Appunti sul linguaggio di programmazione C++

Struttura di un programma C++

Ogni programma C++ ha una parte principale, chiamata main, che contiene le istruzioni da eseguire. Le istruzioni devono essere racchiuse all'interno delle parentesi graffe[1].

```
main()
{}
```

In questo paragrafo realizziamo un programma molto semplice il cui scopo è scrivere sul monitor la frase "Hello World!".

Per fare ciò, dobbiamo inserire all'interno delle parentesi graffe la seguente istruzione:

```
cout << "Hello World!";
```

dove:

- `cout` significa "consolle output" e indica il monitor
- la frase `Hello World!`, detta stringa, è contenuta tra virgolette
- l'operatore '<<' è usato per inviare la stringa a `cout`
- il punto e virgola indica la fine dell'istruzione

Inoltre dobbiamo aggiungere, fuori dal main, la seguente direttiva:

```
#include <iostream.h>
```

Il significato di questa direttiva verrà approfondito nei capitoli seguenti. Qui basti dire che essa permette di effettuare scritture sul monitor e letture dalla tastiera.

Il programma completo quindi è:

```
#include <iostream.h>
main()
{
     cout << "Hello World!";
}
```

1 Le parentesi tonde indicano che si tratta di una funzione)

Compilazione

Ogni programma, per poter essere eseguito dal calcolatore, deve essere scritto in un file mediante un editor e poi compilato, cioè tradotto in un formato numerico eseguibile dal processore.

La traduzione viene effettuata da un programma detto compilatore, il quale legge il file ontenente il programma da tradurre (detto programma sorgente) e produce un nuovo file contenente il programma tradotto in formato numerico (detto programma eseguibile).
Ad esempio possiamo scrivere il programma del paragrafo precedente in un file e salvarlo con il nome `helloworld.cpp`; a questo punto è possibile compilarlo creando il file eseguibile `helloworld.exe`.

Alcuni compilatori sono integrati in un ambiente di sviluppo, cioè in un software che comprende un editor mediante il quale è possibile scrivere il file sorgente e salvarlo su disco, per poi ompilarlo e infine eseguirlo.

In appendice 7.1 è mostrato come compilare un programma C++ su alcuni dei più comuni compilatori.

Tipi, variabili ed espressioni

Il linguaggio C++ consente di manipolare dati. I dati possono essere letti dalla tastiera, mostrati sullo schermo, mantenuti nella memoria del calcolatore e utilizzati all'interno di espressioni.

In C++ esistono diversi tipi di dato, ognuno dei quali ha un nome. Ne elenchiamo qui i principali:

- `int` i numeri interi
- `float` i numeri decimali (numeri floating point)
- `char` i caratteri
- `bool` il tipo booleano

Nei prossimi capitoli mostreremo come sia possibile definire dei nuovi tipi di dato.
Per poter memorizzare ed utilizzare un dato è necessario effettuare una dichiarazione, che specifichi il tipo e il nome del dato da utilizzare nel programma. Il nome è formato da uno o più caratteri alfanumerici, ma il primo carattere deve essere una lettera.

Un esempio di dichiarazione è il seguente:

```
int a = 12;
```

In questa dichiarazione il nome `a` indica un dato di tipo intero; il valore iniziale di `a` è 12, ma in seguito tale valore potrà essere modificato. Per questa ragione si dice che `a` è una variabile.

La memoria del calcolatore è organizzata in locazioni, ognuna delle quali può contenere un dato. Le locazioni sono numerate, quindi ogni locazione è identificata da un numero detto indirizzo. Il compilatore associa ogni variabile ad una locazione di memoria. Il contenuto della locazione di

memoria è detto valore della variabile, mentre l'indirizzo della locazione è detto anche indirizzo della variabile.

Analizziamo ora alcuni esempi di dichiarazione di variabile.

- Le due dichiarazioni seguenti sono dichiarazioni senza inizializzazione: alle due variabili non viene assegnato un valore iniziale. Il valore delle variabili, finché non sarà modificato, è quello lasciato dai programmi precedenti nelle rispettive locazioni di memoria.

```
int b;
int c;
```

- La riga seguente dichiara tre variabili di tipo `float`, senza inizializzazione:
  `float lunghezza, larghezza, altezza;`
- La seguente espressione dichiara due variabili di tipo carattere, inizializzando soltanto la prima con il carattere `f`:

```
char k1 = 'f', k2;
```

- Nella dichiarazione seguente `ris` è una variabile booleana, cioè una variabile che può assumere solo il valore `true` o il valore `false` (vero o falso).

```
bool ris = true;
```

Per assegnare un nuovo valore a una variabile si usa l'operatore '=', detto operatore di assegnazione. A sinistra dell'operatore di assegnazione si specifica la variabile da modificare, mentre a destra si indica un'espressione il cui risultato sarà memorizzato nella variabile. Ad esempio, mediante l'istruzione seguente assegniamo un nuovo valore alla variabile `a`. Un eventuale vecchio valore di `a` viene sovrascritto, cioè viene perso.

```
a = 20;
```

Nell'esempio precedente l'espressione a destra dell'uguale è molto semplice (è costituita solamente dalla costante intera 20), mentre nella seconda riga dell'esempio seguente si usa un'espressione più complessa che contiene l'operatore '+' per effettuare la somma tra i due valori delle variabili `a` e `b`. Il risultato della somma, 30, viene passato all'operatore di assegnazione che lo memorizza nella variabile `c`.

```
b = 10;
c = a + b;
```

In C++ è possibile effettuare concatenazioni di assegnazioni:

```
a = b = c;
```

in cui il valore dell'espressione più a destra, in questo caso il valore di `c`, viene propagato verso sinistra. A causa di questa istruzione le tre variabili `a`, `b` e `c` avranno tutte valore uguale a 30. In particolare, il primo operatore '=' da destra assegna a `b` il valore di `c` e poi passa questo valore verso sinistra. Il valore viene quindi utilizzato dall'altro operatore '=', che lo assegna ad `a` e a sua

volta lo passa verso sinistra. A questo punto il valore passato viene eliminato, non essendovi alcun altro operatore che lo utilizzi. In effetti tutte le espressioni C++ passano un valore verso sinistra, anche se non sempre il valore viene utilizzato.

Le quattro operazioni, sia su dati di tipo `int` che di tipo `float`, si eseguono mediante gli operatori '+', '-', '*', '/'. Analizzando l'esempio seguente, in cui la variabile `r` assume il valore 2.25, si può notare l'ordine di precedenza fra gli operatori.

```
float r, s, t;
s = 1.5;
t = 2.0;
r = s*t–s/t;
```

Scriviamo adesso un programma che effettua la somma di due numeri interi.
In particolare, il programma

- dichiara tre variabili di tipo intero `x`, `y`, e `z`
- richiede di inserire due numeri interi da tastiera e li memorizza in `x` e `y`
- somma tali numeri e memorizza il risultato nella variabile `z`
- scrive sul monitor il valore di `z`

In questo programma, oltre agli operatori già analizzati, utilizziamo l'operatore '>>' per inviare un dato dalla tastiera (indicata da `cin`, consolle input) a una variabile. Per richiedere più dati e inviarli a più variabili si possono concatenare più operatori '>>', come mostrato nel programma.

```
#include <iostream.h>
main()
{
     int x, y, z;
     cin >> x >> y;
     z = x + y;
     cout << z;
}
```

Un'altra categoria di operatori è quella degli operatori relazionali, che confrontano valori e restituiscono un risultato di tipo booleano. Ad esempio, l'operatore relazionale '>' (operatore di maggioranza) confronta il valore dei suoi operandi e restituisce `true` se il primo è maggiore del secondo, oppure `false` in caso contrario. Considerando che nell'esempio precedente la variabile `t` vale 2.0 e la variabile `s` vale 1.5, l'espressione

```
t > s;
```

restituisce `true`.
Nell'espressione:

```
ris = t > s;
```

l'operatore '>' confronta il valore di `t` con il valore di `s` e passa il risultato all'operatore '=', che lo assegna alla variabile `ris`. Quindi la variabile `ris` contiene adesso il valore `true`.

Gli operatori relazionali sono i seguenti:

> maggiore
< minore
>= maggiore o uguale
<= minore o uguale
== uguale (verifica di uguaglianza)
!= diverso

Infine passiamo ad analizzare gli operatori logici, usati nell'ambito delle espressioni booleane.

&& and
il risultato è `true` se entrambi gli operandi sono `true`

|| or
il risultato è `true` se almeno uno dei due operandi è `true`

! not
il risultato è la negazione dell'operando, cioè se un operando `ris` è `true`, allora

`!ris` è `false` e viceversa.

Ad esempio, nel seguente frammento di programma si assegna alla variabile `ris` il valore `false`:

```
bool ris1 = true, ris2 = false, ris3 = true;
ris = !(ris1 || ris2) && ris3;
```

Un'ultimo esempio di assegnazione che vale la pena menzionare è l'incremento del valore di una variabile.

```
a = a + 5;
```

Questa istruzione assegna un nuovo valore ad `a`. Il nuovo valore è il risultato della somma del vecchio valore di `a` con il valore 5. Poiché, nel nostro precedente esempio, `a` valeva 30, il nuovo valore è adesso 35.
Allo stesso modo, data una variabile

```
int i;
```

l'istruzione

```
i = i + 1;
```

incrementa di 1 il valore di `i`. Quando l'incremento è unitario si può utilizzare anche l'operatore '++' nel modo seguente:

```
i++;
```

Questa istruzione e quella precedente sono equivalenti.

Si noti che l'operatore '++', utilizzato come sopra, passa verso sinistra il valore che la variabile aveva prima dell'incremento. Allora l'espressione seguente assegna il valore 36 ad `a` e il valore 35 a `b`:

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

Array

Quando è necessario manipolare collezioni di dati omogenei, è opportuno ordinarli in una struttura, detta array. Un array consente di accedere ai dati in esso contenuti specificando un indice che esprime la posizione del dato cercato all'interno della struttura.

Per dichiarare un array è necessario specificare il tipo dei dati in esso contenuti, il nome dell'array e le dimensioni. Ad esempio, mediante la dichiarazione:

```
float v[10];
```

il compilatore costruisce un array di `float`, chiamato `v`, e gli assegna 10 locazioni di memoria consecutive. Il primo elemento dell'array ha indice 0, e si indica con `v[0]`, mentre l'ultimo ha indice 9, e si indica con `v[9]`. Questa dichiarazione è senza inizializzazione, quindi il contenuto degli elementi dell'array `v` non è definito.

Le istruzioni seguenti assegnano i valori 12.0, 5.0 e 60.0 rispettivamente al primo, al secondo e al terzo elemento dell'array `v`:

```
v[0] = 12.0;

v[1] = 5.0;
v[2] = v[0] * v[1];
```

12.0	v[0]
5.0	v[1]
60.0	v[2]
	v[3]
	v[4]
	v[5]
	v[6]
	v[7]
	v[8]
	v[9]

Un vantaggio legato all'uso degli array consiste nel poter utilizzare come indice una
espressione intera. Ad esempio, data una variabile intera `i`, si può scrivere:

```
v[i] = 30.0;
```

In molti casi il valore dell'espressione usata come indice non è definito finché il programma non viene eseguito, quindi la scelta dell'elemento a cui accedere viene fatta solamente a run-time, e non a compile-time. Ad esempio nel programma seguente il compilatore non può stabilire in quale elemento dell'array verrà memorizzato il valore 12.5. Solo durante l'esecuzione del programma l'utente inserirà da tastiera il valore di `i`, decidendo così l'elemento dell'array da utilizzare.

```
main()
{
```

```
    int i;
    float v[10];
    cin >> i;
    v[i] = 12.5;
}
```

Questo costituisce un vantaggio poiché dà una maggiore flessibilità nell'accesso e nella modifica dei dati utilizzati dal programma.

Una dichiarazione di array con inizializzazione si effettua indicando i valori degli elementi tra parentesi graffe separati da virgole. Ad esempio:

```
float u[3] = {1.50, 3.15, 2.92};
```

È possibile dichiarare array multidimensionali, cioè in cui sono necessari più indici per identificare un elemento. Ad esempio, possiamo dichiarare un array bidimensionale di interi di dimensioni 3x5 come segue:

```
int w[3][5];
```

In modo simile all'array monodimensionale, possiamo accedere agli elementi di questo array specificando i due indici:

```
w[0][0] = 100;
w[2][4] = 200;
```

Gli array monodimensionali sono anche chiamati vettori e gli array multidimensionali matrici.

Strutture di controllo

Le strutture di controllo sono delle istruzioni che permettono di eseguire più di una volta, o di non eseguire affatto, una parte del programma. Per fare ciò, queste istruzioni valutano una espressione, anche detta condizione, che restituisce in generale un risultato di tipo booleano.

È possibile condizionare l'esecuzione di una porzione di programma mediante le istruzioni `if-else` e `switch`, mentre è possibile eseguire più volte una parte del programma effettuando un ciclo iterativo con le istruzioni `while`, `for` e `do-while`.

Tratteremo esclusivamente le istruzioni `if-else`, `while` e `for`.

If-else

L'istruzione `if-else` ha il seguente formato:

```
if (espressione)
istruzione;
else
istruzione;
```

L'`if-else` valuta l'espressione e:

- se il risultato è `true`, esegue esclusivamente la prima istruzione (quella del ramo if)
- se il risultato è `false`, esegue esclusivamente la seconda istruzione (quella del ramo else)

Il programma prosegue poi con l'istruzione successiva all'`if-else`.

È importante sottolineare che, quando un ramo `if` o un ramo `else` è composto da più di un'istruzione, è necessario racchiudere tutte le istruzioni che ne fanno parte all'interno di una coppia di parentesi graffe, in modo da formare un blocco.

In un'istruzione `if-else` il ramo `else` può anche essere omesso. In tal caso, se la condizione è vera viene eseguito il blocco `if`, mentre se la condizione è falsa si procede direttamente alla prima istruzione successiva.

Vediamo ora come il programma seguente usa l'`if-else` per valutare quale sia il maggiore tra due numeri letti dalla tastiera.

```
// max2.cpp
#include <iostream.h>
main()
{
     int a,b;
     cin >> a >> b;
     if (a > b)
     cout << "max(a,b)=" << a;
     else
     cout << "max(a,b)=" << b;
}
```

Se, per esempio, i numeri inseriti da tastiera sono rispettivamente 13 e 7, allora la condizione è vera, cioè `a` è maggiore di `b`, e viene eseguita solo l'istruzione del ramo `if` che mostra in output la stringa:

```
max(a,b)=13
```

In questo caso notiamo che non viene eseguita l'istruzione di output posta nel ramo `else`.
Se invece vengono inseriti i numeri 3 e 5, allora viene eseguita l'istruzione del ramo else e non quella del ramo if, e la stringa mostrata in output è:

```
max(a,b)=5
```

Infine, si noti come il programma, nel caso i due numeri siano uguali, esegua il ramo `else`.

La prima riga del programma, quella che inizia con una doppia barra '`//`', è un commento.
I commenti sono delle indicazioni utili per chi legge il programma, ma vengono ignorati dal compilatore. I commenti in C++ sono preceduti dalla doppia barra, e il compilatore considera commento tutto ciò che segue la doppia barra fino alla fine della linea.
Il commento di questo esempio indica il nome del programma, `max2.cpp`.

Il programma successivo fa un uso più compilcato delle strutture `if-else` per distinguere tre casi diversi:

- `a` è maggiore di `b`
- `a` è minore di `b`
- `a e b` sono uguali

Per fare ciò il programma esegue due `if` annidati, cioè due `if,` uno dei quali è contenuto nel ramo if o nel ramo else dell'altro.

```
// max2plus.cpp
#include <iostream.h>
main()
{
     int a,b;
     cout << "inserire a e b:" << endl;
     cin >> a >> b;
     if (a != b)
     {
          cout << "max(" << a << "," << b << ")=";
     if (a > b)
          cout << a;
     else
          cout << b;
     }
     else
     cout << "a e b sono uguali";
     cout << endl;
}
```

Se `a e b` sono diversi si entra nel ramo `if` dell'istruzione `if-else` più esterna. All'interno di esso si trova un'altra istruzione `if-else` che distingue i due casi `a>b` e `a<b`.

Nella tabella seguente sono mostrati gli output corrispondenti ad alcuni input esemplificativi. Le stringhe si formano sul monitor come concatenazione delle diverse istruzioni di output. In particolare se a e b sono diversi viene stampata prima la parte di stringa fino al simbolo '=', poi il valore della variabile a o della variabile b e infine un simbolo di fine linea, `endl`, che non è visibile e serve a far iniziare un eventuale output successivo dall'inizio della riga seguente.

a	b	output
13	7	`max(13,7)=13`
3	5	`max(3,5)=5`
2	2	`a e b sono uguali`

While

Il ciclo `while` è formato da una espressione e da un'istruzione:
`while (espressione)`

```
istruzione;
```

In ogni iterazione del ciclo viene valutata l'espressione e, se il risultato è `true`, viene eseguita l'istruzione. Fatto ciò si ritorna a valutare l'espressione e così via. Se, ad un certo punto, la valutazione dell'espressione dà risultato `false`, si esce dal ciclo, cioè si esegue la prima istruzione successiva del programma. Notare che l'istruzione di un ciclo `while` potrebbe non essere mai eseguita se l'espressione dà risultato `false` già alla prima valutazione.

Il corpo del ciclo può essere composto da più istruzioni. In questo caso è necessario racchiuderle tra parentesi graffe a formare un blocco.

Per illustrare il ciclo while realizziamo un programma che calcola il massimo comun divisore (M.C.D.) tra due numeri. Per fare ciò, dati due numeri interi positivi m ed n diversi fra loro, sottraiamo il più piccolo al più grande. Ripetiamo la sottrazione finché i due numeri diventano uguali. Il numero ottenuto è il M.C.D.

Ad esempio, dati i numeri 48 e 18 effettuiamo le sottrazioni seguenti.

m	n	
48	18	sottraiamo n ad m
30	18	sottraiamo n ad m
12	18	sottraiamo m ad n
12	6	sottraiamo n ad m
6	6	il Massimo Comun Divisore è 6

Il programma utilizza un ciclo iterativo che termina quando i due numeri diventano uguali.
Il numero di iterazioni, ovviamente, non è noto a compile-time, poiché dipende dai numeri m ed n inseriti da tastiera a runtime.

```
// mcd_i.cpp
#include <iostream.h>
main()
{
     int m,n;
     cin >> m >> n;
     while (m != n)
     {
          if (m > n)
               m = m – n;
          else
               n = n – m;
     }
     cout << “M.C.D. = “ << m << endl;
}
```

Mostriamo adesso un altro esempio di uso del while. Questo programma ha lo scopo di calcolare il fattoriale di un numero n letto in input, cioè il prodotto `1*2*...*n`.

```
//fatt.cpp
#include <iostream.h>
```

```
main()
{
     int i = 1, f = 1, n;
     cout << “Fattoriale iterativo. Inserire n: “;
     cin >> n;
     while (i <= n)
          f = f * i++;
     cout << “fattoriale = “ << f << endl;
}
```

Il ciclo compie `n` iterazioni, e la variabile `i` varia da 1 a `n`. Il corpo del ciclo è composto da un’unica istruzione che assegna ad `f` il risultato del prodotto di `f * i`. Notare che `i` viene incrementata ma il valore passato all’operatore ‘`*`’ è quello precedente all’incremento.

For

Il ciclo `for` ha la seguente sintassi:

```
for (espr1; espr2; espr3)
istruzione;
```

dove:

- `espr1` è un’espressione che viene eseguita una volta per tutte prima di iniziare il ciclo
- `espr2` è l’espressione che costituisce la condizione del ciclo e viene valutata all’inizio di ogni iterazione; si esegue l’iterazione se il risultato di `espr2` è `true`, altrimenti si esce dal ciclo

- `espr3` è un’espressione che viene valutata alla fine di ogni iterazione, dopo aver eseguito l’istruzione

Il ciclo `for` può sempre essere sostituito da un ciclo `while` come segue:

`for (espr1; espr2; espr3)`
`istruzione;`
	`espr1;`
	`while (espr2)`
	`{`
	`  istruzione;`
	`  espr3;`
	`}`

Mostriamo il ciclo for mediante il programma seguente in cui si effettua la somma di dieci numeri interi letti da tastiera.

```
// somma.cpp
#include <iostream.h>
main()
{
     int i,n,somma = 0;
```

```
     cout << “Inserire 10 numeri da sommare:” << endl;
     for (i = 0; i < 10; i++)
     {
          cin >> n;
          somma = somma + n;
     }
     cout << “somma = “ << somma << endl;
}
```

Il ciclo viene eseguito 10 volte; alla prima iterazione la variabile `i` vale 0 (è stata posta a zero mediante la `espr1` del ciclo `for`); all'ultima iterazione vale 9 (è stata incrementata alla fine di ogni iterazione mediante `espr3`); alla fine dell'ultima iterazione la variabile `i` viene ancora incrementata di uno, quindi passa al valore 10, e quindi l'`espr2` restituisce `false` e il ciclo termina.

Nel corpo del ciclo viene letto uno dei 10 numeri e il suo valore viene aggiunto al valore della variabile `somma` che viene usata come accumulatore.

Mostriamo un altro esempio di uso del ciclo for per calcolare il massimo tra otto numeri interi.

```
// max8.cpp
#include <iostream.h>
main()
{
     cout << “Massimo tra 8 numeri interi” << endl;
     int n,max;
     cout << “Inserire gli 8 numeri” << endl;
     cin >> max;
     for (int i = 1; i < 8; i++)
     {
          cin >> n;
          if (n > max)
               max = n;
     }
     cout << “max=” << max << endl;
}
```

Questo programma funziona mantenendo un massimo corrente, cioè un massimo valido finché non viene inserito un numero maggiore. Il primo degli otto numeri è per definizione il massimo corrente all'inizio (ancora non sono stati inseriti altri numeri) e quindi viene memorizzato direttamente nella variabile `max`. I numeri successivi vengono inseriti uno alla volta nelle iterazioni[2] del ciclo e, quando uno di essi è maggiore del massimo corrente, viene aggiornato il massimo corrente. La variabile `max` alla fine del ciclo contiene il massimo degli otto numeri letti.

Notare che l'`espr1` contiene una dichiarazione di variabile con inizializzazione. La variabile dichiarata esisterà solamente fino alla fine del ciclo.

Mostriamo un altro esempio di uso di cicli for per effettuare la trasposizione di una matrice. Data una matrice di 9 elementi organizzati in 3 righe per 3 colonne, copiamo i suoi elementi in una nuova matrice scambiando le righe con le colonne. Al termine mostriamo in output le due matrici.

```
//trasp.cpp
#include <iostream.h>
main()
{
     int m[3][3], n[3][3];
     int i, j;
     // inizializzazione della matrice m
     for (i = 0; i < 3; i++)
          for (j = 0; j < 3; j++)
          cin >> m[i][j];
     // trasposizione di m in n
     for (i = 0; i < 3; i++)
          for (j = 0; j < 3; j++)
               n[j][i] = m[i][j];
     // output
     cout << “Matrice m” << endl;
     for (i = 0; i < 3; i++)
     {
          for (j = 0; j < 3; j++)
               cout << m[i][j] << '\t';
          cout << endl;
     }
     cout << endl << “Matrice n” << endl;
     for (i = 0; i < 3; i++)
     {
          for (j = 0; j < 3; j++)
               cout << n[i][j] << '\t';
          cout << endl;
     }
     cout << endl;
}
```

Un output di esempio del programma è il seguente:

```
Matrice m
10   20   30
40   50   60
70   80   90
Matrice n
10   40   70
20   50   80
30   60   90
```

Notare l’uso dell’`endl` e del carattere arresto di tabulazione ‘`\t`’ per incolonnare gli elementi delle matrici.

Altri due esempi

Due esempi ulteriori a proposito delle strutture di controllo del flusso del programma sono i seguenti.

Nel primo si effettua una ricerca esaustiva di un elemento in un array di 10 numeri interi. Notare l'uso dell'operatore logico '&&' (and, vedi paragrafo 1.3) .

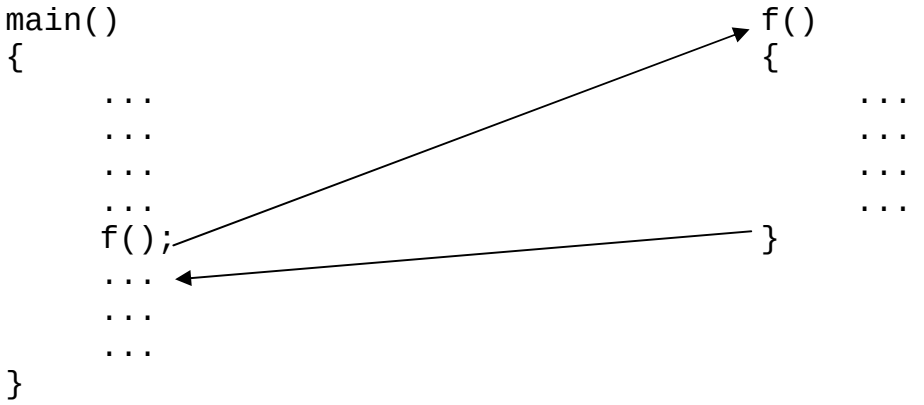
```
// ric.cpp
#include <iostream.h>
main()
{
     float v[10];
     // inizializzazione array
     for (int i = 0; i < 10; i++)
     cin >> v[i];
     //ricerca elemento
     cout << "Inserire il numero da cercare: ";
     cin >> f;
     cout << endl;
     int j = 0;
     while (j < 10 && v[j] != f)
          j++;
     if (j < 10)
          cout << "Trovato in posizione " << j << endl;
     else
          cout << "Non trovato." << endl;
}
```

Il secondo esempio è il gioco "Hi-Lo Game" in cui l'utente pensa un numero tra 0 e 1000 e il programma prova a indovinarlo mediante tentativi successivi. Il programma ad ogni tentativo calcola un numero e lo mostra in output chiedendo all'utente se è troppo alto, troppo basso o corretto. L'utente inserisce il numero 1 se il tentativo è superiore al numero pensato, -1 se è inferiore, e 0 quando il numero pensato viene indovinato.

```
// hilo.cpp
// hi-lo game
#include <iostream.h>
main()
{
     int max = 1000, min = 0, n, r = 1, tentativi = 0;
     while (r != 0)
     {
          n = (max + min) / 2;
          cout << n << endl;
          tentativi++;
          cin >> r;
          if (r == 1)
               max = n;
          if (r == -1)
               min = n;
     }
     cout << tentativi << " tentativi." << endl;
}
```

Modularizzazione mediante funzioni

Una funzione in C++ è una porzione di programma che effettua un determinato compito e che può essere chiamata ed eseguita una o più volte durante lo svolgimento del programma principale.



Un compito tipico, molto semplice, di una funzione è quello di calcolare un valore a partire da un dato di ingresso.

Ad esempio è possibile scrivere un programma che, mediante una funzione, calcola il cubo di un numero intero.

La funzione viene chiamata dall'interno del main, e le viene passato dal main un parametro, il numero di cui calcolare il cubo. L'esecuzione del main viene interrotta e viene eseguita la sequenza di istruzioni che compongono la funzione. Al termine di queste istruzioni il risultato (il cubo del numero dato) viene restituito al main che può usarlo nelle sue istruzioni successive.

In questo caso la funzione è server e il main è client. L'interfaccia della funzione è costituita dal suo nome, dal tipo degli argomenti o parametri, (i dati di ingresso e uscita, in questo caso due numeri interi) e dalle modalità con cui gli argomenti devono esserle passati.

I criteri esposti nel paragrafo precedente, applicati alle funzioni C++, suggeriscono che la funzione esegua un solo compito chiaro e preciso (per mantenere alta la coesione), che la sua interfaccia epliciti chiaramente i parametri di ingresso e di ritorno (vedi capitolo 3) e che la funzione, al fine minimizzare l'accoppiamento, usi solo i parametri definiti nell'interfaccia per comunicare con il resto del programma (e non, ad esempio, variabili globali accessibili anche dal main, vedi paragrafo 3.4.1). Infine l'information hiding è garantita a patto di usare all'interno della funzione variabili locali non accessibili dall'esterno (vedi paragrafo 3.4.1).

Funzioni

Funzioni in C++

Una funzione è una porzione di programma che effettua un determinato compito e che può essere chiamata ed eseguita una o più volte durante lo svolgimento del programma principale. Un compito tipico, molto semplice, di una funzione è quello di calcolare un valore a partire da un dato di ingresso. Ad esempio è possibile scrivere un programma che, mediante una funzione, calcola il cubo di un numero intero:

```
// cubo.cpp
#include <iostream.h>
int cubo(int i)
{
     int c = i * i * i;
     return c;
}

main()
{
     int a,b;
     cin >> a;
     b = cubo(a);
     cout << b << endl;
}
```

La definizione della funzione comprende l'intestazione (la prima riga) e il corpo (le istruzioni racchiuse tra parentesi graffe). L'intestazione specifica tre informazioni:

- il tipo del valore restituito dalla funzione (anche detto tipo di ritorno): in questo esempio, `int`
- il nome della funzione: in questo esempio, `cubo`
- i parametri formali passati alla funzione, cioè i dati di ingresso sui quali la funzione lavora, racchiusi tra parentesi; in questo esempio l'unico parametro formale è un intero, `int i`

Nel `main()` la funzione viene chiamata specificandone il nome e mettendo tra parentesi il parametro attuale, cioè il dato di ingresso da passarle. Il valore del parametro attuale viene copiato nel parametro formale prima di iniziare ad eseguire il corpo della funzione (questo modo di passare i parametri si chiama passaggio per valore, vedi 3.2).

La funzione è definita nel programma prima del `main()`, ma il programma inizia sempre dal `main()`. Infatti il main è una funzione speciale, in quanto è quella che viene chiamata dal sistema operativo per lanciare il programma. D'altra parte, il compilatore, per poter compilare le istruzioni del main nelle quali si fa riferimento a una funzione, deve essere a conoscenza dell'esistenza della funzione, cioè deve avere già incontrato la dichiarazione.

Il `main()` di questo programma legge da tastiera un numero intero, lo memorizza in `a` e chiama la funzione `cubo()` specificando `a` come parametro attuale. A questo punto l'esecuziuone del `main()` viene temporaneamente interrotta, il valore del parametro attuale `a` viene copiato nel parametro formale `i`, e si passa all'esecuzione del corpo della funzione. La funzione calcola il cubo moltiplicando `i` per se stessa due volte e memorizza il risultato nella variabile `c`. Poi, mediante l'istruzione `return`, il risultato viene restituito al chiamante, cioè al `main()`. L'istruzione return causa anche la terminazione della funzione e quindi si riprende l'esecuzione del `main()` che, mediante l'operatore '=', assegna a `b` il valore restituito dalla funzione e poi lo stampa. A questo punto il programma termina.

Notare che la variabile `i` e la variabile `c` vengono create quando la funzione viene chiamata e vengono eliminate al termine della funzione. Si dice che il loro ciclo di vita termina quando termina la funzione (vedi paragrafo 3.4.2).

Analizziamo adesso un altro esempio: vogliamo scrivere un programma che legga da tastiera una parola e che converta in maiuscolo tutti i caratteri alfabetici minuscoli contenuti in essa. Per fare ciò scandiamo la parola e passiamo un carattere alla volta ad una funzione che, se il carattere è minuscolo, ci restituisce il corrispondente carattere maiuscolo, altrimenti ci restituisce il carattere stesso.

Nel corpo della funzione sfruttiamo il fatto che i caratteri sono codificati secondo il codice ASCII. Nella tabella ASCII i caratteri maiuscoli e i caratteri minuscoli hanno codici consecutivi tra loro e sono ordinati alfabeticamente. Pertanto la distanza tra un carattere minuscolo e il corrispondente carattere maiuscolo è costante. Ad esempio il carattere 'A' ha codice ASCII 65, e il carattere 'a' ha codice 97, quindi la loro distanza è 32. Anche tra 'B' (ASCII 66) e 'b' (ASCII 98) c'è distanza 32, e così via per tutte le lettere fino a 'Z' e 'z'.
Quindi per ottenere il codice di un carattere minuscolo è sufficiente sottrarre la distanza 32 ('a'-'A') al codice del corrispondente carattere minuscolo.

```
// upcase.cpp
#include <iostream.h>
char up(char c)
{
     if (c >= 'a' && c <= 'z')
     c = c – ('a' - 'A');
     return c;
}

main()
{
     int i = 0;
     char s[16];
     cout << "Conversione in maiuscole." << endl;
     cout << "Inserire parola (max 15 caratteri): ";
     cin >> s;
     while (s[i] != '\0')
     {
          s[i] = up(s[i]);
          i++;
```

```
     }
     cout << s << endl;
}
```

In questo esempio la funzione `up()` viene chiamata tante volte quanti sono i caratteri della parola `s`. Il parametro attuale è `s[i]`, con `i` che varia di volta in volta, mentre il parametro formale è `c`. Ogni volta che la funzione inizia viene creata una nuova variabile `c` che viene distrutta quando la funzione termina.

L'esempio seguente è un programma che ribalta un array monodimensionale, cioè scambia il posto degli elementi in modo che il primo venga messo in ultima posizione, il secondo in penultima e così via. Questo programma utilizza tre funzioni diverse per realizzare tre diversi compiti: l'inizializzazione dell'array (funzione `init()`), il ribaltamento dell'array (funzione `revvect()`) e la stampa dell'array (funzione `printout()`).

In questo esempio ci sono due novità:

- l'uso di parametri di tipo puntatore (usati per passare l'indirizzo del vettore); questo argomento verrà approfondito in seguito;
- l'uso della parola chiave `void` per indicare che una funzione non restituisce alcun valore (per esempio, dato che l'unico scopo della funzione `print()` è quello di stampare su `cout` gli elementi dell'array, la funzione non restituisce alcun valore al `main()`e quindi il suo tipo di ritorno è dichiarato `void`);

```
// revvect.cpp
#include <iostream.h>
void print(int* v, int n)
{
     cout << endl;
     for (int i = 0; i < n; i++)
          cout << "v[" << i << "]=" << v[i] << endl;
}

int* init(int n)
{
     int* v = new int[n];
     for (int i = 0; i < n; i++)
          v[i] = (n – i) * 10;
     return v;
}

void reverse(int* v, int n)
{
     int x;
     for (int i = 0; i < n/2; i++)
     {
          x = v[i];
          v[i] = v[n-1-i];
          v[n-1-i] = x;
     }
```

```
}

main()
{
     cout << “Ribaltamento array monodimensionale.” << endl;
     cout << “Inserire la dimensione dell’array: “;
     int dim;
     cin >> dim;
     int* arr = init(dim);
     print(arr,dim);
     reverse(arr,dim);
     print(arr,dim);
}
```

Passaggio di parametri per valore e per riferimento

Negli esempi del paragrafo precedente abbiamo utilizzato sempre il passaggio di parametri per valore. Questo tipo di passaggio di parametri è chiamato così poiché durante la chiamata della funzione viene copiato il valore del parametro attuale nel parametro formale. Quando la funzione inizia, parametro attuale e parametro formale hanno lo stesso valore: da questo momento in poi, il parametro formale (che è una variabile a se stante)
può essere modificato senza che questa modifica si ripercuota sul parametro attuale. Non ci sono variabili in comune tra il `main()` e la funzione, il parametro formale è una variabile a se stante. Notare che se la funzione modificasse il parametro formale, questa modifica non potrebbe influire sul valore del parametro attuale.

L’altro tipo di passaggio di parametri del C++ è il passaggio per riferimento, in cui non viene effettuata alcuna copia. Infatti in questo caso il parametro formale è un alias, cioè un nome alternativo, del parametro attuale.

Nel passaggio di parametri per riferimento ogni modifica apportata al parametro formale è a tutti gli effetti una modifica al parametro attuale, in quanto si tratta della stessa variabile.

Per specificare un passaggio di parametri per riferimento si usa il simbolo ‘&’, ad esempio `int& i`.

L’esempio seguente mostra il diverso comportamento di due funzioni che usano i due tipi di passaggio di parametri. Per evidenziare il fatto che la prima usa il passaggio per valore e la seconda il passaggio per riferimento, abbiamo chiamato le funzioni “pv” e “pr”.

```
#include <iostream.h>
void pv(int x) // passaggio per valore: COPIA
{
     x = x + 1;
     cout << x << endl; // x == 11
}
v
oid pr(int& y) // passaggio per riferimento: ALIAS
{
     y = y + 1;
```

```
     cout << y << endl; // y == 11
}

main()
{
     int i = 10;
     pv(i);
     cout << i << endl; // i == 10
     pr(i);
     cout << i << endl; // i == 11
}
```

All'inizio del `main()` `i` vale 10. Nella chiamata a `pv()` il valore 10 viene copiato in `x`, `x` viene incrementata a 11 e viene stampato 11. Si ritorna al `main()` e si stampa `i` che vale ancora 10. Poi viene fatta la chiamata a `pr()`. Qui `y` vale 10, ma non è una variabile indipendente, bensì un alias di `i`. Allora se `y` viene incrementata di 1, anche `i` risulta incrementata di 1. Perciò all'interno della funzione `pr()` viene stampato 11, poi si torna al `main()` e si stampa 11, il novo valore di `i`.

Visibilità e ciclo di vita delle variabili

Visibilità

La visibilità o scope di una variabile indica la parte del programma dove la variabile è visibile, cioè la parte di programma che può accedere alla variabile per leggerla o per modificarla.

La visibilità di una variabile in C++ è limitata all'interno del blocco (coppia di parentesi graffe) in cui è stata dichiarata.

Ad esempio, una variabile dichiarata all'interno del `main()` è visibile solo nel `main()` mentre non è visibile dall'interno delle funzioni:

```
#include <iostream.h>
void f()
{
     cout << i << endl; // ERRORE, i non è visibile
}
main()
{
     int i = 100; // i è visibile da qui ...
     f();
} // ... a qui
```

Questo implica che in una funzione può essere dichiarata una variabile (o un parametro formale) che ha lo stesso nome di una variabile del main(). Ovviamente si tratterà di due variabili diverse e ognuna di esse sarà visibile nel blocco in cui è stata dichiarata:

```
#include <iostream.h>
void f(int i)
```

```
{
     int j;
     j = i * 2; // sono le i e j dichiarate dentro f()
}
main()
{
     int i,j;
     i = 12; // sono le i e j dichiarate
     j = 34; // nel main()
     f(i); // è la i del main()
}
```

Attenzione: il parametro attuale e il parametro formale usati nell'esempio sono sempre due variabili diverse anche se hanno lo stesso nome.

In C++ è possibile dichiarare variabili al di fuori di ogni funzione. Queste variabili sono globali, cioè visibili da ogni funzione del programma, incluso ovviamente il `main()`.

```
#include <iostream.h>
int a = 0;
void f(int i)
{
     a = a + i;
}
main()
{
     cout << a << endl; // 0
     for (int i = 0; i < 10; i++)
     f(i);
     cout << a << endl; // 45
}
```

La possibilità di dichiarare variabili globali deve essere usata con cautela poiché va contro la modularizzazione e nasconde eventuali errori.

Ciclo di vita

Il ciclo di vita di una variabile è il periodo compreso tra il momento in cui la variabile viene creata (cioè in cui le viene associata una locazione di memoria) e il momento in cui viene eliminata (e la locazione di memoria viene resa libera e riutilizzabile).
In generale una variabile ha un ciclo di vita che inizia con la dichiarazione e termina con la parentesi graffa di chiusura del blocco in cui è dichiarata. Per esempio, una variabile dichiarata all'inizio di una funzione termina alla fine della funzione stessa, una variabile dichiarata all'interno di un `if` o di un `while` termina alla fine del corpo del `if` o del `while`:

```
#include <iostream.h>
main()
{
     int i = 5,j;
     if (i < 10)
```

```
     {
          int k;
          k = i * 2;
          j = k / 3;
     }
     cout << j << endl; // 3
     cout << k << endl; // ERRORE, k non è più viva
}
```

Nell'esempio seguente si mostra che una variabile locale di una funzione termina il suo ciclo di vita con la fine della funzione stessa. Se la funzione viene richiamata nel seguito del programma, la variabile locale verrà ri-dichiarata e ri-allocata ex-novo.
Nel `main()` dell'esempio si chiama due volte una funzione `f()`, nella quale esiste una variabile locale `x`.

Nella prima chiamata (o attivazione) della funzione, viene creata la `x`, viene stampata, poi viene incrementata di 10 e poi viene stampata di nuovo.
Dopo la prima chiamata di `f()` si effettua una chiamata a `g()`: è probabile che una delle variabili locali di `g()` occupi la locazione che prima era della `x`, modificandone il valore.
Nella seconda chiamata di `f()`, viene creata una nuova `x`, che molto probabilmente avrà un valore diverso da quello che aveva alla fine della chiamata precedente. Questo può dipendere da vari fattori:

- la nuova `x` potrebbe essere allocata in una locazione diversa da quella della vecchia `x`
- la nuova `x` potrebbe essere allocata nella locazione della vecchia `x`, ma tra una chiamata e l'altra della funzione `f()` la locazione potrebbe essere stata utilizzata da una variabile di un'altra funzione, e quindi modificata
- il compilatore potrebbe effettuare un'inizializzazione a 0 delle variabili `int` locali alle funzioni: in questo caso la `x` verrebbe comunque posta a zero all'inizio della nuova chiamata, perdendo qualsiasi valore precedente

```
#include <iostream.h>
void f()
{
     int x;
     cout << "f(): all'inizio: " << x << endl;
     x = x + 10;
     cout << "f(): alla fine:" << x << endl;
}
void g(int a)
{
     int b;
     b = a;
}
main()
{
     f();
     g(100);
     f();
}
```

Una variabile all'interno di una funzione può essere dichiarata `static` per indicare che non deve essere distrutta alla fine della attivazione della funzione ma deve essere lasciata viva e riutilizzabile dalle attivazioni successive:

```
#include <iostream.h>
int accumula(int i)
{
     static int a = 0;
     a = a + i;
     return a;
}
main()
{
     int n;
     for (int i = 0; i < 10; i++)
     {
          cin >> n;
     accumula(n);
     }
     cout << "Totale = " << accumula(0) << endl;
}
```

In questo esempio la funzione ha lo scopo di accumulare nella variabile statica `a` la somma dei valori che le vengono passati nelle dieci chiamate.

Alla prima chiamata la variabile `a` viene inizializzata con il valore 0. Ad ogni chiamata la funzione somma il valore di `i` alla variabile `a`. Al termine viene stampato il valore finale che è la somma dei 10 valori inseriti. La variabile statica viene distrutta soltanto quando il programma termina.

La variabile statica dichiarata all'interno di una funzione ha ciclo di vita uguale a quello di una variabile globale (dall'inizio al termine del programma) ma non è visibile al di fuori della funzione, quindi è migliore ai fini della modularizzazione.

Il ciclo di vita è un concetto distinto dalla visibilità, infatti:

- il fatto che una variabile sia viva in un dato istante non implica che sia visibile dalla porzione di programma che è in esecuzione in quell'istante (si pensi a una variabile globale e una variabile dichiarata in una funzione aventi lo stesso nome: all'interno della funzione sono vive entrambe, ma solo quella locale è visibile; un'altro esempio è l'uso di variabili `static`)
- una variabile locale di una funzione può esistere in più istanze, perché la funzione può essere richiamata più volte (vedi paragrafo 3.5): durante l'esecuzione della funzione tutte le istanze sono vive, ma una sola è visibile.