Strutture di Controllo

- In generale, un programma non è semplicemente una sequenza di istruzioni.
- Durante la sua esecuzione esso può ripetere più volte uno stesso gruppo di istruzioni (detto blocco di istruzioni).
- Un blocco di istruzioni è un gruppo di istruzioni separate da punti e virgola (;) e racchiuse tra parentesi graffe: { e } .
- Il C++ fornisce delle strutture di controllo per permettere la ripetizione di blocchi:
  - Strutture condizionali: **if** ed **else**
  - Istruzioni iterative o cicli
  - Biforcazioni di controllo e salti
  - L'istruzione di scelta: **switch**

50

Strutture Condizionali: if ed else

- La struttura condizionale **if** viene usata per eseguire una istruzione o un blocco di istruzioni soltanto se si verifica una determinata condizione. La sua forma è:

```
if (Condizione) Istruzione
```

  dove **Condizione** è una espressione di tipo **bool** e **Istruzione** indica una singola istruzione o un blocco di istruzioni.
- Se la valutazione della **Condizione** ritorna il valore **true**, l'istruzione (o gruppo di istruzioni) **Istruzione** viene eseguita
- Se la valutazione della **Condizione** ritorna il valore **false**, l'istruzione (o gruppo di istruzioni) **Istruzione** viene ignorata e l'esecuzione del programma continua con le istruzioni che seguono immediatamente la struttura condizionale.

Ad esempio il seguenti codici stampano *x e' 100* soltanto se il valore della variabile **x** è proprio 100:

```
if (x == 100)
       cout << "x e' 100";
```

```
if (x == 100)
 {
  cout << "x e' ";
  cout << x;
 }
```

51

Strutture Condizionali: if ed else

- Si può anche usare la parola chiave **else** per specificare una diversa istruzione o blocco di istruzioni da eseguire nel caso in cui la condizione sia falsa. La forma della struttura condizionale è in questo caso la seguente:

```
if (Condizione) Istruzione1 else Istruzione2
```

Ad esempio:

```
if (x == 100)
    cout << "x e' 100";
else
        cout << "x non e' 100";
```

il seguente codice stampa x e' 100 se x vale proprio 100, altrimenti stampa x non e' 100

```
if (x > 0)
    cout << "x e' positivo";
else if (x < 0)
        cout << "x e' negativo";
     else
        cout << "x e' 0";
```

il seguente codice verifica se il valore della variabile x è negativo, positivo o nessuno dei due (ossia è zero), e stampa in output il risultato della verifica

Notare che la struttura if + else si può concatenare (anche più volte)

52

Istruzioni iterative o cicli

- I cicli (**while**, **do-while**, **for**) hanno lo scopo di ripetere una istruzione o gruppo di istruzioni un certo numero di volte oppure fino a che una certa condizione è soddisfatta.
- **while:** il suo formato è:

```
while (Condizione) Istruzione
```

- Il suo effetto è semplicemente ripetere **Istruzione** fintanto che **Espressione** ha il valore true

```
// conto alla rovescia usando while
#include <iostream.h>
int main ()
{
  int n;
  cout << "Dammi il valore da cui partire > ";
  cin >> n;
  while (n>0) {
    cout << n << ", ";
    --n;
  }
  cout << "FUOCO!";
  return 0;
}
```

```
Dammi il valore da cui partire > 8
8, 7, 6, 5, 4, 3, 2, 1, FUOCO!
```

53

Istruzioni iterative o cicli

- I cicli (**while**, **do-while**, **for**) hanno lo scopo di ripetere una istruzione o gruppo di istruzioni un certo numero di volte oppure fino a che una certa condizione è soddisfatta.
- **while:** il suo formato è:

  while (Condizione) Istruzione
- Il suo effetto è semplicemente ripetere **Istruzione** fintanto che **Espressione** ha il valore true

```
// conto alla rovescia usando while
#include <iostream.h>
int main ()
{
  int n;
  cout << "Dammi il valore da cui partire > ";
  cin >> n;
  while (n>0) {
    cout << n << ", ";
    --n;
  }
  cout << "FUOCO!";
  return 0;
}
```

```
Dammi il valore da cui partire > 8
8, 7, 6, 5, 4, 3, 2, 1, FUOCO!
```

ATTENZIONE: occorre SEMPRE assicurarsi che il ciclo ad un certo punto termini !

54

Istruzioni iterative o cicli

- **do-while:** il suo formato è:

  do Istruzione while (Condizione);
- Funziona esattamente come il ciclo while tranne il fatto che **Condizione** viene controllata dopo l'esecuzione di **Istruzione**, che quindi viene eseguita sempre almeno una volta, anche se la condizione **Condizione** non è vera.
- Ad esempio, il seguente programma ripete ogni numero che l'utente inserisce finché non viene inserito 0.

```
// ripetitore di numeri
#include <iostream.h>
int main ()
{
  unsigned long n;
  do {
    cout << "Dammi un numero (0 per finire): ";
    cin >> n;
    cout << "Mi hai dato: " << n << "\n";
  } while (n != 0);
  return 0;}
```

```
Dammi un numero (0 per finire): 12345
Mi hai dato: 12345
Dammi un numero (0 per finire): 160277
Mi hai dato: 160277
Dammi un numero (0 per finire): 0
Mi hai dato: 0
```

Il **do-while** si usa quando
la condizione che deve terminare il ciclo viene determinata all'interno del ciclo stesso.

55

Il ciclo for

- Il suo formato è:

```
for ( Inizializzazione; Condizione ; Incremento) Istruzione
```

- Il suo scopo è quello di ripetere **Istruzione** finché la condizione **Condizione** rimane vera.
- Il ciclo **for** permette inoltre di indicare anche:
  - una istruzione di inizializzazione **Inizializzazione**
  - una istruzione di incremento **Incremento**.
- Il ciclo **for** è quindi particolarmente adatto ad eseguire delle ripetizioni usando un contatore.

56

Il ciclo for

Il ciclo **for** opera nel seguente modo:

1. Viene eseguita l'istruzione **Inizializzazione** . Generalmente essa è un assegnamento di un valore iniziale al contatore. Essa viene eseguita una sola volta.
2. Viene controllata la condizione **Condizione** , se essa è true il ciclo continua (al punto 3), altrimenti il ciclo termina.
3. Viene eseguita l'istruzione (o blocco di istruzioni) **Istruzione**.
4. Infine, viene eseguita l'istruzione **Incremento** e si ritorna al punto 2.

57

Il ciclo for

- Esempio di conto alla rovescia

```
//conto alla rovescia con for
#include <iostream.h>
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FUOCO!";
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FUOCO!
```

- I campi **Inizializzazione** e **Incremento** sono opzionali.
- Essi si possono quindi omettere ma non si può omettere il punto e virgola che li separa dalla **Condizione** .
- Possiamo quindi scrivere **for (;n>0;)** se non vogliami indicare né **Inizializzazione** né **Incremento** oppure **for (;n>0;n--)** se vogliamo includere il campo **Incremento** ma non **Inizializzazione** .

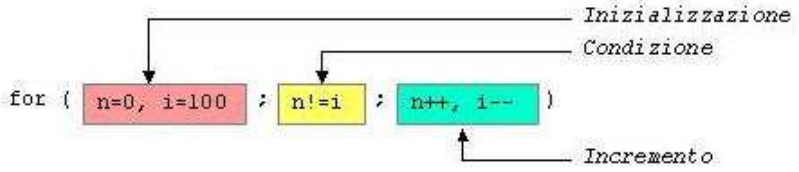
58

Il ciclo for

- Si può inoltre usare l'operatore virgola (,) per specificare più di una azione in uno qualsiasi dei campi di **for**.
- L'operatore virgola è un separatore di istruzioni che serve a separare più istruzioni dove ne è richiesta una soltanto. Ad esempio:

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
{
   // qualcosa...
}
```

- Questo ciclo viene ripetuto 50 volte sempre che né **n** né **i** vengano modificate all'interno del ciclo:



59

Biforcazioni di controllo e salti

- L'istruzione **break**: si può usare per uscire da un ciclo infinito oppure per forzare la terminazione di un ciclo in presenza di qualche anomalia.
- Ad esempio:

```
// esempio di interruzione di un ciclo
#include <iostream.h>
int main ()
{  int n;
   for (n=10; n>0; n--) {
    cout << n << ", ";
    if (n==3)
    {
      cout << "conto alla rovescia interrotto!";
      break;
    }
   }
   return 0;
}
```

```
10, 9, 8, 7, 6, 5,  4, 3, conto alla rovescia interrotto!
```

60

Biforcazioni di controllo e salti

- L'istruzione **continue**: termina immediatamente l'iterazione che si stà eseguendo ma non fa uscire dal ciclo.
- Essa salta tutto il resto del blocco di istruzioni che si sta eseguendo andando direttamente alla fine del blocco e passando quindi all'iterazione successiva.
- Ad esempio:

```
// esempio di interruzione di un ciclo
#include <iostream.h>
int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "FUOCO!";
  return 0;
}
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, FUOCO!
```

61

Biforcazioni di controllo e salti

- L'istruzione **goto** permette di saltare direttamente ad un altro punto del programma. Deve essere usata soltanto nei casi di effettiva necessità (che sono estremamente rari) perché essa sovverte la struttura del programma.
- Il punto di arrivo del salto è indicato dall'etichetta che compare come argomento dell'istruzione goto . L'etichetta deve essere premessa all'istruzione a cui saltare e separata da essa dal carattere due punti (:).

```
//  esempio di ciclo con goto
#include <iostream.h>
int main ()
{
  int n=10;
  loop:
  cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "FUOCO!";
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FUOCO!
```

Consiglio: non utilizzate goto !!!

62

Biforcazioni di controllo e salti

- La funzione **exit**: è definita nella libreria standard (**stdlib.h**).
- Essa termina l'esecuzione del programma ritornando un codice di terminazione intero. La sua forma è:

```
exit(cod) ;
```

- dove **cod** è un valore intero che viene ritornato al sistema operativo.
- Per convenzione il valore 0 di **cod** significa che il programma è terminato correttamente mentre un valore diverso da zero indica che è stato riscontrato un errore di esecuzione.

63

L'istruzione di scelta: switch

- Lo scopo di **switch** è quello di confrontare il valore di una espressione con un certo numero di valori costanti ed eseguire il blocco di istruzioni associato al valore dell'espressione. La sua forma è:

```
for switch (Espressione) {
     case Costante1:
            blocco di istruzioni 1
            break;
     case Costante2:
            blocco di istruzioni 2
            break;
     .
     .
     .
            default:
            blocco di istruzioni di default
}
```

Attenzione: se si dimentica di inserire l'istruzione break alla fine dei blocchi di istruzioni l'esecuzione continua controllando anche le successive costanti ed infine eseguendo il blocco di istruzioni di default

scelto se l'espressione non risulta uguale a nessuna delle costanti elencate

64

L'istruzione di scelta: switch

- Codici equivalenti:

```
switch (x) {
  case 1:
    cout << "x e' 1";
    break;
  case 2:
    cout << "x e' 2";
    break;
  default:
    cout << "valore di x ignoto";
  }
```

usando switch . . .

```
if (x == 1) {
  cout << "x e' 1";
  }
  else if (x == 2) {
    cout << "x e' 2";
  }
     else {
       cout << "valore di x ignoto";
     }
```

usando if+else . . .

65

L'istruzione di scelta: switch

- "dimenticare" il **break** a volte può essere utile:

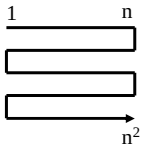
```
switch (x) {
  case 1:
  case 2:
  case 3:
      cout << "x e' 1, 2 o 3";
      break;
  default:
      cout << "x non e' 1, 2 o 3";
}
```

- Notare che **switch** si può usare soltanto per confrontare una espressione con delle costanti.
- Non si possono usare variabili o espressioni o intervalli di valori (**case (n*2):** e **case (1..3):** non vanno bene).
- Se occorre controllare intervalli di valori o valori non costanti bisogna usare una sequenza di frasi **if** ed **else if**

66

Esercizio!

Scrivere un programma che chiede all'utente un intero n in ingresso e che stampi in output gli interi da 1 ad n^2 secondo lo schema seguente



Ad esempio, se n == 5 l'output sarà

```
 1  2  3  4  5
10  9  8  7  6
11 12 13 14 15
20 19 18 17 16
21 22 23 24 25
```

Per ora non preoccupatevi della formattazione!

67

Funzioni

L'uso di funzioni permette di strutturare il programma in modo modulare.

- Una funzione è un blocco di istruzioni con un nome che viene eseguito in ogni punto del programma in cui viene chiamata la funzione usando il nome:

  Tipo Nome (Argomento1, Argomento2 , ...) Istruzione

  dove:
  - Tipo è il tipo del valore ritornato dalla funzione.
  - Nome è il nome con cui possiamo richiamare la funzione.
  - Argomento (possiamo indicarne quanti ne vogliamo, anche nessuno) è costituito da un nome di tipo seguito da un identificatore (ad esempio int x):
    - All'interno della funzione, un argomento si comporta come una variabile (locale).
    - Gli argomenti permettono di passare dei parametri quando la funzione viene chiamata.
    - I parametri sono separati da virgole.
  - Istruzione è il corpo della funzione: un blocco di istruzioni racchiuse tra parentesi graffe {} .

68

Funzioni

Esempio

```
// esempio di funzione
#include <iostream.h>
int main ()
{
  int z;
  z = somma (5,3);
  cout << "Il risultato e' " << z;
  return 0;
}

int somma (int a, int b)
{
  int r;
  r=a+b;
  return r;
}
```

```
Il risultato e' 8
```

69

Funzioni

Esempio equivalente

```
// esempio di funzione
#include <iostream.h>
int somma (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = somma (5,3);
  cout << "Il risultato e' " << z;
  return 0;
}
```

```
Il risultato e' 8
```

Notare:

```
int somma (int a, int b)

z = somma ( 5 , 3 );
```

Quando la funzione **somma** viene richiamata dal **main**, il controllo passa dalla funzione **main** alla funzione **somma**. I valori **5** e **3** passati come parametri vengono copiati nelle due variabili **int a** ed **int b** locali alla funzione **somma** .

70

Funzioni

Esempio equivalente

```
// esempio di funzione
#include <iostream.h>
int somma (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = somma (5,3);
  cout << "Il risultato e' " << z;
  return 0;
}
```

```
Il risultato e' 8
```

Notare:

```
int somma (int a, int b)

z = somma ( 5 , 3 );
```

La funzione **somma** dichiara una nuova variabile (**int r;**), e quindi, con l'istruzione **r=a+b;** , assegna ad **r** il risultato di **a** più **b** .
Siccome i valori passati come parametri sono 5 per **a** e 3 per **b**, il risultato è 8 .

71

Funzioni

Esempio equivalente

```
// esempio di funzione
#include <iostream.h>
int somma (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = somma (5,3);
  cout << "Il risultato e' " << z;
  return 0;
}
```

```
Il risultato e' 8
```

L'istruzione **return r;** infine termina la funzione **somma** e ritorna alla funzione che l'aveva richiamate (la funzione **main**) riprendendo l'esecuzione dal punto in cui era stata interrotta con la chiamata **somma(5,3)** .
L'istruzione **return** ha come argomento la variabile **r** (**return r;**), che al momento dell'esecuzione della **return** ha valore 8; di conseguenza **8** è il valore ritornato dalla funzione.

```
int somma (int a, int b)
 8
z = somma ( 5  ,  3 );
```

72

Funzioni

Esempio equivalente

```
// esempio di funzione
#include <iostream.h>
int somma (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
  int z;
  z = somma (5,3);
  cout << "Il risultato e' " << z;
  return 0;
}
```

```
Il risultato e' 8
```

Il campo di validità (scopo) delle variabili dichiarate in una funzione o in un blocco di istruzioni è limitato alla funzione stessa e al blocco di istruzioni e quindi tale variabile non può essere usata al di fuori di tale ambito.

Nell'esempio precedente **non** sarebbe possibile usare le variabili **a** , **b** ed **r** nella funzione **main** in quanto esse sono locali alla funzione **somma**.

Analogamente non sarebbe possibile usare la variabile **z** direttamente nella funzione **somma** in quanto essa è locale alla funzione **main**.

73

Funzioni

Altro esempio

```
// esempio di funzioni
#include <iostream.h>
int sottrazione (int a, int b)
{
  int r;
  r=a-b;
  return r;
}

int main ()
{
  int x=5, y=3, z;
  z = sottrazione (7,2);
  cout << "Il primo risultato e' " << z << '\n';
  cout << "Il secondo risultato e' " << sottrazione (7,2) << '\n';
  cout << "Il terzo risultato e' " << sottrazione (x,y) << '\n';
  z = 4 + sottrazione (x+2,y);
  cout << "Il quarto risultato e' " << z << '\n';
  return 0;
}
```

```
Il primo risultato e' 5
Il secondo risultato e' 5
Il terzo risultato e' 2
Il quarto risultato e' 8
```

74

Funzioni

Ricordiamo la sintassi di una dichiarazione di funzione:

Tipo Nome (Argomento1, Argomento2 , ...) Istruzione

Supponiamo di voler scrivere una funzione che deve soltanto scrivere qualcosa sullo schermo, senza ritornare un valore e neppure abbiamo bisogno di passargli dei parametri. Allo scopo il C fornisce un **particolare tipo `void`**. Osserviamo il seguente esempio:

```
// esempio di funzione void
#include <iostream.h>

void stampa (void)
{
  cout << "Sono una funzione!";
}

int main ()
{
  stampa ();
  return 0;
}
```

anche se non ci sono parametri bisogna sempre usare le parentesi tonde !

```
Sono una funzione!
```

In C++ l'indicazione di **`void`** come tipo del risultato o come parametro si può omettere scrivendo semplicemente **`stampa ()`**. L'uso esplicito di **`void`** è comunque consigliato.

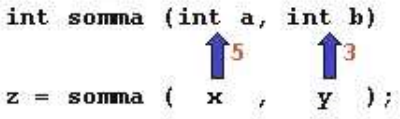
75

Funzioni: passaggio per valore

- Negli esempi di funzioni visti finora i parametri sono passati per valore. Questo significa che quando viene chiamata una funzione quello che viene passato alla funzione è il valore dei parametri (siano essi delle costanti o delle variabili o delle espressioni).
- In particolare, se il parametro è una variabile viene passato alla funzione il valore della variabile ma non la variabile stessa. Supponiamo, ad esempio, di richiamare la funzione somma nel modo seguente:

```
int x=5, y=3, z;
z = somma ( x , y );
```

- In questo caso viene richiamata la funzione somma passandogli i valori di **x** ed **y** , ossia 5 e 3 , ma non le variabili stesse:



- Una modifica di **a** o **b** all'interno della funzione **somma** non cambia i valori delle variabili **x** ed **y** esterne ad essa. Questo perché non sono state passate le variabili **x** ed **y** alla funzione somma ma soltanto il loro valore .

76

Funzioni: passaggio per riferimento

- Ci sono però casi in cui vogliamo modificare dall'interno di una funzione il valore di variabili definite esternamente alla funzione stessa.
- A questo scopo possiamo usare dei parametri passati per **riferimento**, come nella funzione **raddoppia** dell'esempio seguente:

```
// passaggio di parametri per riferimento
#include <iostream.h>

void raddoppia (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  raddoppia (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

```
x=2, y=6, z=14
```

Nella dichiarazione di **raddoppia** il tipo di ciascun parametro è seguito dal carattere e commerciale (&); esso sta ad indicare appunto un passaggio di parametro **per riferimento** invece dell'usuale passaggio per valore .

77

Funzioni: passaggio per riferimento

- Quando passiamo una variabile per riferimento è la variabile stessa che noi passiamo alla funzione e non soltanto il suo valore.
- Di conseguenza una modifica del valore del parametro all'interno della funzione modifica il valore della variabile passata come parametro.

```
void raddoppia (int& a, int& b, int& c)
                 x      y      z
raddoppia (      x  ,   y  ,   z);
```

- In altre parole noi abbiamo associato le variabili locali **a** , **b** e **c** (i parametri formali della funzione) alle variabili **x** , **y** e **z** (i parametri attuali passati nella chiamata alla funzione) in modo tale che **a** diventa sinonimo di **x** , **b** sinonimo di **y** e **c** sinonimo di **z** .
- Ricordando che una variabile è il nome di una zona di memoria in cui può essere memorizzato un valore (il valore della variabile appunto), dire che **a** e **x** sono sinonimi significa che essi sono nomi diversi per la stessa zona di memoria.
- Se **a** ed **x** sono sinonimi, una modifica del valore di **a** ha come conseguenza la modifica del valore registrato nella zona di memoria comune ad **a** e **x** e dunque anche il valore di **x** cambia.

78

Funzioni: passaggio per riferimento

- Il passaggio di parametri per riferimento permette di scrivere funzioni che calcolano più di un valore. Ad esempio, ecco una funzione che calcola il numero precedente ed il numero successivo del primo parametro che gli viene passato:

```
// calcolo di piu' di un valore
#include <iostream.h>

void precsucc (int x, int& prec, int& succ)
{
  prec = x-1;
  succ = x+1;
}

int main ()
{
  int x=100, y, z;
  precsucc (x, y, z);
  cout << "Precedente=" << y << ", Successivo=" << z;
  return 0;
}
```

```
Precedente=99, Successivo=101
```

79

Funzioni: valori di default

- Nella dichiarazione di una funzione si possono specificare dei valori di default per i parametri. I valori di default vengono usati nel caso in cui tali parametri vengano omessi nella chiamata di funzione. Ad esempio:

```
// valori di default per i parametri
#include <iostream.h>

int dividi (int a, int b=2)
{
  int r;
  r=a/b;
  return r;
}

int main ()
{
  cout << dividi (12);
  cout << endl;
  cout << dividi (20,4);
  return 0;
}
```

```
6
5
```

Notare che la corrispondenza tra parametri attuali e parametri formali **è posizionale** e quindi in una chiamata si possono omettere soltanto gli ultimi parametri.
Di conseguenza, in una chiamata di funzione con valori di default per i parametri è possibile omettere tutti i parametri da un certo punto in poi **ma non ometterne uno intermedio**.

80

Funzioni Sovraccaricate

- Due funzioni distinte possono avere lo stesso nome purché la lista degli argomenti sia diversa.
- Questo significa che possiamo dare lo stesso nome a più di una funzione purché esse abbiano un diverso numero di parametri o almeno un parametro di tipo diverso.

```
// funzione sovraccaricata
#include <iostream.h>

int dividi (int a, int b)
{
  return a/b;
}

float dividi (float a, float b)
{
  return a/b;
}

int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << dividi (x,y);
  cout << "\n";
  cout << dividi (n,m);
  cout << "\n";
  return 0;
}
```

```
2
2.5
```

Nell'esempio le due funzioni hanno lo stesso corpo ma questo non è necessario: funzioni con lo stesso nome possono anche fare cose completamente diverse.

81

Funzioni: Ricorrenza

- La ricorrenza (o ricorsività) è la proprietà di una funzione di poter essere richiamata da se' stessa, ossia all'interno del corpo della funzione possono comparire chiamate alla funzione stessa.
- Questa possibilità risulta particolarmente utile in certe situazioni in cui il valore da calcolare può essere definito per induzione . Ad esempio, il fattoriale di un numero intero n:

  `n! = n * (n-1) * (n-2) * (n-3) ... * 1`

  si può definire induttivamente nel seguente modo:

  - se **n=1** allora **n! = 1**
  - se **n>1** allora **n!= n * (n-1)!**

- Vediamo di seguito come realizzare un programma ricorsivo che calcola il fattoriale.

82

Funzioni: Ricorrenza

- Calcolo del fattoriale **n!**:

```
// calcolo del fattoriale
#include <iostream.h>

long fattoriale (long a)
{
  if (a > 1)
   return a * fattoriale (a-1);
  else
   return 1;
}

int main ()
{
  long n;
  cout << "Dammi un numero: ";
  cin >> n;
  cout << n << "!"<< " = " << fattoriale (n);
  return 0;
}
```

Attenzione a essere sicuri che ci sia una CONDIZIONE DI TERMINAZIONE certa !!

```
Dammi un numero:9
9! = 362880
```

83

Funzioni: prototipi

- Quando viene richiamata, una funzione deve essere già nota al compilatore.
- Per questa ragione abbiamo dovuto mettere sempre la funzione **main** alla fine.
- In realtà il compilatore per poter effettuare una chiamata di funzione ha bisogno di conoscere soltanto il nome della funzione ed il numero e tipo dei suoi parametri (il prototipo della funzione) mentre non ha alcun bisogno di conoscerne il corpo.
- Il C++ permette di dichiarare il prototipo di una funzione in modo tale da renderla nota al compilatore e rimandare in seguito la definizione vera e propria della funzione (comprendente anche il corpo).
- La forma di una dichiarazione di prototipo è la seguente :

  tipo nome (tipo_parametro1, tipo_parametro2, ...);

  ed è simile alla intestazione di una dichiarazione di funzione eccetto:
  - manca la parte Istruzione, ossia il blocco di istruzioni racchiuso tra parentesi graffe {} che costituisce il corpo della funzione.
  - termina con il carattere punto e virgola (;).
  - nell'elenco dei parametri basta indicare i tipi degli argomenti anche se è consigliabile mettere anche il nome del parametro benché esso sia opzionale.

84

Funzioni: prototipo

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

void dispari (int a);
void pari (int a);
int main ()
{
  int i;
  do {
    cout << "Scrivi un numero: (0 per uscire)";
    cin >> i;
    dispari (i);
  } while (i!=0);
  return 0;
}

void dispari (int a)
{
  if ((a%2)!=0) cout << "Il numero è dispari.\n";
  else pari (a);
}
void pari (int a)
{
  if ((a%2)==0) cout << "Il numero è pari.\n";
  else dispari (a);
}
```

```
Scrivi un numero (0 per uscire): 9
Il numero è dispari.
Scrivi un numero (0 per uscire): 6
Il numero è pari.
Scrivi un numero (0 per uscire): 1030
Il numero è pari.
Scrivi un numero (0 per uscire): 0
Il numero è pari.
```

85

Funzioni: prototipo

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

void dispari (int a);
void pari (int a);
int main ()
{
  int i;
  do {
    cout << "Scrivi un numero: (0 per uscire)";
    cin >> i;
    dispari (i);
  } while (i!=0);
  return 0;
}

void dispari (int a)
{
  if ((a%2)!=0) cout << "Il numero è dispari.\n";
  else pari (a);
}
void pari (int a)
{
  if ((a%2)==0) cout << "Il numero è pari.\n";
  else dispari (a);
}
```

```
Scrivi un numero (0 per uscire): 9
Il numero è dispari.
Scrivi un numero (0 per uscire): 6
Il numero è pari.
Scrivi un numero (0 per uscire): 1030
Il numero è pari.
Scrivi un numero (0 per uscire): 0
Il numero è pari.
```

Molti programmatori esperti consigliano di prototipare tutte le funzioni.
Questo è particolarmente utile quando un programma contiene molte funzioni o le definizioni delle funzioni sono molto lunghe.
In tal caso raccogliere tutti i prototipi nello stesso posto all'inizio facilita la ricerca se non si ricorda come devono essere richiamate (numero e tipo dei parametri).

86