Corso di Laurea in Ingegneria Biomedica

Informatica Medica

I semestre

Docenti: Fabio Solari e Manuela Chessa

Prof. Fabio Solari: fabio.solari@unige.it (010-3532059)
Prof. Manuela Chessa: manuela.chessa@unige.it (010-3532289)

http://www.pspc.unige.it/~fabio
http://www.pspc.unige.it/~manuela

Informatica Medica – C++

Obiettivi formativi

In generale, conoscere la sintassi ed i principali costrutti del linguaggio C++ e sviluppare programmi ad oggetti utilizzando le caratteristiche di tale linguaggio.

Testo di riferimento

- Materiale distribuito a lezione, reso disponibile sul sito web: http://www.pspc.unige.it/~manuela/teaching/informatica_medica_I.html
- H. Schildt. *C++: La guida completa (4a ed.), McGraw-Hill.*

Altri riferimenti bibliografici

- B. Eckel, *Thinking in C++ (2a ed.), Prentice Hall, 2000. Online a:* http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html
- B. Stroustrup, *C++ Linguaggio, libreria standard, principi di programmazione (3a ed.), Addison-Wesley, 2000.*

Informatica Medica – C++

Forme didattiche

Lezioni ed esercitazioni di laboratorio.

Tipologia dell'esame

- Valutazione delle esercitazioni, prova intermedia scritta e orale.

In particolare, le modalità dell'esame sono**:**

- Svolgere le esercitazioni.
- Consegnare le esercitazioni mediante l'apposito servizio su Aulaweb (www.aulaweb.unige.it), seguendo le modalità comunicate e indicate sui siti.
- La *prova intermedia scritta* consiste nello svolgimento di alcuni esercizi sugli argomenti presentati a lezione.
- La *prova intermedia orale* consiste in una discussione sugli argomenti sviluppati nelle esercitazioni e sugli argomenti presentati a lezione. Potrà accedere alla prova intermedia orale solo chi avrà ottenuto la sufficienza nella prova intermedia scritta.

Sommario

- Sviluppo di un'applicazione eseguibile
- C e C++
- I/O dati
- Istruzioni di controllo
- Puntatori
- Funzioni
- Array
- Allocazione dinamica
- Generazione di numeri pseudo-casuali
- Programma su più file

Sviluppo di un'applicazione eseguibile

- Il codice sorgente scritto dal programmatore non è direttamente eseguibile dalla CPU (Central Processing Unit), è necessario *tradurlo* in linguaggio macchina: un *file binario* contenente microistruzioni gestibili dalla CPU.
  - Diverse architetture di processori prevedono diversi linguaggi macchina.

- Un **traduttore** "*elabora*" un codice sorgente (un programma) per trasformarlo in uno equivalente ma in un diverso linguaggio.

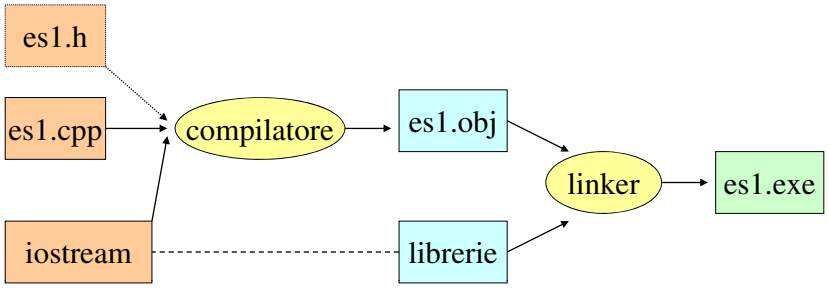
Sviluppo di un'applicazione eseguibile

- I **compilatori** leggono il programma sorgente (per esempio in linguaggio C++) e lo *traducono* in linguaggio macchina, generando un *programma oggetto*. La fase di compilazione è accompagnata dalla rilevazione degli errori.

- Per produrre il file eseguibile è necessario *collegare* (utilizzando il **linker***)* tra loro i diversi *file oggetto* e le *librerie di funzioni standard* che sono file forniti insieme al programma compilatore.

Sviluppo di un'applicazione eseguibile

- Le *librerie di funzioni standard* sono raccolte di programmi *oggetto* che forniscono particolari funzionalità: per esempio, funzioni matematiche, funzioni per la grafica, algoritmi e strutture dati. I prototipi di tali funzioni sono nei *file header*.



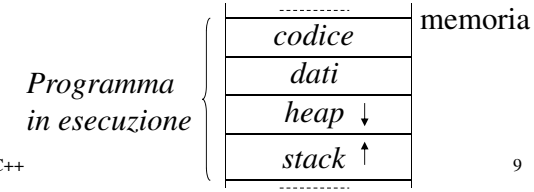
Sviluppo di un'applicazione eseguibile

- Gli strumenti di sviluppo sono un insieme di programmi (editor, traduttore, linker, debugger) che consentono la scrittura, la verifica e l'esecuzione di *nuovi programmi* per applicazioni specifiche.

- Il risultato dell'attività di sviluppo di un programma è il *file eseguibile*: un file binario che viene *caricato* dall'hard disk in *memoria* RAM e svolge i compiti per cui è stato progettato.

Sviluppo di un'applicazione eseguibile

- L'applicazione in esecuzione dispone delle seguenti aree di memoria:
  - *Area codice*: contiene il codice (come microistruzioni di CPU) del programma.
  - *Area dati*: contiene le variabili globali e statiche.
  - *Area heap*: disponibile per allocazioni dinamiche.
  - *Area stack*: contiene i *record di attivazione* delle funzioni (tutti i dati necessari alla chiamata, esecuzione e ritorno di una funzione).



Sviluppo di un'applicazione eseguibile

- Un file eseguibile è specifico per una particolare *architettura hardware e software*, pertanto può risultare non utilizzabile per due ragioni: (1) la CPU ha un set di microistruzioni differenti, (2) il sistema operativo fornisce chiamate a sistema differenti (anche con la stessa CPU).

- I moduli oggetto (moduli rilocabili cioè con riferimenti relativi) di un progetto devono contenere una sola funzione `main()` che richiama tutte le altre funzioni (se una funzione è stata richiamata ma non collegata, il linker genera un errore).

Ambienti di sviluppo

- In ambiente Windows, Microsoft Visual C++ permette di creare un *Empty Project* di tipo *Win32 Console Application* e quindi di implementare un'applicazione in C++ "unmanaged".
- Esistono anche altri ambienti di sviluppo (per esempio, Eclipse, Dev-C++, Code::Bloks) e compilatori (per esempio, g++)
- Il risultato che si ottiene è del tutto simile a quello visto per il C.
- Si possono scaricare dal sito della Microsoft le versioni *express* di .NET :

http://www.microsoft.com/exPress/.

C e C++

- La sintassi del C++ è compatibile con quella del C, vi sono tuttavia delle differenze anche nella parte comune.
- Per poter sviluppare un programma è necessario fare input/output di dati: in C si usano le funzioni `scanf()` e `printf()`, mentre in C++ si usano gli oggetti `cin` e `cout`.
- Tuttavia questa è solo una prima differenza, ma non è la differenza che caratterizza i due linguaggi.

I/O dati

- Per poter fare I/O è necessario includere il file *header* della libreria standard `iostream`.
- I nomi definiti nella *libreria standard C++* sono dichiarati in un `namespace` chiamato `std` e non sono visibili nel file sorgente a meno che non li si renda espliciti. Per fare ciò si deve usare una *direttiva di uso* (`using namespace std;`).
- Lo standard input è legato all'oggetto predefinito `cin`, lo standard output all'oggetto predefinito `cout`.

I/O dati

- L'operatore di output `<<` è usato per dirigere un valore sullo standard output. Oppure sullo standard error `cerr`.
- L'operatore di input `>>` è usato per leggere un valore dallo standard input.

- Vediamo un programma che legge da tastiera un intero e lo visualizza a monitor. *Il codice sorgente è salvato in un file con estensione .cpp.*

I/O dati

```
/*primo_esempio.cpp
Il primo programma in C++*/

#include <iostream>
using namespace std;

int main(){
   cout<<"Inserire un valore intero: ";

   int i;
   cin>>i;

   cout<<"Il valore inserito e` ";
   cout<<i<<"\n";

   return 0;
}
```

Una possibile uscita

```
Inserire un valore intero: 23
Il valore inserito e` 23
```

Direttiva di uso

Non si usano puntatori

Non c'è formattazione

C++

- I file *header* sono inseriti senza estensione (si usa `" "` per i propri file header).
- La funzione speciale `main()` deve tornare un intero.
- Le variabili sono dichiarate dove servono e non solo all'inizio di un blocco.
- Si possono concatenare variabili e stringhe per stampare frasi complete.
- Per commentare una singola riga si usa `//`.

I/O dati

Una possibile uscita

```
Inserire un valore float: 2.5
Il quadrato di 2.5 vale 6.25
```

```
//esempio2.cpp
#include <iostream>
using namespace std;

int main(){
   cout<<"Inserire un valore float: ";

   float x;
   cin>>x;

   cout<<"Il quadrato di "<<x<<" vale "<<x*x<<endl;
   return 0;
}
```

Manipolatore

Istruzioni di controllo

- Le istruzioni per il controllo del flusso di esecuzione di un programma sono le stesse di quelle viste in C.
- Pertanto si mostrano solo attraverso degli esempi concreti.
  - Esempio3: stampare a monitor i numeri pari compresi tra 0 e 7.
  - Esempio4: stampare a monitor le vocali minuscole comprese all'interno di una sequenza di caratteri letta da tastiera e terminata da un '.' o un '!', altrimenti stampare il carattere -.

Esempio

Una possibile uscita

```
0 2 4 6
```

```
//esempio3.cpp
#include <iostream>
using namespace std;

int main(){
   int a=0, b=7;

   for(int i=a; i<=b;i++)
       if (i%2==0)
             cout<<i<<" ";

   //cout<<i; //errore compile-time
   cout<<endl;
   return 0;
}
```

La variabile contatore è dichiarata nel ciclo e la sua visibilità è il ciclo

Esempio

Una possibile uscita

```
bvcaareytruiutgv
---aa-e---uiu---
```

```
//esempio4.cpp
#include <iostream>
using namespace std;

int main(){
   char c;
   cin>>c;
   while(c!='.'&&c!='!'){
       if (c=='a'||c=='e'||c=='i'||c=='o'||c=='u')
             cout<<c;
       else
             cout<<'-';
       cin>>c;
   }
   cout<<endl;
   return 0;
}
```

Funzioni

- Se si dovesse riusare più volte lo stesso gruppo di istruzioni, allora si deve definire una funzione che contiene tale gruppo di istruzioni e chiamare la funzione dove serve.
- Per esempio, si vuole stampare a monitor i numeri pari compresi tra due estremi passati da tastiera.

- Si può trasformare l'esempio3 in una *funzione che non torna niente* e ha due argomenti interi.

Esempio: funzione

```
//esempio5.cpp
#include <iostream>
using namespace std;

void pari(int, int);

int main(){
   int a1,a2;

   cin>>a1>>a2;
   pari(a1,a2);

   cin>>a1>>a2;
   pari(a1,a2);
   return 0;
}

void pari(int a, int b){
   for(int i=a; i<=b;i++)
        if (i%2==0)
                cout<<i<<" ";
   cout<<endl;
}
```

Prototipo o firma della funzione

Una possibile uscita

```
0 10
0 2 4 6 8 10
271 283
272 274 276 278 280 282
```


Funzioni: passaggio per valore

- Il passaggio dei dati tra la funzione chiamante (`main`) e la funzione chiamata (`pari`) avviene per valore.
- Il passaggio per valore consiste nel copiare i valori delle variabili della funzione chiamante (`a1` e `a2`) nelle variabili argomento della funzione chiamata (`a` e `b`).
- Le variabili di una funzione (compresi gli argomenti) sono visibili solo nel corpo della funzione stessa. Pertanto le variabili della funzione chiamante non sono visibili nella funzione chiamata.

Esempio: visibilità

```
//esempio6.cpp
#include <iostream>
using namespace std;
int somma(int, int);

int main(){
   int a1,a2,res;
   cout<<"Inserire due valori interi: ";
   cin>>a1>>a2;
   res=somma(a1,a2);

   //cout<<tmp;
   cout<<a1<<"+"<<a2<<" vale "<<res<<"\n";
   return 0;
}
int somma(int a, int b){
   int tmp;
   //int a, b;
   //tmp=a1+a2;
   tmp=a+b;
   return tmp;
}
```

Una possibile uscita

```
Inserire due valori interi: 3 5
3+5 vale 8
```

Errore compile-time

Errore compile-time

Errore compile-time

Puntatori

- Una variabile è caratterizzata dal *valore* che contiene e dall'*indirizzo* in cui tale valore è memorizzato.

```
double d =123.456;
```

- L'operatore *indirizzo di*, `&`, restituisce l'indirizzo della variabile a cui è applicato.
- Tale *indirizzo* può essere memorizzato in una variabile puntatore, il cui valore, pertanto, è un indirizzo.

```
double *pd = &d;
```


Puntatori

- Per accedere al valore effettivo cui la variabile puntatore si riferisce si deve *dereferenziare*, `*`, tale variabile puntatore.

```
cout<<*pd;
```

- La variabile puntatore contiene un valore che è l'indirizzo di un'altra variabile, quindi si può accedere indirettamente, cioè attraverso l'indirizzo, alla variabile cui si fa riferimento.
- Una variabile puntatore prima di essere utilizzata deve essere inizializzata con un indirizzo valido o di una variabile già dichiarata o di un'allocazione.

Puntatori

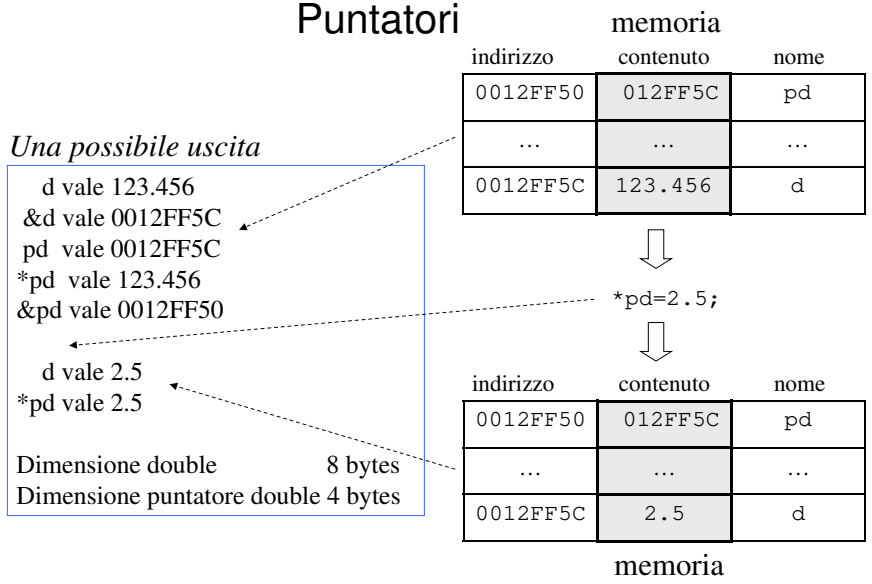
```
#include <iostream>
using namespace std;
int main(){
   double d=123.456;
   double *pd =&d;
   cout<<"  d vale "<< d <<endl;
   cout<<" &d vale "<< &d <<endl;
   cout<<" pd vale "<< pd <<endl;
   cout<<"*pd vale "<< *pd <<endl;
   cout<<"&pd vale "<< &pd <<endl<<endl;

   *pd=2.5;

   cout<<"  d vale "<< d <<endl;
   cout<<"*pd vale "<< *pd <<endl<<endl;

   cout<<"Dimensione double           "<< sizeof(double) <<
        " bytes"<<endl;
   cout<<"Dimensione puntatore double "<< sizeof(double *)<<
        " bytes"<<endl;
   return 0;
}
```


Puntatori



Una possibile uscita

```
  d vale 123.456
 &d vale 0012FF5C
 pd  vale 0012FF5C
*pd  vale 123.456
&pd vale 0012FF50

  d vale 2.5
*pd vale 2.5

Dimensione double           8 bytes
Dimensione puntatore double 4 bytes
```

Puntatori e funzioni

- I puntatori sono utilizzati per accedere ad una variabile in modo indiretto, cioè attraverso l'indirizzo e non il nome simbolico.
- I *puntatori come argomenti di funzione* permettono di modificare i valori delle variabili della funzione chiamante (esterne alla funzione chiamata): utilizzando l'indirizzo di tali variabili e non il nome che non è visibile.

- Esempio: scrivere una funzione che modifichi una variabile del `main`.

Puntatori e funzioni

```
#include <iostream>
using namespace std;
void set(double );
void setp(double *);

int main(){
   double d=123.456;
   cout<<"main: &d vale "<< &d <<endl;
   set(d);
   cout<<"main: d vale "<< d <<endl;
   setp(&d);
   cout<<"main: d vale "<< d <<endl;
   return 0;
}
void set(double x){
   cout<<"  set: &x vale "<< &x <<endl;
   x=0;
}
void setp(double *y){
   cout<<"  setp: y vale "<< y <<endl;
   *y=0;
}
```

Passaggio per indirizzo o *riferimento*

Una possibile uscita

```
main: &d vale 0012FF5C
  set:  &x vale 0012FE84
main:  d vale 123.456
  setp:  y vale 0012FF5C
main:  d vale 0
```

Modifica il valore locale di `x` (non quello di `d`)

Modifica il valore di `d` attraverso l'indirizzo

Puntatori e funzioni

```
#include <iostream>
using namespace std;
void Leggi(int *a, int *b);

int main(){
   int v,w;
   Leggi(&v,&w);
   cout<<"Valori letti: "<<v<< " e "<<w<<endl;

   //int *r,*t;
   //Leggi(r,t);
    return 0;
}
void Leggi(int *a, int *b){
   cout<<"Inserire due valori interi: ";
   cin>>*a>>*b;
}
```

Scrivere una funzione che legge due interi da tastiera e li ritorna al `main`.

Errore run-time: l'applicazione non funziona. Il compilatore segnala un *warning*.

Una possibile uscita

```
Inserire due valori interi: 2 -5
Valori letti: 2 e -5
```


Qualificatore `const`

- Non è opportuno inserire valori costanti all'interno dei programmi. Per esempio, il seguente frammento di codice `for (int i=0; i<512; i++)` presenta due problemi:
  - Leggibilità. Il significato del numero 512 (*magic number*) non è evidente nel contesto del suo uso.
  - Manutenibilità. Non è semplice sostituire tale tipo di valore in codici molto lunghi.
- Utilizzare il preprocessore (`#define`) non è opportuno perché non esegue un controllo sui tipi.

Qualificatore `const`

- La soluzione è utilizzare un oggetto inizializzato ad un valore ed renderlo non modificabile:

```
const int bufsize = 512;
```

- In tal modo si è *localizzato* il valore e ogni tentavo di modifica genera un errore a tempo di compilazione.

```
bufsize = 33; //errore: compile-time
```

- Poiché una costante non può essere modificata dopo la definizione, deve essere inizializzata.

```
const int dim; //errore: compile-time
```


Array

- Gli array sono una *collezione* (insieme) di valori dello stesso tipo caratterizzati da un nome e gestibili attraverso un indice.
- La dimensione dell'array è un valore *costante* non modificabile durante l'esecuzione del programma.
- Per esempio, si dichiara un vettore di 5 elementi di interi con `int vt[5];` e si modifica il secondo elemento con `vt[1]=100;`.
- Un array di N elementi ha valori di indice compresi nell'intervallo [0, N-1].

Array: esempio

Scrivere un programma che memorizza i quadrati dei primi 5 interi e li stampa.

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

int main(){
  const int dim=5;
  int vt[dim];

  for(int i=0;i<dim;i++)
      vt[i]=i*i;

  for(int i=0;i<dim;i++)
      cout<<vt[i]<<" ";
  cout<<endl;
  return 0;
}
```

Una possibile uscita

```
0 1 4 9 16
```


Array e puntatori

- Vi è uno stretto legame tra gli array e puntatori: il nome dell'array è l'indirizzo del suo primo elemento.
  - Quando si incrementa di una unità il puntatore, l'indirizzo del successivo elemento è incrementato del *corretto numero di byte*, in relazione al tipo di dato dell'array. Questa è la base *dell'aritmetica dei puntatori*.
  - Quando si passa un array ad una funzione viene passato solo il suo indirizzo: (1) non si copiano i dati (*efficiente*) e (2) si può modificare l'array della funzione chiamante.

Esempio: array e puntatori

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
   const int dim=5;
   double a[dim];
   for(int i=0;i<dim;i++)
       a[i]=pow(1.23,i);

   double *pa=a;

   for(int i=0;i<dim;i++)
       cout<< a[i] <<" ";
   cout<<endl;
   for(int i=0;i<dim;i++)
       cout<< *(pa+i) <<" ";
   return 0;
}
```

Libreria C

Una possibile uscita

```
1 1.23 1.5129 1.86087 2.28887
1 1.23 1.5129 1.86087 2.28887
```

Il nome dell'array è l'indirizzo del suo primo elemento

Aritmetica dei puntatori

Esempio: array e funzioni

```
#include <iostream>
using namespace std;
void Leggi(float *, int);
void Stampa(float *, int);
int main(){
   const int dim=5;
   float a[dim];

   Leggi(a,dim);
   Stampa(a,dim);
   return 0;
}
```

Una possibile uscita

```
Inserire 5 float: 1 -2.2 3.4 0 6.14
Valori nell'array:
1 -2.2 3.4 0 6.14
```

```
void Leggi(float *vt, int d){
   cout<<"Inserire "<<d<<" float:";
   for(int i=0;i<d;i++)
       cin>>vt[i];
}

void Stampa(float *vt, int d){
   cout<<"Valori nell'array: \n";
   for(int i=0;i<d;i++)
       cout<<vt[i]<<" ";
   cout<<endl;
}
```

Modifica i valori dell'array `a` del `main` attraverso l'indirizzo contenuto in `vt`

Allocazione dinamica

- Per sviluppare programmi "reali" è necessario acquisire e utilizzare la memoria durante l'esecuzione del programma.
- Fino ad ora abbiamo visto *allocazioni statiche*, memoria allocata dal compilatore, ora vediamo le *allocazioni dinamiche*, memoria allocata durante l'esecuzione.
- Vediamo le differenze fondamentali tra i due tipi di allocazione.

Allocazione dinamica

- Gli oggetti statici sono variabili (aree di memoria) dotate di nome che vengono manipolate direttamente, mentre gli oggetti dinamici sono variabili (aree di memoria) prive di nome che vengono manipolate indirettamente attraverso i puntatori.
- L'allocazione e deallocazione di oggetti statici è gestita automaticamente dal compilatore, l'allocazione e deallocazione di oggetti dinamici deve essere gestita esplicitamente dal *programmatore*.

Allocazione dinamica

- In C l'allocazione dinamica è gestita dalle funzioni `malloc()` e `free()`, in C++ l'allocazione dinamica è integrata nel linguaggio con le keyword `new` e `delete`.

- Si alloca un singolo oggetto di un tipo specificato:
```
int *pi = new int(123);
```
- Si alloca un array di elementi di tipo e dimensione specificati:
```
int *pia = new int[4];
```

- L'espressione `new` restituisce un puntatore all'oggetto appena allocato.

Allocazione dinamica

- Una volta terminato di usare l'oggetto allocato dinamicamente, occorre deallocare esplicitamante la memoria usando l'espressione `delete` (deve essere applicata solo a puntatori che si riferiscono a memoria allocata con `new`). Altrimenti si ha una perdita di memoria (*memory leak*).

- Elimina un singolo oggetto:
```
delete pi;
```
- Elimina un array:
```
delete [] pia
```


Esempio: allocazione dinamica

Scrivere un programma che memorizza `dim` valori inseriti da tastiera, anche `dim` letto da tastiera

```
#include <iostream>
using namespace std;
int main(){
   int dim;
   int *vt;
   cout<<"Inserire la dimensione: ";
   cin>>dim;

   vt = new int[dim];
   cout<<"Inserire i valori: ";
   for (int i=0;i<dim;i++)
       cin >> vt[i];
   cout<<"Valori memorizzati: ";
   for (int i=0;i<dim;i++)
       cout<< vt[i]<<" ";

   delete [] vt;
   return 0;}
```

Alloca dinamicamente la memoria per un vettore

Una possibile uscita

```
Inserire la dimensione: 3
Inserire i valori: 2 -3 7
Valori memorizzati: 2 -3 7
```

Rilascia la memoria

Allocazione dinamica

- Qundo si usa l'espressione `delete` si deve tenere in considerazione che:
  - Il puntatore e il suo contenuto non sono modificati da `delete`, quindi punta a memoria non valida (*dangling pointer*). Consigliabile impostare a 0 il puntatore immediatamente dopo l'eliminazione dell'oggetto.
  - Applicare due volte `delete` alla stessa locazione di memoria genera un errore run-time.

Allocazione dinamica

- Quindi:
  - L'operatore `new` alloca la memoria dinamica nello *heap*.
  - La memoria allocata con `new` deve essere liberata con `delete`.

- Mettiamo in evidenza la memoria occupata e i relativi indirizzi con un esempio: scrivere un programma che memorizza i primi *dim* quadrati, con *dim* letto da tastiera.

Esempio: allocazione dinamica

```
#include <iostream>
using namespace std;
int main(){
   int dim, *vt;
   cout<<"Inserire la dimensione: ";
   cin>>dim;
   vt = new int[dim];
   for (int i=0;i<dim;i++)
        vt[i]=i*i;

   for (int i=0;i<dim;i++){
        cout<<"vt["<<i<<"] vale "<< vt[i]<<";\t";
        cout<<"&vt["<<i<<"] vale "<< &vt[i]<<endl;
   }
   cout<<"vt vale "<< vt<<endl;
   cout<<"&vt vale "<< &vt<<endl;

   delete [] vt;
   return 0;
}
```


Esempio: allocazione dinamica

Una possibile uscita

```
Inserire la dimensione: 5
vt[0] vale 0;   &vt[0] vale 00366498
vt[1] vale 1;   &vt[1] vale 0036649C
vt[2] vale 4;   &vt[2] vale 003664A0
vt[3] vale 9;   &vt[3] vale 003664A4
vt[4] vale 16;  &vt[4] vale 003664A8
vt vale 00366498
```

&vt vale 0012FF54

memoria

indirizzo	contenuto	nome
0012FF54	00366498	vt
…	…	…
00366498	0	vt[0]
0036649C	1	vt[1]
003664A0	4	vt[2]
003664A4	9	vt[3]
003664A8	16	vt[4]

…

Generazione di numeri pseudo-casuali

- Per poter verificare i programmi sviluppati con un grande numero di ingressi è utile poter generare *valori pseudo-casuali*.
- La funzione `rand()` contenuta nella libreria `cstdlib` produce un valore intero pseudo-casuale nell'intervallo [0, RAND_MAX]. RAND_MAX e un `define` presente in `cstdlib`.
- Viene generata una sequenza di numeri pseudo-casuali identica se viene usato lo stesso seme per l'inizializzazione (utilizzare la funzione `srand()`).

Generazione di numeri pseudo-casuali

Scrivere un programma che calcola e stampa sequenze di `dim` numeri pseudo-casuali interi con seme `seed`. Il programma termina quando `dim` <=0.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   int dim, seed;
   cout<<"RAND_MAX vale "<<RAND_MAX<<endl;

   do{
        cout<<"Inserire dim e seed: ";
        cin>>dim>>seed;
        srand(seed);

        for(int i=0;i<dim;i++)
               cout<<rand()<<" ";
        cout<<endl;
   }while(dim>0);
   return 0;
}
```

Una possibile uscita

```
RAND_MAX vale 32767
Inserire dim e seed: 4 1
41 18467 6334 26500
Inserire dim e seed: 6 1
41 18467 6334 26500 19169 15724
Inserire dim e seed: 6 2
45 29216 24198 17795 29484 19650
Inserire dim e seed: 0 0
```


Generazione di numeri pseudo-casuali

Scrivere un programma che calcola e stampa numeri pseudo-casuali reali compresi nell'intervallo [a,b].

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   int dim=7, seed=2;
   float a=3, b=-1;

   srand(seed);
   for(int i=0;i<dim;i++)
       cout<<((float)rand()/RAND_MAX)*(b-a)+a<<" ";
   cout<<endl;

   return 0;
}
```

Il cast è necessario

Valori compresi in [0,1]

Una possibile uscita

```
2.99451 -0.566515 0.0460524 0.827693 -0.599231 0.601245 1.21894
```


Progetto

- Per sviluppare programmi più complessi è buona norma dividere il programma su più file:
  - un file *.cpp* deve contenere il `main()` e le funzioni sviluppate per verificare l'applicazione;
  - Altri file *.cpp* contengono le funzioni che costituiscono l'applicazione.
  - I file *header* contengono le dichiarazioni delle funzioni dell'applicazione (e non quelle di verifica).

File header

- In generale, le dichiarazioni di funzione sono poste in un *file header* (con altre risorse comuni). Le definizioni di tali funzioni sono poste in un *file sorgente* con lo stesso nome del file header.

- A causa della possibilità di annidare i file header, può accadere che uno stesso file header sia incluso più volte nello stesso file sorgente.
- Le *direttive condizionali* ci proteggono dall'elaborazione multipla di un file.

File header

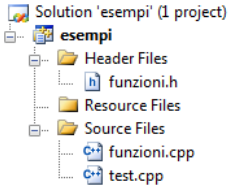
- Ad esempio:

```
#ifndef    BOOKSTORE_H
#define    BOOKSTORE_H
      //contenuto del file bookstore.h
#endif
```

- La direttiva condizionale `#ifndef` controlla se `BOOKSTORE_H` è stato definito in precedenza. Se non è stato definito, la direttiva risulta vera e tutte le righe comprese tra `#ifndef` e `#endif` sono incluse ed elaborate. Al contrario se la direttiva è falsa tutte le righe sono ignorate.

Esempio di progetto

- Il `main()` è inserito nel file *test.cpp*.
- Le funzioni sono dichiarate in *funzioni.h* e definite in *funzioni.cpp*.



Esempio di progetto

funzioni.h

```
#ifndef    FUNZIONI_H
#define    FUNZIONI_H

void Leggi(float *, int);
void Stampa(float *, int);

#endif
```


Esempio di progetto

funzioni.cpp

```
#include <iostream>
#include "funzioni.h"
using namespace std;
void Leggi(float *vt, int d){
       cout<<"Inserire "<<d<<" float:";
       for(int i=0;i<d;i++)
              cin>>vt[i];
}
void Stampa(float *vt, int d){
       cout<<"Valori nell'array: \n";
       for(int i=0;i<d;i++)
              cout<<vt[i]<<" ";
       cout<<endl;
}
```

Esempio di progetto

test.cpp

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

using namespace std;
int main(){
       const int dim=5;
       float a[dim];
       cout<<"Esempio di progetto\n";

       Leggi(a,dim);
       Stampa(a,dim);
       return 0;
}
```