Corso sul linguaggio C++

Modulo 3

1 - Le funzioni

Prerequisiti

- Programmazione elementare in C++
- Tecnica top-down
- Concetto matematico di funzione

Introduzione

In questa lezione fissiamo l'attenzione su come realizzare un programma modulare, ossia un programma scomposto in **funzioni** o **sottoprogrammi** o **moduli.**

Attraverso la tecnica **top-down**, è possibile rappresentare la soluzione di un problema come composta da sottoproblemi più semplici.

Una **funzione** deve:

- svolgere **un solo compito**
- essere di **uso generale**
- rappresentare la soluzione di un singolo **sottoproblema**

Argomenti

- Istanza di una funzione
- Definizione di una funzione
- Visibilità globale delle variabili
- Visibilità locale delle variabili
- Vantaggi della visibilità
- Valore di ritorno
- Il tipo **void**
- Parametri formali
- Parametri attuali
- Scopo dei parametri
- Utilizzo dei parametri
- La struttura di una funzione
- Progettare una funzione
- Compilazione e link di funzioni
- Esecuzione di funzioni
- Il prototipo della funzione
- Le funzioni di libreria
- Libreria **ctype.h**
- Libreria **cmath**
- Libreria **cstdlib**
- Librerie utente
- Passaggio di parametri
- Passaggio per valore
- Passaggio per indirizzo
- Funzioni inline
- Generazione pseudocasuale
- Un gioco con i dadi

Informazioni generali

In questa Unità didattica mettiamo in pratica la tecnica **top down** l'importanza di suddividere un programma in parti più piccole dette **sottoprogrammi** o **funzioni**: ciò comporta

- un codice più leggibile
- una progettazione più semplice
- una maggior semplicità
  - nel collaudo
  - nella correzione di eventuali errori
  - nella eventuale modifica del codice.

Iniziamo con un esempio semplice come la somma di due numeri

Istanza di una funzione

```
#include <iostream>
#include <cstdlib>
using namespace std;
void lettura();
void calcolo();
void stampa();
int a, b, somma;
int main()
{  lettura();
   calcolo();
   stampa();
   system("pause");
   return(0);
}
```

Le funzioni utilizzate vanno dichiarate in questo modo. Queste righe sono dette intestazione o **prototipo** delle funzioni.

Quando il programma fa uso di funzioni, un modo per comunicare i valori delle variabili tra le varie funzioni è dichiarare le variabili esternamente alle funzioni (**variabili globali**)

la funzione *main*() contiene le **istanze** delle funzioni:
- *lettura*()
- *somma()*
- *stampa*().

Le funzioni vanno definite di seguito alla funzione **main**() come è mostrato di seguito.

Definizione di funzioni

```
void lettura()
{
  cout<<"Primo valore: ";
  cin>>a;
  cout<<"Secondo valore: ";
  cin>>b;
}
void calcolo()
{
  somma=a+b;
}
void stampa()
{
  cout<<"La somma e' "<<somma<<endl;
}
```

Vedremo più avanti il significato di **void**

I blocchi di istruzioni nei riquadri si dicono **definizione** della funzione

Visibilità globale delle variabili

Il programma può essere rappresentata come mostrato nel disegno.

static int a, b, somma;

int main ()

void lettura ()

void calcolo ()

void stampa ()

variabili globali

Le **variabili globali** *a, b* e *somma*:

- sono dichiarate all'esterno del **main**()
- hanno **visibilità globale**, ossia possono essere *utilizzate* e *modificate* da tutte le funzioni; quando il controllo torna al programma chiamante, la **variabile risulta modificata rispetto al valore originario**

Visibilità globale delle variabili

```
/* Programma di esempio sulle variabili a visibilità globale */
#include <iostream.h>
int i;    /*  variabile globale */
void funzione(void);
void main()
{   i=5;
    cout<<"\nNel main: lvalue di i = "<<&i;
    cout<<"\nNel main: rvalue di i = "<<i;
    funzione(); /* non ci sono argomenti poichè i è globale */
    cout<<"\nNel main dopo la funzione: lvalue di i = "<<&i;
    cout<<"\nNel main dopo la funzione: rvalue di i = "<<i;
    return;
}
void funzione(void)
{   i=i*i;
    cout<<"\nNella funzione: lvalue di i = "<<&i;
    cout<<"\nNella funzione: rvalue di i = "<<i;
}
```

Il carattere **&** stampa l'*lvalue* della variabile

Se si prova ad eseguire il programma si nota che in stampa gli *lvalue sono sempre uguali* a riprova che la variabile su cui opera la funzione *è la stessa* di quella presente nel main.

Inoltre, il valore di *i*, modificato nella funzione, viene mantenuto modificato anche nel **main**(), proprio perché si tratta della **stessa** variabile.

Visibilità globale delle variabili

La variabile *i* dell'esempio precedente:

- è dichiarata all'esterno del **main**() come **variabile globale**
- **è visibile** nelle funzioni esterne al **main**()
- **può essere usata e modificata** da altre funzioni esterne
- quando il controllo torna al programma chiamante, **risulta modificata rispetto al valore originario**

Visibilità globale delle variabili

```
/* Programma di esempio sulle variabili a visibilità globale */
#include <iostream.h>
int i;      /* variabile globale */
void funzione(void);
void main(void)
{   i=5;
    cout<<"\nNel main: lvalue di i = "<<&i;
    cout<<"\nNel main: rvalue di i = "<<i;
    funzione(); /* non ci sono argomenti poichè i è globale */
    cout<<"\nNel main dopo la funzione: lvalue di i = "<<&i;
    cout<<"\nNel main dopo la funzione: rvalue di i = "<<i;
    return;     }
void funzione(void)
{   int i=8;           /* variabile locale alla funzione */
    cout<<"\nNella funzione: lvalue di i = "<<&i;
    cout<<"\nNella funzione: rvalue di i = "<<i;    }
```

Se si prova ad eseguire il programma si nota che in stampa gli *lvalue sono diversi* poiché la variabile è definita anche localmente alla funzione. **Il valore nel main() al ritorno è quello originario**

Visibilità globale delle variabili

La variabile *i* dell'esempio precedente:

- è dichiarata all'esterno del **main**() come **variabile globale**
- **è visibile** nelle funzioni esterne al **main**() **a meno che in esse non sia ridichiarata**
- se in una funzione è dichiarata di nuovo, la **precedenza** viene data a quest'ultima, poiché è la **variabile visibile al momento**. Al termine della funzione tale variabile "muore" e non è più visibile all'esterno

Visibilità locale delle variabili

La visibilità di una variabile può essere limitata ad una sola funzione. In questo caso, la variabile va dichiarata all'interno della funzione.

```
int a, b, c;  // globali

int main ()
{ int d, e;
   …
}

void f1 ()

void f2 ()
```

variabili locali al **main**().

Le **variabili locali** *d* ed *e* dell'esempio:

- sono dichiarate all'interno del **main**()
- hanno **visibilità locale** al **main**() ma **non sono visibili** nelle altre funzioni
- le altre funzioni **possono usare variabili locali con lo stesso nome e modificarle** senza rischio di confusione. Quando il controllo torna al programma chiamante, la **variabile riprende automaticamente il valore originario**

Visibilità locale delle variabili

Per verificare che ogni funzione possiede un proprio **ambiente**, consideriamo una variabile *i* dichiarata nel **main()** ed *una sua omonima*, dichiarata localmente ad un sottoprogramma chiamato *funzione*() e proviamo a stampare l'**lvalue** delle due variabili, per vedere quando rappresentano effettivamente la *stessa variabile*, e quando invece rappresentano variabili diverse.

Visibilità locale delle variabili

```
#include <iostream>
#include <cstdlib>
using namespace std;
void funzione();
int main()
{  int i=10;
   cout<<"\nNel main: lvalue di i = "<<&i;
   cout<<"\nNel main: rvalue di i = "<<i;
   funzione(i);      /* i è il valore attuale */
   system("Pause");
   return(0);
}
void funzione()
{  int i=5;
   cout<<"\nNella funzione: lvalue di i = "<<&i;
   cout<<"\nNella funzione: rvalue di i = "<<i<<endl;
}
```

Il programma stampa **lvalue** e **rvalue** della *i* nel programma e nella funzione.

Se si fa eseguire il programma si noterà che gli **lvalue** sono diversi come gli **rvalue** (nel **main**() vale 10, nella funzione vale 5)

Visibilità locale delle variabili

Ciò significa che la funzione opera su una **copia della variabile i**.

Qualunque operazione si esegua su *i* nella funzione, al ritorno nel **main**() riassumerà il valore che aveva nel **main**() prima dell'istanza.

Le due variabili *i* dell'esempio precedente:

- sono rispettivamente una **locale al main()**, l'altra **locale a funzione();**
- il valore che assume nel **main**() **NON è visibile** in funzione() e viceversa;
- può essere *ridichiarata, utilizzata e modificata da altre funzioni*, ma quando il controllo torna al programma chiamante, la *variabile riprende automaticamente il valore originario.*

Vantaggi della visibilità

La conoscenza e l'uso delle variabili con visibilità consente di ottenere programmi e funzioni con:

- maggiore leggibilità
- maggiore facilita di correzione degli errori
- maggiore generalità
- migliore utilizzo della memoria

Valore di ritorno

Nel programma della somma di due numeri, il calcolo della somma può essere fatto anche mediante la seguente funzione:

```
int calcolo()
{
   return a+b;
}
```

La parola chiave **void** viene sostituita con **int**, che indica il tipo del **valore di ritorno** della funzione *calcolo*().

La parola chiave **return** indica il valore del risultato calcolato dalla funzione *calcolo*(). Questo valore si dice **valore di ritorno.**

Valore di ritorno

La funzione **main**() allora assume la forma:

```
int main ()
{
  ….
  lettura();
  cout<<(a + " + " + b + " = " + calcolo());
}
```

L'istanza della funzione *calcolo*() viene effettuata come se questa fosse una variabile.

ATTIVITA': scrivere una funzione che calcoli l'area di un trapezio isoscele note le basi *B* e *b* e l'altezza *h* ed una analoga funzione per il perimetro

Valore di ritorno

Variabili globali:

```
float B, b, h;
….

float area()
{
    return ((B + b) * h / 2);
}
```

Il valore di ritorno è l'area del trapezio.

Variabili globali:

```
float B, b, h;
….

float perimetro()
{  float seg_lat=(B - b) / 2;
   float lObliquo = Math.sqrt(Math.pow(h, 2.0)
                  + Math.pow(seg_lat, 2.0));
   return (B + b + 2 * lObliquo);
}
```

Il valore di ritorno è il perimetro del trapezio.

Il tipo **void**

Alcune volte la funzione **non deve restituire alcun valore**, ma solo eseguire certe istruzioni: in tal caso per il tipo del risultato della funzione si usa la parola chiave **void** e non è necessario il **return.**
Ad esempio:

```
void stampa (int x, int y)        /* Stampa dei due valori */
{   cout<<x<< " "<< y;            /* return non serve poiché la
                                     funzione è di tipo void */
}

void  titolo ()                   /* Esempio di funzione di tipo void
{ cout<<TITOLO;                      senza argomenti */
}
```


Parametri formali

La funzione *calcolo*()

```
int calcolo()
{
    return a+b;
}
```

presenta però lo svantaggio di eseguire l'operazione sempre sulle stesse variabili *a* e *b*. Se si volesse eseguire l'addizione tra le variabili *c* e *d*, occorrerebbe scrivere una funzione praticamente identica, avente come valore di ritorno *c+d*.

Come si può generalizzare la funzione *calcolo*()?

Parametri formali

Pensiamo la funzione *calcolo*() scritta come segue:

```
int calcolo (int x, int y)
{
  return x + y;
}
```


nella quale le variabili *x* ed *y* servono ad indicare due ingressi generici della funzione (i valori da sommare) e prendono il nome di **parametri formali**.

I valori di *x* ed *y* verranno stabiliti solo al momento dell'esecuzione della funzione.

Parametri attuali

```
#include <iostream>
#include <cstdlib>
using namespace std;
void lettura();
int calcolo (int x, int y);
int a,b ;
int main()
{
   lettura();
   cout<<calcolo(a, b)<<endl;
   system("pause");
   return(0);
}
```

Scopo dei parametri

Lo scopo dei parametri formali (nel prototipo) ed attuali (nell'istanza) è quello di costituire una **interfaccia,** tra la funzione ed il programma chiamante, che consenta di istanziare la funzione con certi valori

L'input della funzione deve avvenire mediante i **parametri** e **l'output** è il risultato che la funzione fornisce al programma chiamante mediante il **return.**

Utilizzo dei parametri

I **parametri formali**:

- **NON** vanno dichiarati nella funzione **main**()
- servono a far sapere al programma chiamante **l'ordine**, il **numero** ed il **tipo** degli ingressi della funzione (chiamati anche **argomenti**)

Nell'istanza della funzione i **parametri attuali**:

- devono essere dichiarate nel programma chiamante
- devono essere nello stesso **ordine**, **numero** e **tipo** dei parametri formali

Utilizzo dei parametri

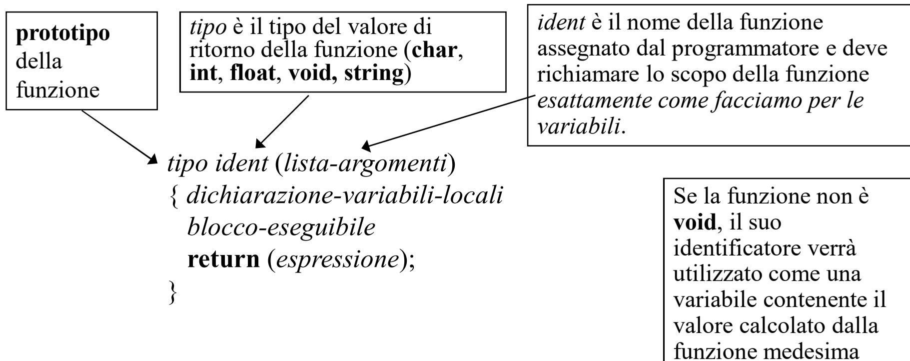
Al momento dell'istanza di una funzione, i **parametri attuali**, che devono essere nello stesso

- ordine
- numero
- tipo

dei **parametri formali**, vanno a sostituirsi a questi ultimi e la funzione viene eseguita con i valori attuali o effettivi.

La struttura di una funzione

Da quanto abbiamo detto, la struttura generale di una funzione è:



La struttura di una funzione

Da quanto abbiamo detto, la struttura generale di una funzione è:

lista-argomenti contiene l'elenco dei parametri formali che passiamo alla funzione affinchè essa possa svolgere un dato compito. Ad esempio, se la funzione calcola l'area di un quadrato, occorre fornire ad essa un argomento che rappresenta la misura del lato.

tipo ident (*lista-argomenti*)
{ *dichiarazione-variabili-locali*
blocco-eseguibile
return (*espressione*);
}

La struttura di una funzione

Da quanto abbiamo detto, la struttura generale di una funzione è:

tipo ident (*lista-argomenti*)
{ *dichiarazione-variabili-locali*
blocco-eseguibile
return (*espressione*);
}

La sezione *dichiarazione-variabili-locali* serve a dichiarare le variabili che userà la funzione. Si noti che queste variabili **NON** saranno utilizzabili all'esterno della funzione stessa perché sono *variabili locali*.

La struttura di una funzione

Da quanto abbiamo detto, la struttura generale di una funzione è:

tipo ident (*lista-argomenti*)
{ *dichiarazione-variabili-locali*
  blocco-eseguibile
  return (*espressione*);
}

La sezione *blocco-eseguibile* conterrà tutte le istruzioni che la funzione dovrà svolgere una volta che verrà chiesta la sua esecuzione.

La struttura di una funzione

Da quanto abbiamo detto, la struttura generale di una funzione è:

espressione rappresenta il risultato prodotto dalla esecuzione della funzione. La parola chiave **return** indica il termine delle istruzioni ed ha il compito di assegnare *espressione* al nome della funzione

tipo ident (*lista-argomenti*)
{ *dichiarazione-variabili-locali*
  blocco-eseguibile
  return (*espressione*);
}

Nel caso di funzioni **void** il **return** non viene utilizzato

Progettare una funzione

Per scrivere una funzione, dobbiamo prima definire:

- il **prototipo**
  - **nome** della funzione
  - gli eventuali **parametri** da passare alla funzione
  - il tipo del **valore di ritorno**
- le **istruzioni** da eseguire

Compilazione e link di funzioni

Il **compilatore**, nel tradurre il programma principale, opera anche sulle funzioni:

- controlla il tipo del valore di ritorno dalla funzione
- ordine, numero e tipo dei parametri

Avviene tramite il prototipo

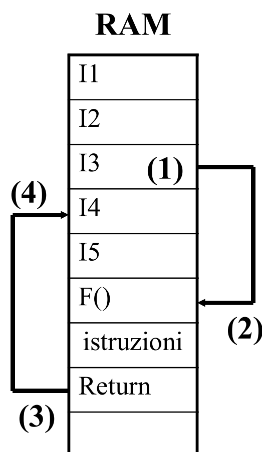
- alloca in memoria il codice della funzione
- fa intervenire il **linker**, un programma che:
  - collega il codice della funzione con il programma principale
  - genera il programma *eseguibile*

Esecuzione di funzioni

L'**istanza** di una funzione all'interno del programma chiamante, produce:

- **sospensione** temporanea del chiamante
- **salvataggio** del valore del program counter e dei valori degli altri registri al momento dell'istanza, in una apposita area di memoria detta **pila di attivazione**
- **esecuzione** della funzione
- **ripristino** dei valori dei registri salvati in precedenza
- **ripristino** dell'esecuzione del programma chiamante

Esecuzione di funzioni



Il programma è formato dalle 5 istruzioni da I1 ad I5. L'istruzione I3 è l'istanza alla funzione F(). Il programma viene sospeso (1) all'istruzione I3. Dopo il salvataggio dei registri nella **pila di attivazione**, viene eseguita (2) la funzione F(). Al termine della esecuzione (3) della funzione con il Return, vengono ripristinati i valori dei registri, ripristinandoli dalla pila di attivazione, e ripresa l'esecuzione (4) del programma sospeso

Il prototipo della funzione

Poiché l'utilizzo delle funzioni avviene nel **main**(), mentre la loro definizione è posta dopo di questo, come fa il compilatore a conoscere:

- **il tipo restituito da una funzione?**
- **il numero degli argomenti di essa?**
- **il tipo degli argomenti di essa?**

Il **prototipo** svolge il compito di far conoscere al **main**() in anticipo la presenza delle funzione attraverso la sua **dichiarazione.**

All'atto pratico è sufficiente copiare la riga di intestazione prima del **main**() nel punto detto.

OSSERVAZIONE: La riga di dichiarazione (il prototipo) deve essere terminata con il ";".

Le funzioni di libreria

Il C++ mette a disposizione del programmatore un cospicuo numero di librerie (*header file*) all'interno delle quali possiamo trovare i prototipi di moltissime funzioni già implementate.

I prototipi delle funzioni raggruppati in ogni libreria operano su **tipi di dato comuni**.

Il nome del file header ha estensione .h e ricorda il tipo di dato su cui operano le funzioni contenute.

Le funzioni di libreria

Le principali librerie sono indicate nella seguente tabella (alcune di esse sono caratteristiche del linguaggio C)

Header file	Contenuto
ctype.h	Prototipi per la gestione dei caratteri
string	Prototipi funzioni di stringa (esaminati in Unità 2)
float.h	Costanti di tipo **float** (virgola mobile)
limits.h	Intervalli per i vari tipi di dato
cmath	Prototipi funzioni matematiche (ad es. trigonometriche)
iostream	Prototipi funzioni standard di I/O (**cin** e **cout**)
cstdlib	Prototipi funzioni varie
time.h	Prototipi di funzioni per elaborazione data e ora

Libreria ctype.h

Le principali funzioni della libreria **ctype.h** sono

Funzione	Effetto	Restituisce
int isalpha(int *c*)	Verifica se *c* è alfanumerico	0 se falso, non 0 se vero
int i**sascii(int** *c*)	Verifica se *c* è tra 0 e 255	c.s.
int iscntrl(int *c*)	Verifica se *c* è tra 0-31 o 255	c.s.
int isdigit(int *c*)	Verifica se *c* è carattere tra 0 e 9	c.s.
int islower(*c*)	Verifica se *c* è minuscolo	c.s.
int isupper(*c*)	Verifica se *c* è maiuscolo	c.s.
int ispunct(*c*)	Verifica se *c* è car. di punteggiatura	c.s.
int isspace(*c*)	Verifica se *c* è uno spazio	c.s.
int tolower(int *c*)	Converte *c* in minuscolo	Carattere convertito
int toupper(int *c*)	Converte *c* in maiuscolo	Carattere convertito

Libreria cmath

Le principali funzioni della libreria **cmath** sono

Funzione	Restituisce
int abs(int n)	Valore assoluto di *n*
double ceil(double n)	Intero superiore a *n*
double floor (**double** n)	Intero inferiore a *n*
double sin (**double** n)	Calcola sen(*n*)
double cos (**double** n)	Calcola cos(*n*)
double tan (**double** n)	Calcola tg(*n*)
double exp(**double** n)	Calcola l'esponenziale e^n
double log (**double** n)	Calcola il logaritmo naturale di n $\log_e n$
double log10 (**double** n)	Calcola il logaritmo decimale di *n* $\log_{10} n$
double pow (**double** b, **double** e)	Calcola la potenza b^e
double sqrt(double n)	Calcola la radice quadrata di *n*

Libreria cstdlib

Le principali funzioni della libreria **cstdlib** sono

Funzione	Restituisce
double atof (**char** *s)	Converte la stringa *s* in un valore **double**
double atoi (**char** *s)	Converte la stringa *s* in un valore **int**
double atol (**char** *s)	Converte la stringa *s* in un valore **long**
int rand (void)	Valore casuale tra 0 e RAND_MAX
void srand (**unsigned** s)	Inizializza il generatore dei numeri casuali con il seme *s*, in modo da avere valori diversi in esecuzioni multiple del programma (v. esercizio in seguito)
int system (**char** *s)	Esegue il comando DOS contenuto in *s*

Librerie utente

Esiste la possibilità che il programmatore crei egli stesso dei file di libreria, inserendovi le funzioni di sua utilità.

Per fare ciò occorre:

- creare un file *nomefile* contenente le sole funzioni di utilità
- salvare il file come *nomefile*.h
- includere il file nel programma che userà le funzioni mediante la direttiva #include *nomefile*.h

Passaggio di parametri

Come si ricorderà, quando si passa un parametro ad una funzione, il valore di questo, al ritorno al programma chiamante, viene ripristinato. Si dice che il parametro è stato **passato per valore**.

Se si vuole che il valore di questa variabile, modificato, venga restituito tale al chiamante, si può dichiararlo globale, ma così facendo, si perderebbe la sua privatezza. Vediamo una efficace alternativa.

I parametri attuali possono essere passati alle funzioni secondo due modalità

- **passaggio per valore**
- **passaggio per indirizzo**

Passaggio per valore

Il passaggio di **parametri per valore** è quello che abbiamo esaminato finora. I parametri formali assumono il valore dei parametri attuali, ma restano isolati nella funzione in quanto sono copie dei valori attuali.

I parametri formali possono essere modificati nella funzione, ma al ritorno al programma chiamante, vengono distrutti e il valore torna quello che avevano al momento dell'istanza.

Passaggio per valore

Se si vuole che un parametro formale **NON** venga modificato dalla funzione si usa la parola chiave **const** nel seguente modo:

int somma (**const int** a, **int** b)

{ b=b+a; //istruzione corretta

a=a+1; //dà errore poiché *a* è passato come costante

……..

}

Passaggio per indirizzo

Se invece si vuole che il parametro formale venga modificato e che le modifiche vengano poi trasmesse al programma chiamante, si utilizza il **passaggio per indirizzo**.

Ad esempio

```
void scambia(int *a, int *b)
{   int temp;
    temp=*a;
    *a=*b;
    *b=temp;  //istruzione corretta
}
```

In questo modo abbiamo passato alla funzione **NON** la variabile, ma il suo indirizzo di memoria. **Quindi le eventuali modifiche verranno effettuate sul contenuto degli indirizzi, ossia sulle variabili originali**.

Passaggio per indirizzo

Ovviamente, nel programma chiamante si dovrà tenere conto che i parametri attuali devono rappresentare indirizzi e l'istanza della funzione sarà:

```
int main()
{  ………
   scambia(&x,&y);
   ………
   ………
}
```

Il simbolo **"&"** sta ad indicare "indirizzo di" e posto davanti ad una variabile ne indica l'indirizzo di memoria.

In questo modo si passa per valore l'indirizzo delle due variabili che risultano quindi accessibili anche per eventuali modifiche

Passaggio per indirizzo

Un modo alternativo è il seguente:

```
void scambia(int &a, int &b)
{   int temp;
    temp=a;
    a=b;
    b=temp;  //istruzione corretta
}
void main()
{   ……….
    scambia(x,y);
    ……….
}
```

In questo caso alla funzione si passano le variabili attuali *x* ed *y* che vengono trasformate in indirizzi.

Il funzionamento è equivalente al metodo precedente, ma è preferibile in quanto più semplice dal punto di vista sintattico.

Passaggio per indirizzo

ATTIVITA': riscrivere il programma principale *somma.c* mostrato all'inizio dell'Unità per il calcolo della somma di due numeri, facendo uso delle due funzioni *lettura (...)* e *calcolo (...)*.

```
#include <iostream>
#include <cstdlib>
using namespace std;
void lettura(int &x, int &y);
int calcolo (int x, int y);
int main()
{  int a, b;
   lettura (a, b);
   cout<<"La somma e' "<<calcolo(a, b)<<endl;
   system("pause");
   return(0);
}
```

Passaggio per indirizzo (lettura)

Passaggio per valore (calcolo)

Le variabili sono divenute tutte locali al **main**() grazie al passaggio dei parametri

Passaggio per indirizzo

```
void lettura  (int &x, int &y)
{
   cout<<"Primo valore: ";
   cin>>x;
   cout<<"Secondo valore: ";
   cin>>y;
}

int calcolo (int x, int y)
{
  return x + y;
}
```

ATTIVITA': scrivere la codifica delle due funzioni *lettura (...)* e *calcolo (...)*.

Attraverso il meccanismo di passaggio dei parametri (per valore o per indirizzo, le funzioni diventano *parti di programma indipendenti dal contesto e riutilizzabili anche in altri programmi* e questo è uno dei vantaggi del loro utilizzo.

Funzioni inline

Quando una funzione prevede una definizione non troppo complessa, si può usare la **definizione inline della funzione** (dette anche **macro**)

La definizione inline va posta subito fianco del prototipo nella parte iniziale del codice.

Ad es:

```
double cubo(int n) {return n*n*n;}
```

Le funzioni inline consentono la scrittura di programmi più compatti e sintetici

Generazione pseudocasuale

```
#include <iostream>
#include <cstdlib>
int main()
{ int x=0;
do {
      cout<<rand()<<endl;
      x++;
   } while (x<10);
   system("PAUSE");
   return 0;
}
```

Il programma di esempio a fianco usa la funzione **rand**() per generare pseudocasualmente una serie di 10 valori.

Provando il programma, si nota però che la serie generata è sempre la stessa.

Generazione pseudocasuale

```
#include <iostream>
#include <cstdlib>
int main()
{ int x=0, seed;
   cout<<"Seme: "; cin>>seed;
   srand (seed);  //seme
   do {
         cout<<rand()<<endl;
         x++;
     } while (x<10);
   system("PAUSE");
   return 0;
}
```

Una prima soluzione è quella di innescare la generazione di numeri mediante un valore detto "seme" da passare come parametro alla funzione **srand** (*seed*). Leggendo ad ogni esecuzione il valore di *seed*, la serie generata è sempre diversa dalla precedente.

Generazione pseudocasuale

```
#include <iostream>
#include <cstdlib>
#include <time.h>
int main()
{ int x=0;
    srand (time (0));
    do {
      cout<<rand() % 6 +1<<endl;
      x++;
    } while (x<10);
    system("PAUSE");
   return 0;
}
```

Una seconda soluzione evita di leggere il seme dall'esterno ed usa il **valore dell'orologio interno** al momento dell'esecuzione. Questo valore, ottenuto dalla funzione **time(0)**, viene passato alla funzione **srand** (). La sequenza generata è diversa ad ogni esecuzione.

In questo modo, ad esempio, è possibile simulare il lancio di un dado; il range dei valori pseudocasuali è tra 1 e 6 grazie all'istruzione **rand**() % 6 + 1

Un gioco con i dadi

Applichiamo i concetti visti finora ad un gioco con i dadi, che si svolge come segue.

Il giocatore lancia 2 dadi e se ottiene

- 7 o 11 al primo lancio, il giocatore vince
- 2, 3 o 12 il giocatore perde
- 4, 5, 6, 8, 9 o 10 questo sarà il suo punteggio P. In tal caso, lancia nuovamente i dadi fino a quando o esca di nuovo il punteggio P (nel qual caso il giocatore vince) oppure esca il valore 7 (nel qual caso il giocatore perde).

Un gioco con i dadi

Consideriamo

- **enum** stato {CONTINUA, VINCE, PERDE}; che esprime lo stato del gioco
- una funzione **int** lancio **(void)**, che simula il lancio di due dadi e restituisce il punteggio ottenuto
- La struttura di controllo **switch** (P) che analizza i vari valori del punteggio P
- Due costanti stringa VITTORIA="Hai vinto!", SCONFITTA="Hai perso!"

Un gioco con i dadi

L'algoritmo potrebbe essere il seguente:

Inizio

Inizializza il seme con l'orologio

P=lancio(); // punteggio del primo lancio

Nel caso che (P)

7,11: *vittoria*;

2,3,12:sconfitta;

continua i lanci fino a quando si ottiene un punteggio P' uguale a P oppure a 7;

Se (P'==P) **allora** *vittoria*

altrimenti *sconfitta*

Fine

ATTIVITA': codificare per esercizio, l'algoritmo proposto e verificarne la corretta esecuzione.

Cosa ho imparato

- Quali sono i vantaggi della programmazione modulare
- Qual è la struttura di una funzione
- Cosa si intende per ambiente locale e globale
- Come si possono rendere generali le funzioni
- Qual è la differenza tra il passaggio per valore e per indirizzo
- Quali sono le principali librerie del C++
- Cosa avviene quando si compila e si linka una funzione
- Cosa si intende per valore di ritorno di una funzione

Cosa ho imparato a fare

- Progettare un'applicazione modulare
- Utilizzare funzioni con parametri
- Utilizzare funzioni con valore di ritorno
- Saper istanziare funzioni di diverso tipo
- Utilizzare le librerie più comuni del C++
- Utilizzare il passaggio dei parametri per valore e per indirizzo

Terminologia

- Funzione
- Dichiarazione
- Definizione
- Istanza
- Prototipo
- Pila di attivazione
- Parametri formali
- Parametri attuali o effettivi
- **void**
- Passaggio parametri per valore
- Passaggio parametri per indirizzo
- Visibilità delle variabili
- Variabili locali
- Variabili globali
- Funzioni inline
- Libreria di funzioni
- Generazione pseudocasuale di numeri

Altre fonti di informazione

- J. Purdum, C – ed. Jackson
- A.Lorenzi- D.Rossi, Il linguaggio C++ - ed. ATLAS