vista nel capitolo precedente.

La risposta è in generale negativa, infatti la condizione di un ciclo `while` potrebbe dipendere da dati che mano a mano arrivano al computer, ma che non sono disponibili all'inizio del ciclo, per cui in tali situazioni è materialmente impossibile determinare a priori il numero di iterazioni necessarie.

Ma anche nel caso in cui i dati fossero già tutti disponibili, esistono esempi di funzioni che non si possono calcolare con cicli `for` su intervallo, ma che necessitano di cicli `while`. Un esempio di funzione con questa caratteristica è la funzione di Ackermann (in termini tecnici si dice che questa funzione è

ricorsiva generale, ma non è *ricorsiva primitiva*). Il lettore interessato può trovare la definizione su qualsiasi testo di teoria della calcolabilità.

L'uso dei cicli `while` però solleva il problema della terminazione: è possibile scrivere cicli `while` che non terminano mai, ciò accade quando la condizione resta sempre vera. Un uso corretto di `while` impone che il programmatore si accerti, ad esempio con una dimostrazione matematica o mediante un'argomentazione convincente, che prima o poi la condizione diventa falsa.

Si noti che un programma che contiene un ciclo `while` è quindi potenzialmente non terminante, mentre l'uso dei cicli `for` (quando è scritto correttamente nella forma ad iterazione limitata vista nel capitolo precedente) garantisce la terminazione in qualsiasi situazione.

Perciò da un lato sarebbe auspicabile non usare i cicli `while`, ma d'altro canto alcune situazioni necessitano proprio di tale tipo di istruzione. Perciò la conclusione a cui si giunge è la seguente: un linguaggio di programmazione può essere universale (cioè permettere il calcolo di tutte le funzioni calcolabili) se e solo se consente di scrivere programmi non terminanti.

9.6 Esercizi (da risolvere con il ciclo while)

1. Scrivere un programma che, dato un numero naturale n, trova il più piccolo divisore (eccetto 1) di n.

2. Scrivere un programma che, dati un numero naturale m ed un numero primo p, trova l'inverso moltiplicativo di m modulo p, cioè quel numero n tale che $m \cdot n = 1 \pmod{p}$.

3. Scrivere un programma che, dati m un numero naturale e p un numero primo, trova una radice quadrata di m modulo p, scrivendo "no" se m non è un quadrato modulo p.

4. Scrivere un programma che, dato un numero naturale n, trova il più piccolo numero primo maggiore o uguale a n. Suggerimento: trovare un numero primo tra $n, n+1, n+2, \ldots$.

5. Scrivere un programma che, dato un numero naturale n, scrive sullo schermo i primi n numeri primi. Ad esempio con $n = 7$ la funzione deve scrivere 2,3,5,7,11,13,17.

6. Scrivere un programma che, dato un numero naturale n, trova la somma delle cifre decimali di n. Suggerimento: partendo ad esempio da n=12345, si noti che `n%10` fa 5, mentre `n/10` è 1234

7. Scrivere un programma che, dati un numero naturale m ed un numero primo p, trova l'ordine di m modulo p, ovvero il più piccolo numero k tale che $m^k \pmod{p} = 1$.

8. Scrivere un programma che legge una serie di numeri reali da tastiera e si ferma quando l'utente inserisce 0, scrivendo alla fine la media aritmetica dei numeri letti.

9. Come nell'esercizio precedente, calcolare il massimo ed il minimo dei numeri letti.

10. Scrivere un programma che, tramite l'espressione `rand()%100+1`, sceglie un numero X a caso compreso 1 e 100. L'utente deve indovinare X e ha un numero illimitato di tentativi. Ad ogni tentativo digita un numero Y e il computer deve indicare se $X > Y$, $Y < X$ o se ha indovinato. In tal caso il gioco termina e il computer indica quanti tentativi sono stati usati. Per evitare che il programma scelga sempre lo stesso numero, inserire l'istruzione `srand(time(0))` all'inizio di *main*. Includere le librerie *cstdlib* e *ctime*.

11. Come nell'esercizio precedente, limitare il numero di tentativi. Se entro il limite, l'utente non ha indovinato, visualizzare il numero corretto.

12. Utilizzando la soluzione dell'esercizio 1, scrivere un programma che dato un numero naturale n scrive sullo schermo una scomposizione in fattori primi di n. L'idea è la seguente: partendo da n, si ottenga il più piccolo divisore d di n, dato che d è un numero primo, scrivere d sullo schermo e dividere n per d. Ripetere quindi tale procedimento fino a che n è diventato 1.

Chapter 10

Tipi di dati strutturati

Il C++, al pari dei comuni linguaggi di programmazione, offre la possibilità di lavorare con array ed altri tipi di dati strutturati.

Un tipo di dato si dice **strutturato** se è ottenuto mediante aggregazione di altri dati.

La forma più semplice di aggregazione è l'**array**, in cui si aggregano dati dello stesso tipo. Un array costituisce una generalizzazione del concetto matematico di vettore. Infatti gli elementi di un array non sono necessariamente numeri, ma è possibile avere array di caratteri, di stringhe o di altri tipi. Inoltre un array può essere disposto su più dimensioni e quindi avere più indici.

In queste dispense useremo solo array ad una ed a due dimensioni con elementi numerici, che corrispondono ai concetti matematici di vettore e matrice.

Un'altra forma di aggregazione usata nella programmazione è quella in cui i dati aggregati sono di tipi diversi. Tale forma si chiama record (o struttura), che vedremo in una sezione successiva di questo capitolo. Altre tipologie di aggregazione (vector, liste, insiemi, mappe) saranno descritte nel capitolo relativo alle STL.

10.1 Array unidimensionali

Per dichiarare una variabile di tipo array bisogna specificare il tipo T ed il numero n (detto **dimensione** dell'array) degli elementi che ne fanno parte. La sintassi è quindi

```
T v [ n ]
```

in cui v è il nome della variabile array.

Ad esempio

```
int a[10];
```

dichiara una variabile a di tipo array avente 10 elementi di tipo int.

La dimensione in generale è una costante, esplicita (come nell'esempio precedente) oppure definita tramite `#define`

```
#define N 10
int a[N];
```

o tramite `constexpr`

```
constexpr int N = 10;
int a[N];
```

In queste dispense useremo questa seconda forma di dichiarazione delle costanti.

E' possibile anche inizializzare una variabile di tipo array elencando gli elementi tra parentesi graffe. Ad esempio

```
int v[5] = {4, 5, 7, 9, -1};
```

In questa situazione si può omettere la dimensione perché il compilatore la deduce dal numero di elementi elencati

```
int v[]={4, 5, 7, 9, -1};
```

Ogni elemento di un array di n elementi è identificato con un numero intero, chiamato **indice**, compreso tra 0 e n-1. Ad esempio per l'array v abbiamo

indice	0	1	2	3	4
elemento	4	5	7	9	-1

L'accesso alla i-esima componente di un vettore v, che in matematica si denota con v_i, in C++ si indica con la notazione `v[i]`.

Tale notazione ha senso solo se i è un numero intero compreso tra 0 e $n-1$, ove n è la dimensione di v (in questo caso $n = 5$). L'indice i può essere una costante, una variabile o il risultato di un'espressione a valori interi.

L'accesso alla i-esima componente di v può servire ad utilizzare il valore memorizzato, ad esempio

```
cout << v[3] << endl;
```

scriverà 9 sullo schermo.

L'accesso può anche servire anche per modificare il valore memorizzato nella componente, tramite un'operazione di assegnamento o di lettura da tastiera. Ad esempio dopo l'istruzione

```
v[2] = 8;
```

l'array diventa

indice	0	1	2	3	4
elemento	4	5	**8**	9	-1

Il linguaggio C++ non controlla se l'indice i di `v[i]` abbia un valore compreso tra 0 e $n-1$. Questo controllo è lasciato al programmatore, il quale deve far sì che l'indice sia compreso nell'intervallo giusto. Spesso, ma non sempre, se l'indice non ha un valore corretto si può generare durante l'esecuzione del programma un errore del tipo "segmentation fault", che comporta la terminazione del programma.

Si noti che non esiste alcuna operazione per cambiare la dimensione di un array: la dimensione è fissata al momento della dichiarazione. Esistono in realtà altre strutture dati in C++ (vector e liste) che consentono di memorizzare collezioni con un numero variabile di elementi e che saranno illustrate in un capitolo successivo.

Per trattare una collezione con un numero variabile di elementi tramite un array è possibile dimensionare l'array con il massimo numero di elementi che si vogliono avere e in seguito di usare solo le componenti di cui si ha bisogno. Ovviamente tale tecnica in generale produce uno spreco di memoria.

10.2 Implementazione delle operazioni elementari sugli array

In C++ gli array non hanno nessuna operazione predefinita. Infatti non è possibile, tramite una singola istruzione, leggere da tastiera un array, scrivere gli elementi sullo schermo, e nemmeno copiare, tramite un singolo assegnamento, un array in un altro.

Perciò nel trattamento degli array riveste particolare importanza l'uso del ciclo for. Infatti per svolgere la stessa operazione su tutti gli elementi di un array a di N elementi si può usare un indice che viene fatto variare mediante un ciclo for sull'intervallo $\{0, \ldots, N-1\}$:

```
for(int i = 0; i < N; i++) {
    // esegui l'operazione su a[i]
}
```

Useremo questo schema per implementare alcune operazioni di base sugli array.

Negli esempi seguenti, a sarà un array di N numeri interi, in cui N è una costante definita con la direttiva `constexpr int N = ...`.

10.2.1 Riempimento

Per riempire gli elementi dell'array array ad esempio con il valore costante 0 si scrive

```
for(int i = 0; i < N; i++) {
    a[i] = 0;
}
```

oppure se il valore è calcolato in funzione di i, ad esempio i^2

```
for(int i = 0; i < N; i++) {
    a[i] = i * i;
}
```

10.2.2 Copia

Per copiare tutti gli elementi di un array a in un array b di uguale dimensione (va bene anche se b ha una dimensione maggiore di N), si scrive

```
for(int i = 0; i < N; i++) {
    b[i] = a[i];
}
```

Questa operazione è in sostanza l'equivalente di un assegnamento tra array. Si noti invece che scrivere `a=b` genera un errore.

10.2.3 Lettura da tastiera

Per leggere da tastiera gli elementi di a si usa

```
cout << "Inserisci gli elementi ";
for(int i = 0; i < N; i++) {
    cin >> a[i];
}
```

Anche in questo caso genera errore usare direttamente `cin >> a`.

10.2.4 Scrittura su schermo

Per scrivere gli elementi di a su una riga dello schermo si usa

```
for(int i = 0; i < N; i++) {
cout << a[i] << " ";
}
```

Se al posto dello spazio finale si mette `endl` si dispongono gli elementi in colonna. Si noti che `cout << a` è accettato dal compilatore, ma scrive sullo schemo l'indirizzo in RAM in cui è memorizzato l'array.

10.2.5 Sommatoria e calcolo del massimo

Per calcolare la somma di tutti gli elementi di a si può scrivere

```
int somma = 0;
for(int i = 0; i < N; i++) {
    somma = somma + a[i];
}
```

in cui è aggiornata la variabile *somma*, che contiene la somma parziale dei primi i elementi.

Per trovare il massimo elemento di a si usa un procedimento che si basa sul calcolo del massimo parziale. La variabile m è inizializzata con il primo elemento, così il ciclo può partire da 1.

```
int m = a[0];
for(int i = 1; i < N; i++) {
    if(a[i] > m) {
        m = a[i];
    }
}
```

In maniera alternativa si può inizializzare m con il più piccolo valore possibile e usare un ciclo che parte da 0:

```
int m = -INT_MAX - 1; // è il più piccolo valore int a 32 bit
for(int i = 0; i < N; i++) {
    if(a[i] > m) {
        m = a[i];
    }
}
```

In realtà è sufficiente che m sia inizializzato con un valore più piccolo di ogni elemento di a.

Per trovare la posizione del massimo (corrispondente alla funzione matematica *argmax*), è necessario usare un secondo indice che viene aggiornato ogni volta che si trova un elemento più grande

```
int imax = 0;
for(int i = 1; i < N; i++) {
    if(a[i] > a[imax]) {
        imax = i;
    }
}
```

Se il massimo è presente in più posizioni, con questa soluzione si trova la posizione più a sinistra, cioè quella con l'indice più piccolo. In maniera del tutto analoga è possibile calcolare il valore minimo o la posizione del minimo di un array.

10.2.6 Conteggio

Per contare quanti elementi di a godono di una determinata proprietà (nell'esempio, quella di essere pari) si usa un contatore che viene incrementato di uno ogni qual volta che si trova un elemento che gode della proprietà

```
int conta = 0;
for(int i = 0; i < N; i++) {
    if(a[i] % 2 == 0) {
        ++conta;
    }
}
```

10.2.7 Quantificazione esistenziale e universale

Per controllare se esiste un elemento all'interno dell'array che gode di una data proprietà (nell'esempio, sempre quella di essere pari), si può usare un ciclo for con un break

```
bool trovato = false;
for(int i = 0; i < N; i++) {
    if(a[i] % 2 == 0) {
        trovato = true;
        break;
```

```
    }
}
```

In maniera alternativa, è possibile risolvere lo stesso problema con un'istruzione while

```
bool trovato = false;
int i = 0;
while(i < N && trovato == false) {
    if(a[i] % 2 == 0) {
        trovato = true;
    }
    else {
        i++;
    }
}
```

Alla fine del ciclo while o del for, la variabile *trovato* è *true* se e solo se esiste un elemento che gode della proprietà richiesta. Inoltre, in caso positivo, `a[i]` è il primo elemento a partire da sinistra che gode della proprietà.

Per controllare se tutti gli elementi godono di una certa proprietà è sufficiente controllare che non esiste un elemento che gode della proprietà contraria (essere dispari). Ecco le due soluzioni, la prima con il ciclo for

```
// versione con for/break
bool tutti = true;
for(int i = 0; i < N; i++) {
    if(a[i] % 2 != 0) {
        tutti = false;
        break;
    }
}
```

e la seconda con un ciclo while

```
// versione con while
bool tutti = true;
int i = 0;
while(i < N && tutti == true) {
    if(a[i] % 2 != 0) { // trovato un elemento dispari
        tutti=false;
```

```
    }
    else {
        i++;
    }
}
```

Alla fine del ciclo, la variabile *tutti* è (rimasta) *true* se e solo se esiste ogni elemento di a gode della proprietà richiesta.

10.2.8 Appartenenza

Un caso particolare dell'operazione di quantificazione esistenziale è quella di controllare se in a è presente un particolare valore x.

```
bool trovato = false;
int i = 1;
while (i < N && trovato == false) {
    if(a[i] == x) {
        trovato = true;
    }
    else {
        i++;
    }
}
```

Anche in questo esempio, alla fine del ciclo while la variabile *trovato* è *true* se e solo se x è presente come elemento di a e, in tal caso, i è la posizione di x in a (più precisamente la prima posizione di x in a partendo da sinistra).

10.2.9 Filtro

Per copiare in un array b solo gli elementi di a che godono di una certa proprietà (nell'esempio, quella di essere pari) si scrive

```
int k = 0;
for(int i = 0; i < N; i++) {
    if(a[i]%2 == 0) {
        b[k] = a[i];
        k++;
    }
}
```

In questa operazione gli elementi di a sono copiati in posizioni consecutive in b senza lasciare spazi vuoti. A questo scopo si usa l'indice k, che viene incrementato solo quando l'elemento di a viene selezionato.

Ad esempio se a contiene gli elementi $(5, 4, 2, 8, 7, 9, 10, 1)$, b conterrà $(4, 2, 8, 10)$, mentre le altre posizioni saranno vuote (o meglio inalterate). Gli elementi effettivamente inseriti in b sono k, in questo caso 4.

Se non si conosce a priori il numero di elementi di a che saranno selezionati, la dimensione di b deve essere uguale a N, che è il massimo valore possibile.

10.2.10 Somma tra vettori e prodotto per uno scalare

Negli esempi precedenti, abbiamo visto esercizi in cui gli array sono visti solo dal punto di vista informatico.

In realtà, gli array si possono anche vedere come vettori. A titolo di esempio ecco come calcolare la somma vettoriale di due array di numeri reali u e v in un terzo array w (tutti della stessa dimensione N):

```
for(int i = 0; i < N; i++) {
    w[i] = u[i] + v[i];
}
```

Invece per calcolare il prodotto di un vettore v per uno scalare k, ottenendo un nuovo vettore w si scrive

```
for(int i = 0; i < N; i++) {
    w[i] = k * v[i];
}
```

10.3 Array multidimensionali

Gli array multidimensionali sono una generalizzazione degli array visti fino ad adesso, che sono appunto monodimensionali. Tratteremo in queste dispense solo gli array bidimensionali (che chiameremo per brevità matrici). Comunque il modo di lavorare con array a tre o più dimensioni è analogo a quello relativo alle matrici.

Una matrice ha due dimensioni e viene rappresentata graficamente mediante una tabella, in cui la prima dimensione rappresenta il numero di righe e la seconda il numero di colonne. Ogni elemento è identificato dal numero di riga e di colonna (una sorta di coordinate cartesiane).

	colonne			
righe	0	1	2	3
0	7	8	-5	2
1	5	2	1	9
2	4	-3	-1	0

Una matrice si dichiara con la sintassi

T m [n1] [n2]

in cui T è il tipo degli elementi, m il nome dell'array, $n1$ e $n2$ le due dimensioni, ovvero il numero di righe ed il numero di colonne.

Ad esempio con

```
double a[3][4];
```

si dichiara una variabile a di tipo matrice di numeri reali con 3 righe e 4 colonne.

Per accedere agli elementi di una matrice si usano due indici. Ad esempio l'istruzione

```
r = a[1][2];
```

assegna alla variabile r il valore 1.

Una matrice può essere inizializzata in modo simile a quanto avviene per un array monodimensionale

```
double a[3][3]={ {1, 2, 3}, {3, -2, 1},{8, 9, 1}};
```

Per gestire una matrice a di M righe e N colonne si usa in generale un doppio ciclo for

```
for(int i = 0; i < M ;i++) {
    for(int j = 0; j < N; j++) {
        // fai qualcosa con a[i][j]
    }
}
```

Nel prosieguo M e N sono due costanti dichiarate con constexpr

```
constexpr int M = ...;
constexpr int N = ...;
```

Ad esempio per scrivere sullo schermo il contenuto della matrice a si scrive

```
for(int i=0; i<M; i++) {
    for(int j = 0; j < N; j++) {
        cout << a[i][j] << " ";
    }
    cout << endl;
}
```

Tutte le altre operazioni elementari viste per gli array monodimensionali possono essere facilmente modificate per lavorare su array bidimensionali, inserendo sempre un doppio ciclo for, come ad esempio la lettura da tastiera

```
for(int i = 0; i < M; i++) {
    for(int j = 0; j < N; j++) {
        cin >> a[i][j];
    }
}
```

o la copia della matrice a in una matrice b di pari dimensioni

```
for(int i = 0; i < M; i++) {
    for(int j = 0; j < N; j++) {
        b[i][j] = a[i][j];
    }
}
```

10.4 Operazioni matriciali

Per calcolare la somma di due matrici a e b (di dimensioni $M \times N$), memorizzando il risultato in una matrice c con le stesse dimensioni si può scrivere

```
for(int i = 0; i < M; i++) {
    for(int j = 0; j < N; j++) {
c[i][j] = a[i][j] + b[i][j];
    }
}
```

Per calcolare il prodotto tra una matrice a di dimensioni $M \times N$ e un vettore v di dimensione N, ottenendo come risultato un vettore u di dimensione m si può scrivere

```
for(int i = 0; i < M; i++) {
    double somma = 0;
```

```
    for(int j = 0; j < N; j++) {
        somma = somma + a[i][j] * v[j];
    }
    u[i] = somma;
}
```

Infine, per calcolare il prodotto righe per colonne tra una matrice a, di dimensioni $M \times P$, e una matrice b, di dimensioni $P \times N$, memorizzando il risultato in una matrice c di dimensioni $M \times N$ si può scrivere

```
for(int i = 0; i < M; i++) {
    for(int j = 0; j < N; j++) {
        double somma = 0;
        for(int k = 0; k < P; k++) {
somma = somma + a[i][k] * b[k][j];
        }
        c[i][j] = somma;
    }
}
```

Infatti tale codice è basato sulla formula

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$$

per $i = 1, \ldots, m$ e $j = 1, \ldots, n$.

10.5 Le stringhe

Le stringhe, già introdotte in un capitolo precedente, sono per certi versi assimilabili ad array di caratteri, ma con molte differenze. Una variabile di tipo stringa può infatti contenere una sequenza arbitrariamente lunga di caratteri

```
string nome = "Marco";
string cognome = "Baioletti";
string frase = "questa frase e' cosi' lunga che non entra in una riga di queste dispens
```

E' possibile leggere direttamente da tastiera una stringa

```
string s;
cout << "Inserisci una stringa ";
cin >> s;
```

Se però la stringa contiene degli spazi si deve usare la funzione *getline*

```
cout << "Inserisci una stringa ";
getline(cin, s);
```

Si può scrivere una stringa sullo schermo direttamente

```
cout << nome << " " << cognome << endl;
```

Infine è possibile assegnare una stringa ad una variabile tramite un assegnamento diretto

```
s = frase;
```

Si noti che tutte queste operazioni non sono consentite con gli array. Comunque le stringhe si possono trattare anche come array. Ad esempio `nome[0]` vale `'M'`, che è il primo carattere della stringa *nome*.

Si può anche cambiare il valore di un carattere in una stringa, ad esempio

```
frase[0] = 'Q';
```

Per sapere di quanti caratteri è composta una stringa si usa il "metodo" *length*

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

In questo esempio, la lunghezza di *nome*, ovvero 5, è memorizzata nella variabile *n*.

Le stringhe si possono concatenare con l'operatore `+`. Ad esempio

```
string nominativo;
nominativo = nome + " " + cognome;
```

E' possibile infine scomporre una stringa, prendendone una parte, con il "metodo" *substr*, che vuole due parametri: il primo è l'indice del primo carattere da estrarre, il secondo quanti caratteri estrarre. Ad esempio

```
s = frase.substr(26, 5);
```

mette nella variabile *s* la stringa "lunga".

10.6 Le Struct

Le **struct** (dette anche **record**) sono tipi di dati ad aggregazione eterogenea. Una variabile di tipo struct è composta da **campi**, ognuno dei quali ha un proprio nome e un tipo di dato. Si può dichiarare un nuovo tipo di dato *struct* con la seguente sintassi

```
struct nome_tipo {
    tipo1 nome1,
    ...
    tipoN nomeN;
};
```

in cui *tipo1*,. . . ,*tipoN* sono i tipi dei campi e *nome1*, . . . , *nomeN* sono i rispettivi nomi.

Ad esempio con

```
struct persona {
    string nome;
    string cognome;
    int g_nasc;
    int m_nasc;
    int a_nasc;
};
```

si dichiara un tipo di dato *persona*, composto da cinque campi: *nome* e *cognome*, di tipo stringa, e i tre numeri interi *g_nasc*, *m_nasc* e *a_nasc*, che formano la data di nascita. A questo punto si possono dichiarare e usare una o più variabili di tipo *persona*

```
int main() {
    persona p1,p2,p3;
    ...
}
```

E' possibile accedere ad un campo di una variabile di tipo struct con la notazione `nome_variabile . nome_campo` sia per ottenere il valore del campo, sia per modificarlo.

Un campo di una variabile di tipo *struct* è assimilabile ad una variabile dello stesso tipo e si può usare in tutti i contesti in cui si potrebbe usare tale variabile. Ad esempio

```
int main() {
    persona p;
    p.nome = "Mario";
    p.cognome = "Rossi";
    cout << p.nome << " " << p.cognome << endl;
}
```

Le differenze principali tra array e struct sono evidenti:

- i campi di una struct hanno un nome, gli elementi di un array sono numerati con un indice;
- per definire un array è sufficiente indicare il tipo degli elementi e il loro numero, per una struct è indispensabile elencare i campi ad uno ad uno;
- nelle struct non ha senso il concetto di indice e quindi non si può utilizzare il ciclo for.

Come per gli array, non sono previste operazioni di input e output dirette. Però è possibile usare l'assegnamento, che copia ad uno ad uno i campi da una variabile di tipo struct ad un'altra dello stesso tipo. Ad esempio

```
int main() {
    persona p1, p2;
    p1.nome = "Mario";
    p1.cognome = "Rossi";
    ...
    p2 = p1;
    cout << p2.cognome << endl;
}
```

Inoltre è possibile inizializzare una variabile di tipo struct fornendo un valore iniziale per ciascun campo

```
persona p = {"Mario", "Rossi", 1, 1, 1970};
```

Una struct si usa in generale per mettere insieme valori appartenenti ad uno stesso concetto, come le coordinate e le velocità di un punto, i dati di un veicolo o le caratteristiche di uno strumento.

10.7 Array di struct

E' molto utile creare array di struct: ciò corrisponde ad avere l'equivalente in memoria di una tabella di un database o di un foglio elettronico. Infatti un array di struct si può immaginare come un oggetto rettangolare, in cui le righe sono gli elementi dell'array e le colonne sono i campi della struct. Ad esempio, l'array *popolazione*

```
int main() {
    persona popolazione[5] = {
          {“Mario”, ”Rossi”, 1, 1, 1970},
          {“Maria”, ”Verdi”, 2, 2, 1980},
          {“Laura”, ”Neri”, 3, 3, 1981},
          {“Luisa”, ”Gialli”, 4, 4, 1984},
          {“Giorgio”, ”Rosi”, 5, 5, 1975}
    };
    ...
}
```

corrisponde alla seguente tabella

nome	cognome	g_nasc	m_nasc	a_nasc
Mario	Rossi	1	1	1970
Maria	Verdi	2	2	1980
Laura	Neri	3	3	1981
Luisa	Gialli	4	4	1984
Giorgio	Rosi	5	5	1975

Per accedere ad un dato della variabile *popolazione* devono essere indicati l'indice della riga e il nome del campo, ad esempio `popolazione[3].g_nasc`.

10.8 Esercizi

Vettori e matrici hanno in questi esercizi dimensioni solitamente indicati con delle costanti.

1. Scrivere un programma che dato un array x di N numeri interi, scrive sullo schermo l'array con una notazione matematica del tipo $(5, 1, 10, 4, 3)$.

2. Scrivere un programma che dato un array x di N numeri reali e un numero intero h, minore di N, calcola la somma degli elementi dal h-esimo elemento in poi.

3. Scrivere un programma che dato un array x di N numeri reali, trova il più grande elemento tra quelli di posizione dispari.

4. Scrivere un programma che calcola la norma in L_p di un array x di N numeri reali, ovvero

$$||x||_p = \sqrt[p]{\frac{\sum_{i=1}^{n} |x_i|^p}{n}}$$

5. Scrivere un programma che concatena due array x e y di N numeri interi in un array z di numeri interi avente una dimensione sufficiente allo scopo.

6. Scrivere un programma che dati due array x e y di N numeri interi, crea un array che prende gli elementi da x e da y in modo alternato, ad esempio da $(1, 2, 3)$ e $(4, 5, 6)$ crea $(1, 4, 2, 5, 3, 6)$.

7. Scrivere un programma che dato un array x di N numeri interi, crea un array identico ad x, senza però l'ultimo elemento.

8. Scrivere un programma che dato un array x di N numeri interi, riposiziona gli elementi di x nell'ordine inverso: ad esempio (1,2,3,4) diventa (4,3,2,1).

9. Scrivere un programma che dato un array x di N numeri interi, copia in un array y gli elementi di x che sono divisibili per 5 ma non per 7.

10. Scrivere un programma che dato un array contenente i coefficienti di un polinomio p (ad esempio $(1, 2, 3)$ rappresenta il polinomio $p(x) = 1 + 2x + 3x^2$) e un numero reale t, calcola $p(t)$.

11. Nelle ipotesi dell'esercizio precedente, calcolare $p'(t)$.

12. Nelle ipotesi dell'esercizio precedente, calcolare

$$\int_a^b p(t)dt$$

ove a, b sono letti da tastiera.

13. Scrivere un programma che controlla se un dato array di N numeri interi è palindromo, ovvero è uguale a se stesso anche rovesciando l'ordine dei suoi elementi. Ad esempio $(1, 2, 3, 2, 1)$ e $(1, 2, 3, 3, 2, 1)$ sono palindromi.

14. Scrivere un programma, che dati due array x e y di numeri interi, rappresentanti due insiemi, calcola l'intersezione, ovvero gli elementi in comune, mettendo il risultato in un array z. Suggerimento: usare la parte di programma che controlla se un elemento fa parte di un array e poi il codice del filtro di un array, dato che

15. Nelle ipotesi dell'esercizio precedente, calcolare l'unione di due insiemi rappresentati con due array.

16. Scrivere un programma che calcola il determinante di una matrice 3×3.

17. Scrivere un programma che crea la matrice identità di dimensione N.

18. Scrivere un programma che, data una matrice quadrata A di $N \times N$ numeri reali, calcola la traccia, ovvero la somma degli elementi sulla diagonale principale.

19. Scrivere un programma che, data una matrice A di $M \times N$ numeri reali, trova l'elemento più grande di A in valore assoluto.

20. Scrivere un programma che, data una matrice A di $N \times N$ numeri reali, controlla se A è triangolare superiore, ovvero che ogni elemento al di sotto della diagonale principale è nullo.

21. Scrivere un programma che, data una matrice A di $M \times N$ numeri reali, riempe un array con le somme degli elementi di ciascuna colonna.

22. Scrivere un programma che, data una matrice A di $M \times N$ numeri reali, restituisce il numero della riga con il maggior numero di zeri (o -1, se nessuna riga ha zeri).

23. Scrivere un programma che, data una matrice quadrata A di $N \times N$ numeri reali, calcola $B = A \times A$.

24. Scrivere un programma che, data una matrice quadrata A di $N \times N$ numeri reali, aggiunge alla prima riga un multiplo opportuno della seconda riga di modo che l'elemento di posto $(1, 1)$ sia zero.

25. Scrivere un programma che, data una matrice A di $N \times N$ numeri reali triangolare superiore e un vettore b di dimensione N, calcola la soluzione del sistema lineare

$$Ax = b$$

Suggerimento: il valore di x_N è facile da trovare, da questo si può trovare x_{N-1}, e così via.

26. Scrivere un programma che, dati una matrice quadrata A di $N \times N$ numeri reali e un numero naturale k, calcola $B = A \times A \times \cdots \times A$ (k volte), ovvero A^k.

27. Scrivere un programma che, dati una matrice quadrata A di $N \times N$ numeri reali e un numero naturale k, calcola $I + A + 1/2A^2 + 1/3!A^3 + \cdots + 1/k!A^k$.

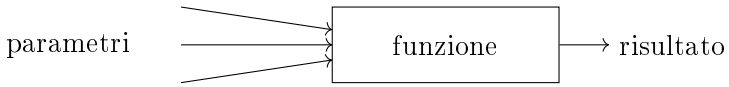
28. Scrivere un programma che data una matrice quadrata A di $N \times N$ numeri reali, trasforma A in forma triangolare tramite il metodo di Gauss. (esercizio difficile, si basa sulla soluzione dell'esercizio 24)

Chapter 11

Funzioni

In questo capitolo vedremo in dettaglio la sintassi, la semantica (intuitiva) e le principali caratteristiche delle funzioni definite dall'utente, fornendo alla fine una serie di motivazioni per il loro impiego. Il termine **funzione definita dall'utente** viene utilizzato in questo capitolo in opposizione al concetto di **funzione predefinita**, come ad esempio `cos`, `sin` o `log`, cioè funzioni già presenti nel linguaggio C++.

In generale una funzione è una parte di programma, separata dal resto, che riceve dall'esterno dei dati attraverso uno o più parametri e che produce un risultato, calcolato a partire dai dati ricevuti tramite l'esecuzione delle sue istruzioni.



E' comunque possibile avere funzioni senza parametri e funzioni senza risultati, come si vedrà in seguito.

La funzione può essere eseguita tramite una chiamata che può utilizzare il risultato ottenuto dalla funzione all'interno di un'espressione.

11.1 Sintassi

La gestione di una funzione distingue due momenti differenti: la definizione e la chiamata.

Nella **definizione** si crea una nuova funzione indicandone il dominio, il codominio e il comportamento. Una funzione in generale viene definita con la seguente sintassi

```
tipo_risultato nomeFunzione(elenco_parametri) {
    corpo_della_funzione
}
```

in cui

- *tipo_risultato* è il tipo del risultato che la funzione restituisce tramite l'istruzione `return`, oppure `void` se non c'è un risultato esplicito;
- *nomeFunzione* è il nome della funzione e deve essere un identificatore;
- *parametri* è un elenco di variabili, separate da virgole, ognuna preceduta dal corrispondente tipo di dato; se la funzione non ha parametri si deve comunque mettere una coppia di parentesi tonde;
- *corpo_della_funzione* è costituito dalle istruzioni interne alla funzione.

Il tipo dei parametri corrisponde al concetto matematico di dominio della funzione, mentre il tipo del risultato corrisponde al codominio.

Nella **chiamata** la funzione viene valutata su determinati argomenti. La chiamata di una funzione ha la sintassi

```
nomeFunzione(argomenti)
```

in cui *argomenti* è un elenco di espressioni separate da virgole. Se la funzione non ha parametri, è obbligatorio mettere una coppia di parentesi tonde anche nella chiamata.

E' importante notare che una funzione viene definita una sola volta, ma può essere chiamata tante volte, anche all'interno di altre funzioni.

Come primo esempio definiamo e chiamiamo una funzione senza parametri e senza risultati, il cui unico scopo è quello di scrivere una frase sullo schermo.

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

void prova() {
    cout << "sono nella funzione prova\n";
}
int main() {
    cout << "sono nel main\n";
    prova();
    cout << "sono di nuovo nel main\n";
    prova();
}
```

si otterrà sullo schermo

```
sono nel main
sono nella funzione prova
sono di nuovo nel main
sono nella funzione prova
```

Si noti che in *main* la funzione *prova* è stata chiamata due volte: ogni volta sono state eseguite le istruzioni al suo interno. In questo primo esempio *tipo_risultato* è `void`, in quanto la funzione non restituisce alcun risultato non ci sono parametri.

Come secondo esempio scriviamo una funzione che calcola il fattoriale di un numero intero.

```
int fattoriale(int n) {
    int f = 1;
    for(int i = 1; i <= n; i++) {
        f = f * i;
    }
    return f;
}

int main() {
    int num;
    cout << "inserisci un numero ";
    cin >> num;
    int ris = fattoriale(num);
    cout << "il suo fattoriale e' " << ris << endl;
}
```

Nel secondo esempio invece

- *tipo_risultato* è `int`, in quanto `return` restituisce un numero intero
- il nome della funzione è *fattoriale*
- n è l'unico parametro ed è di tipo `int`.

Si noti che nella funzione *fattoriale* sono definite due variabili (i e f). Queste sono **variabili locali** della funzione.

Infine, nel *main* la funzione *fattoriale* è stata chiamata usando come argomento *num*.

Una chiamata ad una funzione è valida solo quando il numero degli argomenti forniti è pari al numero dei parametri nella definizione della funzione e ogni argomento deve essere di un tipo compatibile con il tipo del corrispondente parametro.

Infatti al momento della chiamata si stabilisce una corrispondenza biunivoca (temporanea) tra argomenti e parametri. Per esempio, se una funzione ha un parametro di tipo *int*, allora in ogni chiamata l'argomento associato deve essere un *int*, altrimenti si genera una situazione di errore.

Ad esempio, avendo la seguente definizione di funzione

```
double potenza(double x, int n) {
    double p = 1;
    for(int i = 1; i <= n; i++) {
        p = p * x;
    }
    return p;
}
```

una chiamata corretta è

```
int main() {
    double z = 2, r;
    r = potenza(3.1 * z, 4);
    ...
}
```

in cui l'argomento `3.1*z`, il cui valore è 6.2, corrisponde al parametro x e l'argomento 4 corrisponde al parametro n. Invece esempi di chiamate scorrette sono

```
r = potenza(z / 2, 4, -17);
r = potenza(z);
r = potenza(z, "quattro");
```

Infatti la prima chiamata ha troppi argomenti, la seconda ne ha pochi; nella terza invece il secondo argomento è una stringa, mentre dovrebbe essere un `int`.

11.2 Semantica della chiamata

Il fatto che negli esempi le funzioni sono state chiamate in *main* è solo per semplicità: in realtà una funzione può chiamarne un'altra. Inoltre si noti che anche *main* è una funzione particolare: quando si avvia un programma, l'esecuzione parte da essa.

Supponiamo che una funzione f chiama una funzione g con argomenti $a_1, \ldots, a_n$, ovvero con la chiamata `g(a1,...,an)`, mentre siano $p_1, \ldots, p_n$ i parametri di g. Vengono allora nell'ordine le seguenti azioni:

1. L'esecuzione di f è arrestata e l'esecutore memorizza il punto in cui f è stata interrotta.

2. Le variabili della funzione g sono allocate, ovvero viene associato uno spazio in memoria ad ognuna di esse.

3. Gli argomenti della chiamata $a_1, \ldots, a_n$ sono valutati e sono memorizzati nei corrispondenti parametri di g: il valore di a_1 viene memorizzato nel primo parametro p_1, il valore di a_2 in p_2, e così via.

4. Inizia l'esecuzione di g.

5. Ad un certo momento l'esecuzione di g finisce, ciò accade quando è stata eseguita l'istruzione `return` o perché le sue istruzioni sono finite.

6. Le variabili create in g, compresi i parametri $p_1, \ldots, p_n$, sono eliminate dalla memoria (deallocate).

7. L'esecuzione di f riprende dal punto in cui si era interrotta, ottenendo l'eventuale risultato prodotto da g, cioè il valore restituito dall'istruzione `return`.

Si noti che la chiamata ad una funzione sospende momentaneamente l'esecuzione di f, ovvero il linguaggio C++ non prevede l'esecuzione di più funzioni contemporaneamente. Del resto, se f ha bisogno subito del risultato di g, non può far altro che aspettare che g sia finita.

Il punto 3 è valido solo per il passaggio dei parametri per valore e assumerà una forma diversa se il passaggio dei parametri avviene per riferimento, come si vedrà in seguito.

11.3 Parametri e variabili locali

Una funzione f può avere due tipi di variabili locali: i **parametri** e le **variabili interne**. Le variabili di quest'ultima tipologia sono tutte quelle variabili dichiarate e utilizzate all'interno della funzione, di solito sono usate per conservare risultati intermedi.

E' importante capire che da un lato i parametri e le variabili interne sono trattate allo stesso modo:

1. sono utilizzabili solo all'interno della funzione di appartenenza;
2. tali variabili iniziano ad esistere in memoria solo quando la funzione viene chiamata;
3. quando la funzione termina, sono eliminate dalla memoria.

La prima caratteristica fa sì che una variabile interna a f resta confinata solo dentro f e nessuna funzione, anche quelle che vengono chiamate da f, vi può accedere.

La seconda e la terza caratteristica legano la vita delle variabili interne alla funzione stessa: le variabili esistono in memoria solo e quando f è in esecuzione o è sospesa durante una chiamata che f effettua verso altre funzioni. Ciò comporta un'occupazione di memoria ottimale, nel senso che sono presenti in memoria solo le variabili che realmente servono.

D'altro canto, parametri e variabili interne differiscono completamente nell'utilizzo: i parametri servono per ricevere i valori degli argomenti su cui calcolare il valore di f, mentre le variabili interne hanno solo significato all'interno di f.

E' evidente che i parametri sono strettamente collegati al dominio di f, mentre le variabili interne non hanno alcun corrispettivo sulla definizione matematica di f.

11.4 Istruzione return

L'istruzione `return` ha un duplice compito: indicare il risultato della funzione e concludere la sua esecuzione. Nelle funzioni return è sempre obbligatoria, tranne che nella funzione *main* (che implicitamente termina come se ci fosse un `return 0`) e nelle funzioni di tipo `void` (in cui non c'è un risultato da restituire).

Una funzione può avere più `return`, ma solo in percorsi distinti di esecuzione (ad esempio in due rami diversi di un `if`). Infatti l'esecuzione di

due `return` consecutive non è possibile: la prima termina l'esecuzione della funzione e la seconda non sarebbe eseguita.

Questa caratteristica si può sfruttare per risolvere in modo alternativo alcuni problemi. Ad esempio, per calcolare il minimo tra due numeri interi si può scrivere

```
int minimo(int a, int b) {
    if(a > b) {
        return b;
    }
    else {
        return a;
    }
}
```

o anche

```
int minimo(int a, int b) {
    if(a>b) {
        return b;
    }
    return a;
}
```

In questa seconda versione si sfrutta il fatto che se $a > b$, solo l'istruzione `return b` è eseguita. Quest'ultima soluzione è però sconsigliata perché poco leggibile. In generale, sarebbe preferibile avere una sola istruzione `return` in ogni funzione, ovviamente come ultima istruzione.

Infine si noti che nelle funzioni di tipo `void` si può usare l'istruzione `return` per uscire anticipatamente dalla funzione. Ad esempio

```
void prova(int n) {
    if(n < 0) {
        return;
    }
    // resto della funzione
    ...
}
```

In acluni casi, si può sfruttare il fatto che `return` interrompe l'esecuzione di una funzione per semplificare il codice. Ad esempio una funzione che verifica se un numero naturale n è primo è sufficiente scrivere

```
bool primo(int n) {
    int r = (int)(sqrt(n));
    for(int d = 2; d <= r; d++) {
        if(n%d == 0) {
        return false;
    }
}
    return true;
}
```

In questo esempio, l'istruzione `return false` viene eseguita quando si trova un divisore proprio di n. Invece, l'istruzione `return true` viene eseguita alla fine del ciclo for, cioè quando tutti i valori di d, compresi tra 2 e r, sono stati provati e per nessuno di essi accade che d divide esattamente n.

Questo schema di soluzione può essere utilizzato per controllare la verità di una formula quantificata universalmente: nell'esempio

$$\text{n è primo se e solo se } \forall d = 2, \ldots, \sqrt{n}\ d \text{ non divide } n$$

In maniera analoga, scambiando `true` con `false`, si può risolvere problemi in cui si deve verificare una formula quantificata esistenzialmente. Ad esempio, ecco una funzione che controlla se un numero ha almeno una cifra pari a 3

```
bool cifra3(int n) {
    while(n != 0) {
    // si controlla se n finisce con 3
     if(n %10 == 3) {
     return true;
}
        // prendendo il quoziente della divisione con 10
        // e' come se si eliminasse l'ultima cifra
        n = n / 10;
    }
    return false;
}
```

11.5 Passaggio per valore

Senza ulteriori specificazioni, in C++ tutti i parametri, ad eccezione degli array, sono **passati per valore**: il valore dell'argomento è copiato nel rispet-

tivo parametro e se la funzione modifica il parametro non si hanno ripercussioni sull'argomento (ovviamente se si tratta di una variabile).

Ad esempio nella funzione che calcola il massimo comun divisore con l'algoritmo di Euclide, che qui riportiamo per comodità del lettore

```
int mcd(int a, int b) {
    while(b != 0) {
     int r = a % b;
     a = b;
     b = r;
     }
    return a;
}
```

una chiamata del tipo

```
int main() {
    int x = 42, y = 70;
    int z = mcd(x, y);
    cout << x << " " << y << endl;
    cout << z << endl;
}
```

scrive sullo schermo

```
42 70
14
```

Ovvero, la funzione *mcd* non ha alterato il valore degli argomenti x o y, nonostante che, alla fine della sua esecuzione, i corrispondenti parametri a e b sono diventati entrambi 14. Il meccanismo di passaggio si chiama appunto per valore, in quanto la corrispondenza tra l'argomento a e il parametro p comporta solo il passaggio del valore di a in p con una sorta di assegnamento, ma non vi è alcun passaggio di dati nella direzione inversa da p ad a.

11.6 Passaggio per riferimento

In alcune situazioni si vorrebbe un passaggio dal parametro all'argomento, inverso a quanto avviene nel passaggio per valore. Ovvero, si vorrebbe definire una funzione che sia in grado di modificare uno o più argomenti della chiamata. Si noti questo esempio, un po' artificioso

```
void raddoppia(int x) {
    x=2*x;
}

int main() {
    int a = 8;
    raddoppia(a);
    cout << a << endl;
}
```

Ovviamente in tale esempio, la variabile a è non è stata raddoppiata e il programma scrive 8 sullo schermo. Per ottenere il risultato voluto è necessario passare il parametro a per riferimento. La modifica sintattica è minima: è sufficiente scrivere

```
void raddoppia(int &x) {
    x = 2 * x;
}

int main() {
    int a = 8;
    raddoppia(a);
    cout << a << endl;
}
```

In questo nuovo esempio, la variabile a è stata raddoppiata e il programma scriverà sullo schermo il numero 16.

La presenza della `&` davanti al nome di un parametro fa sì che il parametro sia **passato per riferimento** ad ogni chiamata. Un parametro passato per riferimento si comporta "come se fosse" l'argomento stesso e non una sua copia. Per cui, *raddoppia*, modificando x, in realtà modifica la variabile a di *main*.

E' importante capire che questa identificazione tra parametro e argomento è solo teorica (e tra l'altro temporanea), sebbene sia efficace per spiegare il comportamento del passaggio e il suo utilizzo. Infatti in realtà ciò che avviene "dietro le quinte" è che la variabile x contiene l'indirizzo della variabile a e ogni qual volta che la funzione tenta di accedere ad x, accede in realtà ad a.

Per capire che l'identificazione tra parametro e argomento è solo temporanea si noti quest'altro esempio di main

```
int main() {
    int a = 8, b = 11;
    raddoppia(a);
    cout << a << endl;
    raddoppia(b);
    cout << b << endl;
}
```

Nella seconda chiamata, il parametro x si "identifica" con l'argomento b.

Un altro esempio, più utile del precedente, è una funzione che scambia i valori di due variabili intere

```
void scambia(int &x, int &y) {
    int z;
    z = x;
    x = y;
    y = z;
}

int main() {
    int a = 8,b = 11;
    scambia(a, b);
    cout << a << " " << b << endl;
}
```

I valori di a e b, inizialmente 8 e 11, sono scambiati tra di loro dalla funzione *scambia*. Infatti, tale funzione scambia i valori dei due parametri x e y, come si può facilmente desumere dal codice. Ma dato che i parametri sono passati per riferimento, tale modifica si ripercuote sui corrispondenti argomenti a e b.

Uno degli usi più diffusi del passaggio per riferimento si ha quando una funzione dovrebbe restituire più risultati contemporaneamente. Come abbiamo visto precedentemente, l'istruzione `return` consente solo di restituire un unico risultato. Si supponga ad esempio di voler definire una funzione che, dati la base e l'altezza di un rettangolo, restituisce contemporaneamente l'area e il perimetro. Il modo più diretto per ottenere questo effetto è il seguente

```
void rettangolo(double base, double altezza,
        double &area, double &perimetro) {
    area = base * altezza;
```

```
    perimetro = 2 * (base + altezza);
}

int main() {
    double per, sup;
    rettangolo(10, 24,  sup, per);
    cout << "area " << sup << " perimetro " << per << endl;
}
```

Tale funzione ha due parametri passati per valore (*base* e *altezza*) e due passati per riferimento (*area* e *perimetro*). In pratica i parametri passati per valore sono i dati di partenza della funzione, mentre quelli passati per riferimento sono i risultati.

Si noti che mentre per un parametro passato per valore qualsiasi argomento dello stesso tipo è valido (variabile, costante, espressione), nel caso del passaggio per riferimento è indispensabile che il corrispondente argomento sia una variabile.

11.7 Passaggio degli array

Il passaggio degli array in C++ è svolto solo per riferimento. Il programmatore in tale situazione non può fare nulla.

Come primo esempio utilizziamo un parametro array con dimensione fissata.

```
constexpr int N = 5;

double somma_array(double a[N]) {
    double s = 0;
    for(int i = 0; i < N; i++) {
        s = s + a[i];
    }
    return s;
}

int main() {
    double arr[N] = {1, 2, 4, 5, 7};
    cout << somma_array(arr) << endl;
}
```

In realtà non è obbligatorio indicare la dimensione N nel parametro, infatti si può anche scrivere

```
constexpr int N = 5;

double somma_array(double a[]) {
    double s = 0;
    for(int i = 0; i < N; i++) {
        s = s + a[i];
    }
    return s;
}
```

In ogni caso, la funzione *somma_ array* è in grado di sommare solo array di $N = 5$ elementi. Però, in C++ è possibile anche scrivere funzioni che operano su array con un numero generico di elementi, passando come la dimensione dell'array tramite un ulteriore parametro di tipo `int`. Quindi una versione più generale di *somma_ array* è

```
double somma_array(double a[], int n) {
    double s = 0;
    for(int i = 0;i < n; i++) {
        s = s + a[i];
    }
    return s;
}

int main() {
    double arr1[5] = {1, 2, 4, 5, 7};
    cout << somma_array(arr1,5) << endl;
    double arr2[8] = {5, 2, 1, 0, 6, 3, 4, 2};
    cout << somma_array(arr2,8) << endl;
}
```

che calcola prima la somma di tutti gli elementi dell'array *arr1*, poi la somma di quelli di *arr2*.

Poiché gli array sono passati per riferimento, è possibile creare una funzione che modifica un array, ad esempio, raddoppiandone ogni elemento.

```
constexpr int N = 5;
```

```
void raddoppia_array(double a[N]) {
    for(int i = 0; i < N; i++) {
        a[i] = a[i] * 2;
    }
}

int main() {
    double arr[N] = {1, 2, 4, 5, 7};
    raddoppia_array(arr);

    for(int i = 0; i < N; i++) {
        cout << arr[i] << " ";
    }
}
```

Questa caratteristica può essere usata per ovviare al fatto che una funzione (in C++) non può restituire un array come risultato. Ad esempio, ecco una funzione che somma due vettori di $\mathbb{R}^N$ producendo come risultato un terzo vettore, sempre di $\mathbb{R}^N$

```
constexpr int N = 5;

void somma_vettori(double u[N], double v[N], double w[N]) {
    for(int i=0; i<N; i++) {
        w[i] = u[i] + v[i];
    }
}
```

Il risultato, come nell'esempio della funzione rettangolo della sezione precedente, è restituito tramite il parametro w.

Il passaggio dei parametri di tipo array a più dimensioni in C++ è simile al passaggio degli array ad una dimensione.

```
constexpr int M = 4;
constexpr int N = 3;

double somma_matrice(double a[M][N]) {
    double somma=0;
    for(int i = 0; i < M; i++) {
        for(int j = 0; j < N; j++) {
```

```
            somma = somma + a[i][j];
        }
    }
    return somma;
}

int main() {
    double m[M][N]={
     {1, 2, 3},
     {5, 6, 7},
     {10, 11, 12},
     {1, -1, 0}
     };
    double s = somma_matrice(m);
    cout << "la somma e' " << s << endl;
```

Però la funzione *somma_ matrice* non può essere generalizzata in pieno, infatti la prima dimensione del parametro può essere omessa (e quindi passata come ulteriore parametro), ma la seconda dimensione è obbligatoria. Ad esempio

```
constexpr int N = 3;
double somma_matrice(double a[][N], int nr) {
    double somma = 0;
    for(int i = 0; i < nr; i++) {
        for(int j = 0; j < N; ++j) {
            somma = somma + a[i][j];
        }
    }
    return somma;
}
```

Si noti che il numero di righe è generico, mentre il numero di colonne è fissato (in questo caso 3). Quindi la funzione si può usare solo per sommare gli elementi di una matrice $nr \times 3$, ma non si può invece definire una funzione più generale, del tipo

`double somma_matrice(double a[][], int nr, int nc)`

Vista tale limitazione del linguaggio, le funzioni che hanno delle matrici come parametri spesso sono definite per dimensioni fisse, anche perché avere libertà sul numero di righe ma non sul numero di colonne ha poco senso, soprattutto se le matrici sono quadrate.

11.8 Passaggio delle struct*

Per quanto riguarda i parametri di tipo `struct`, il passaggio può avvenire sia per valore, sia per riferimento. Nel passaggio per valore, la struct indicata come argomento è copiata, campo per campo, nel rispettivo parametro.

```
struct persona {
    string nome;
    string cognome;
    double stipendio;
};

void scrivi_persona(persona p) {
    cout << "io sono " << p.nome << " " << p.cognome << endl;
}

int main() {
    persona p;
    p.nome = "Mario";
    p.cognome = "Rossi";
    scrivi_persona(p);
}
```

In maniera analoga, è possibile usare il passaggio per riferimento

```
void scrivi_persona(persona &p) {
    cout << "io sono " << p.nome << " " << p.cognome << endl;
}
```

In questo esempio, l'unico vantaggio è l'efficienza, in quanto con questo tipo di passaggio non si ha una copia dei valori dei campi. In altre situazioni, in cui la funzione deve modificare l'argomento della chiamata, il passaggio per riferimento è d'obbligo.

```
void incrementa_stipendio(persona &p, double aumento) {
    p.stipendio = p.stipendio + aumento;
}
```

In generale, si consiglia di passare le variabili di tipo `struct` sempre per riferimento, se non altro per evitare di avere una copia dei dati, visto che una struct potrebbe avere molti campi e quindi occupare molta memoria.

11.9 Impiego delle funzioni

In questa sottosezione forniremo alcune precisazioni su come si possono impiegare le funzioni. Innanzitutto in C++ non è possibile avere funzioni annidate, cioè una funzione dentro ad un'altra funzione: ogni funzione deve essere esterna a tutte le altre. In altri linguaggi invece tale possibilità è consentita.

Un programma può essere composto da una o più funzioni, oltre a *main*. Per chiamare una funzione f da main o da un'altra funzione, f deve essere definita prima della funzione in cui avviene la chiamata. Ad esempio in

```
void funzione1() {
...
}

int funzione2(int x) {
...
}

double funzione3(double y,int z) {
...
}

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

funzione1 non può chiamare *funzione2* perché quest'ultima è definita successivamente, invece da *funzione3* è possibile chiamare sia *funzione1* che *funzione2*.

In realtà, non c'è bisogno che una funzione sia definita completamente perché possa essere chiamata, è sufficiente che sia solo dichiarata, e poi definita in seguito o anche altrove.

Per dichiarare una funzione si indica il **prototipo**, ovvero il tipo del risultato, il nome e i parametri.

Ad esempio, in

```
// dichiarazione di funzione3
double funzione3(double y, int z);

// dichiarazione di funzione2
```

```
int funzione2(int x);

// definizione di funzione3
void funzione1() {
...
    funzione2();
    funzione3();
}

// definizione di funzione3
double funzione3(double y, int z) {
...
}

// definizione di funzione2
int funzione2(int x) {
...
}
```

Ora è possibile chiamare *funzione2* e *funzione3* da *funzione1* perché queste, pur essendo definite dopo *funzione1*, sono dichiarate prima di essa.

Un altro aspetto importante da ricordare nell'impiego delle funzioni è la presenza delle variabili locali (parametri e variabili dichiarate nelle funzioni). Ogni funzione ha proprie variabili locali alle quali nessun'altra funzione può accedere. Ad esempio

```
void funzione() {
    int x, y;
    double z;
    ...
}

int main() {
    int a,b;
    double z;
    ...
}
```

in funzione non si può accedere alle variabili a e b dichiarate in main. Allo stesso tempo in *main* non si può accedere alle variabili x e y dichiarate in *funzione*. Sia in *funzione* che in *main* esiste una variabile z di tipo double,

ma si tratta di due variabili distinte: *funzione* e *main* possono accedere solo alla propria variabile z.

11.10 Importanza delle funzioni

In questa sezione si cercherà di fornire brevemente alcune motivazioni sull'uso delle funzioni.

La creazione di funzioni è un aspetto importante nella programmazione perché consente di creare una sorta di linguaggio di programmazione esteso, come se fosse una versione personalizzata del linguaggio C++, con cui poter ragionare e programmare a livello superiore.

Un contributo notevole delle funzioni è quello di consentire la risoluzione modulare di un problema computazionale. In pratica ciò corrisponde a

1. dividere il problema di partenza $\mathcal{P}$ in alcuni sottoproblemi $\mathcal{P}_1, \ldots, \mathcal{P}_k$;
2. risolvere ognuno dei sottoproblemi tramite le funzioni $f_1, \ldots, f_k$;
3. creare una funzione f che raccoglie i risultati intermedi e calcola il risultato complessivo.

Questa suddivisione è importante perché facilita in maniera considerevole la scrittura del codice, evitando di risolvere un problema difficile tutto insieme, ma spingendo a ragionare per passi successivi.

Si può anche adottare una tecnica *top-down*, che suggerisce che per risolvere un problema $\mathcal{P}$, è sufficiente prima risolvere in qualche modo (da specificare successivamente) i sottoproblemi $\mathcal{P}_1, \ldots, \mathcal{P}_k$, poi ottenere il risultato finale mettendo insieme i risultati intermedi.

Il passaggio successivo è quello di affrontare nello stesso modo i sottoproblemi $\mathcal{P}_1, \ldots, \mathcal{P}_k$, cercando per ognuno di loro una suddivisione in sottosottoproblemi, e così via, fino ad arrivare a problemi che possono essere risolti in maniera elementare.

Si noti inoltre che, mentre è praticamente impossibile vedere se una parte di un programma è scritta correttamente (ovvero se produce i risultati desiderati), è possibile farlo con una funzione intera. Pertanto se un problema è decomposto in sotto-problemi, ognuno dei quali fa a capo ad una funzione distinta, è possibile controllare la correttezza di ognuna delle parti controllando se la corrispondente funzione lavora in modo corretto.

Una caratteristica positiva della creazione di funzioni legate a sottoproblemi è che può capitare che un sottoproblema faccia parte anche di un nuovo

problema che si affronterà in seguito. In tal caso si può riusare la funzione già definita in precedenza.

Infine l'uso delle funzioni può produrre anche una cospicua riduzione del codice: se le stesse operazioni sono necessarie in più punti di un programma è possibile definire una funzione che svolge tali operazioni e richiamare la funzione tutte le volte che serve. Il vantaggio è quindi che non si devono scrivere più volte le stesse istruzioni.