Corso di Laurea in Ingegneria Elettronica

Tecniche avanzate di progettazione software 1

I semestre – I anno Laurea Specialistica

a.a. 2008/2009

Riferimenti bibliografici

- Materiale distribuito a lezione.
- 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.

Prof.	Fabio Solari	Ing.	Manuela Chessa
Tel.:	010-3532059	Tel.:	010-3532289
E-mail:	fabio.solari@unige.it	E-mail:	manuela.chessa@unige.it

URL: http://www.pspc.dibe.unige.it/~taps1/

Obiettivi Formativi
Il corso ha l'obiettivo di fornire agli studenti di Ingegneria Elettronica le metodologie e gli strumenti per lo sviluppo di sistemi software, secondo criteri di elevata efficienza e produttività. In particolare, si utilizza il linguaggio di programmazione orientato agli oggetti C++.

Argomenti trattati
- Richiami di programmazione in C e C# [2 ore].
- Il linguaggio C++ ed il suo uso come linguaggio multiparadigma: principalmente orientato agli oggetti, ma anche con supporto per la programmazione procedurale e basata sui tipi di dato astratto. Tale linguaggio permette di fornire la soluzione più adatta al problema affrontato. Casi di studio. Librerie grafiche e MFC. [32 ore].
- Esercitazioni [24 ore].

Capacità Operative
In generale, conoscere la sintassi ed i principali costrutti del linguaggio C++ e sviluppare programmi ad oggetti utilizzando le caratteristiche di tale linguaggio. In particolare, saper affrontare la progettazione e la programmazione di sistemi software.

Forme didattiche
Lezioni ed esercitazioni di laboratorio.

Tipologia dell'esame
Valutazione delle esercitazioni e prova orale.
In particolare, le modalità dell'esame sono**:**
- Svolgere le esercitazioni.
- Per ogni esercitazione redigere una breve relazione: descrizione delle scelte di implementazione e delle prove effettuate per verificare la correttezza dell'applicazione.
- Inviare per e_mail (o consegnare) i codici sorgente (comprensivi delle prove effettuate) e le relazioni almeno 10 giorni prima della prova orale.
- La prova orale consiste in una discussione sugli argomenti sviluppati nelle esercitazioni e sugli argomenti presentati a lezione.

Nota:
I testi delle esercitazioni saranno distribuiti a lezione e resi disponibili sul sito web (http://pspc.dibe.unige.it/~taps1/). Allo stesso indirizzo saranno disponibili le copie aggiornate e complete dei lucidi presentati a lezione.

SOMMARIO

- Ambienti di sviluppo.
- I/O dati:
  - `iostream`.
  - `namespace`.
- Puntatori: di dati primitivi e loro uso con funzioni.
- Array: 1-D di dati primitivi.
- Allocazione dinamica: `new` e `delete`.
- Eseguibili ed aree di memoria.

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*.
- 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.

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 "<< i <<" e` stato inserito"<<endl;
}
```

Direttiva di uso

Manipolatore

AMBIENTI DI SVILUPPO

- Vediamo l'uso di un compilatore alla riga di comando in Linux:

```
~/PgmCpp>   vi  primo_esempio.cpp
~/PgmCpp>   g++  primo_esempio.cpp
~/PgmCpp >  a.out
Inserire un valore intero: 24
Il valore 24 e` stato inserito.
~/PgmCpp >
```


AMBIENTI DI SVILUPPO

- In ambiente Windows, .NET permette di creare un *Empty Project* di tipo *Win32 Console Application* e quindi di implementare un'applicazione in C++ "unmanaged".

- Il risultato che si ottiene è del tutto simile a quello visto precedentemente.

I/O DATI

- La libreria standard fornisce molte classi che estendono i tipi predefiniti: per esempio fornisce il tipo stringa:

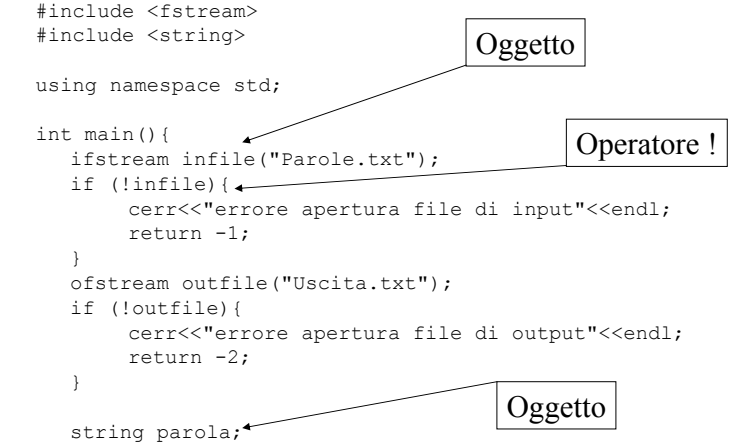
  `string var = "Inizio";`

  `var` è un *oggetto* `string` inizializzato.

- La libreria `iostream` supporta l'I/O su file. Si utilizzano gli stessi operatori visti per lo standard input e output.

I/O DATI

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(){
   ifstream infile("Parole.txt");
   if (!infile){
        cerr<<"errore apertura file di input"<<endl;
        return -1;
   }
   ofstream outfile("Uscita.txt");
   if (!outfile){
        cerr<<"errore apertura file di output"<<endl;
        return -2;
   }

   string parola;
   while(infile >> parola)
        outfile << parola<<endl;
}
```



I/O DATI

- Un esempio di file di input e output:

`Parole.txt`

La libreria standard fornisce molte classi

`Uscita.txt`

La
libreria
standard
fornisce
molte
classi

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* è memorizzato in una variabile puntatore, il cui valore, pertanto, è un indirizzo.

```
double *pd = &i;
```


PUNTATORI

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

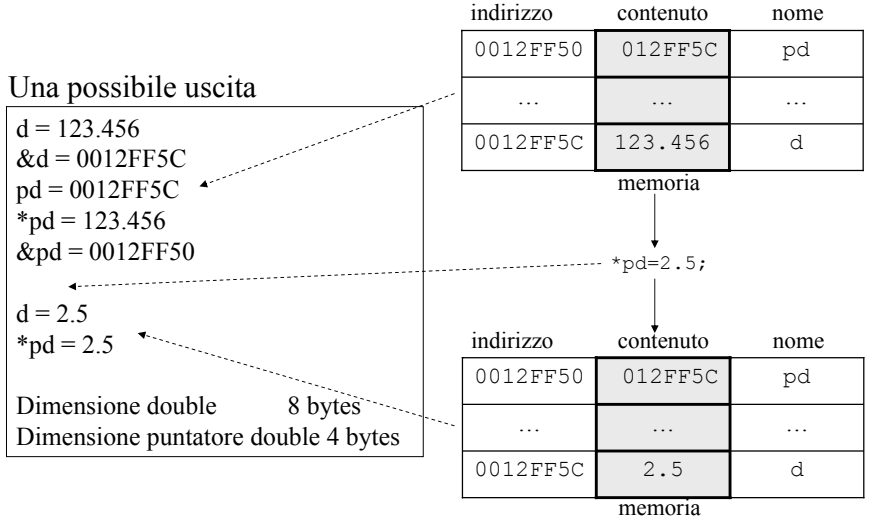
```
cout<<*pd;
```

- 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.

PUNTATORI

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

int main(){
   double d=123.456;
   double *pd =&d;

   cout<<"d = "<< d <<endl;
   cout<<"&d = "<< &d <<endl;
   cout<<"pd = "<< pd <<endl;
   cout<<"*pd = "<< *pd <<endl;
   cout<<"&pd = "<< &pd <<endl<<endl;
   *pd=2.5;
   cout<<"d = "<< d <<endl;
   cout<<"*pd = "<< *pd <<endl<<endl;

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

PUNTATORI



Una possibile uscita

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

d = 2.5
*pd = 2.5

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


PUNTATORI

- Passare i puntatori come argomenti di funzioni permette di modificare i valori delle variabili del chiamante (esterne alla funzione): utilizzando l'indirizzo di tali variabili.

- Si possono avere puntatori anche a variabili strutturate.

PUNTATORI

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

void set(double x);
void setp(double *y);

int main(){
   double d=123.456;
   cout<<"-main: &d = "<< &d <<endl;
   set(d);
   cout<<"-main: d = "<< d <<endl;
   setp(&d);
   cout<<"-main: d = "<< d <<endl;
}
void set(double x){
   cout<<"-set: &x = "<< &x <<endl;
   x=0;}

void setp(double *y){
   cout<<"-setp: y = "<< y <<endl;
   *y=0;}
```

Una possibile uscita

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


PUNTATORI

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

struct Persona{
   int id;
   string nome;
};

void leggi(Persona *x);

int main(){
   Persona p;
   leggi(&p);
   cout<<"Nome: "<< p.nome <<endl;
   cout<<"Matricola: "<< p.id <<endl;
}
void leggi(Persona *x){
   cout<<"Inserire nome: ";
   cin>>x->nome;
   cout<<"Inserire matricola: ";
   cin>>x->id;}
```

Una possibile uscita

```
Inserire nome: Mario
Inserire matricola: 123
Nome: Mario
Matricola: 123
```

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.

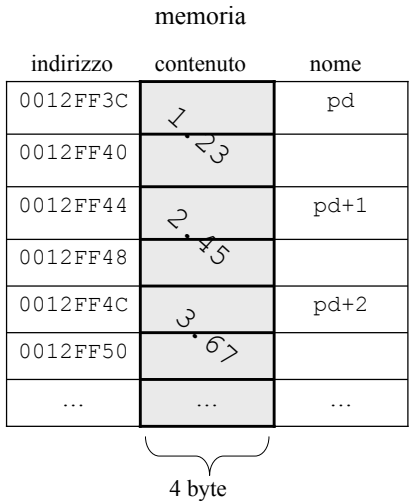
ARRAY E PUNTATORI

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

int main(){
   double a[]={1.23,2.45,3.67};
   cout<<"a     = "<< a <<endl;
   cout<<"&a[0] = "<< &a[0] <<"; a[0] = "<<a[0]<<endl;
   cout<<"&a[1] = "<< &a[1] <<"; a[1] = "<<a[1]<<endl;
   cout<<"&a[2] = "<< &a[2] <<"; a[2] = "<<a[2]<<endl;

   cout<<"&a[2] - &a[1]= "<< &a[2] -&a[1] <<endl;

   double *pd = a;

   cout<<"pd    = "<< pd <<"; *pd     = "<<*pd<<endl;
   cout<<"pd+1  = "<< pd+1 <<"; *(pd+1) = "<<*(pd+1)<<endl;
   cout<<"pd+2  = "<< pd+2 <<"; *(pd+2) = "<<*(pd+2)<<endl;
}
```


ARRAY E PUNTATORI

Una possibile uscita

```
a    = 0012FF3C
&a[0] = 0012FF3C;    a[0] = 1.23
&a[1] = 0012FF44;    a[1] = 2.45
&a[2] = 0012FF4C;    a[2] = 3.67
&a[2] - &a[1]= 1
pd    = 0012FF3C;     *pd    = 1.23
pd+1  = 0012FF44;    *(pd+1) = 2.45
pd+2  = 0012FF4C;    *(pd+2) = 3.67
```



PUNTATORI E FUNZIONI

Attenzione: non si devono ritornare indirizzi di variabili locali alla funzione, quando si torna al chiamante i dati locali si perdono!

```
#include <iostream>
using namespace std;
double* prova();
int main(){

   double *dp = prova();
   for(int i=0; i<5;i++)
        cout<< dp[i] <<" ";
   cout<<endl;
}
double* prova(){
   double a[]={1,2,3,4,5};
   for(int i=0; i<5;i++)
        cout<< a[i] <<" ";
   cout<<endl;
   return a;
}
```

No

Una possibile uscita

```
1  2  3  4  5
1  1.6976e-314    2.64068e-308    1.92615e-307    2.64195e-308
```

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
```

ALLOCAZIONE DINAMICA

Si possono ritornare indirizzi di memoria allocata dinamicamente, quando si torna al chiamante i dati non si perdono. *Attenzione:* deve essere deciso chi libera le risorse!

```
#include <iostream>
using namespace std;
double* prova();
int main(){

   double *dp = prova();
   for(int i=0; i<5;i++)
        cout<< dp[i] <<" ";
   cout<<endl;
   delete [] dp;

}
double* prova(){
   double *pa=new double[5];
   for(int i=0; i<5;i++){
        pa[i]=i+1;
        cout<< pa[i] <<" ";}
   cout<<endl;
   return pa;
}
```

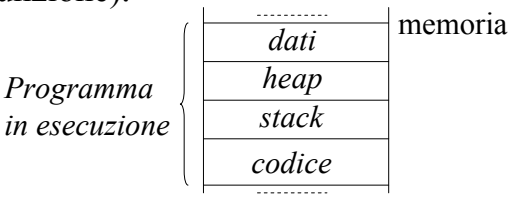
ok

Una possibile uscita

```
1  2  3  4  5
1  2  3  4  5
```


ESEGUIBILI ED 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).



ESEGUIBILI ED AREE DI MEMORIA

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

double g=123.4;

int main(){
   double s=567.8;
   double *pd =new double(901.2);

   cout<<"&g = "<< &g <<endl;
   cout<<"&s = "<< &s <<endl;
   cout<<"pd = "<< pd <<endl<<endl;

   cout<<"&g -&s = "<< &g -&s <<endl;
   cout<<"&g -pd = "<< &g -pd <<endl;
   cout<<"&s -pd = "<< &s -pd <<endl;
}
```

Una possibile uscita

```
&g = 00419008
&s = 0012FF5C
pd = 00353408

&g -&s = 381461
&g -pd = 101248
&s -pd = -280214
```

SOMMARIO

- Puntatori:
  - Allocazione dinamica.
  - Dati strutturati e loro uso con funzioni.
  - Considerazioni su memoria e prestazioni.
- Cast:
  - Conversioni di tipo.
  - Puntatore `void`.
  - Rappresentazione binaria dei tipi.
- Classi : un primo esempio.

ALLOCAZIONE DINAMICA

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

int main(){
   int *frame = 0;

   for (int i=0; i< 100; i++){
       frame = new int[1000*1000];
       //...elaborazione
   }

   delete [] frame;

   //...
}
```

Attenzione: quando si alloca dinamicamente memoria è necessario liberare le risorse prima di riutilizzare lo stesso puntatore.

Informazioni sul processo

PID	USER	SIZE	%CPU	COMMAND
9365	Fabio	389MB	0	myplayer.exe

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.

PUNTATORI E DATI STRUTTURATI

- Dal punto di vista delle *prestazioni*, vi è convenienza ad utilizzare puntatori a variabili strutturate come parametri di funzioni.
- Il dato non deve essere copiato nell'area *stack*, quindi, (1) si occupa meno memoria e (2) si ottiene di conseguenza una maggiore velocità di esecuzione della funzione.

PUNTATORI E DATI STRUTTURATI

```
#include <iostream>
#include <ctime>
using namespace std;
struct Image{
   int data[1000000];
};
void elab(Image img);
void elab_p(Image *img);
int main(){
   Image frame;
   cout<<"main:    "<<sizeof frame<<" bytes"<<endl;
   elab ( frame );
   elab_p ( &frame );
}
void elab(Image img){
   cout<<"-elab:   "<<sizeof img<<" bytes"<<endl;
   for(int i=0; i<1000000; i++)
        img.data[i]=i;
}
void elab_p(Image *img){
   cout<<"-elab_p: "<<sizeof img<<" bytes"<<endl;
   for(int i=0; i<1000000; i++)
        img->data[i]=i;
}
```

Una possibile uscita

main:	4000000 bytes
-elab:	4000000 bytes
-elab_p:	4 bytes

Operatore sizeof

PUNTATORI E DATI STRUTTURATI

```
#include <iostream>
#include <ctime>
using namespace std;
struct Image{
   int data[1000000];
};
void elab(Image img);
void elab_p(Image *img);
int main(){
   Image frame;
   clock_t start,finish;

   start=clock();
   for(int i=0; i<100;i++)
        elab ( frame );
   finish=clock();
   cout<<"Tempo di esecuzione elab   "<<(finish - start)<<" ms"<<endl;

   start=clock();
   for(int i=0; i<100;i++)
        elab_p ( &frame );
   finish=clock();
   cout<<"Tempo di esecuzione elab_p "<<(finish - start)<<" ms"<<endl;
}
```

Libreria C

Una possibile uscita

Tempo di esecuzione elab	781 ms
Tempo di esecuzione elab_p	453 ms

CAST E PUNTATORI VOID

- Attraverso i *cast* (conversione di tipo) si riduce la funzionalità di controllo del linguaggio sui tipi, quindi possono essere fonte di errori.

- I cast espliciti "vecchio stile" assumono la forma
  `(tipo) espressione;`
- Un cast esplicito standard assume la forma
  `nome-cast<tipo>(espressione);`

CAST E PUNTATORI VOID

- Un puntatore a qualsiasi tipo può essere assegnato ad un puntatore di tipo `void *`, detto *puntatore generico*.

- Un puntatore `void *` non può essere dereferenziato direttamente: non vi è nessuna informazione di tipo che possa guidare il compilatore *nell'interpretazione dell'insieme sottostante di bit*.
- Tale puntatore deve essere prima convertito in un puntatore ad un tipo specifico. Non esiste una conversione automatica.

CAST E PUNTATORI VOID

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

int main(){
   int i=123;
   float x=123;

   int *ip = &i;
   float *fp = &x;
   void *vp;

   vp=ip;

   //fp=ip;   //errore: compile-time
   //fp = (float *) vp;   //ok, ma non Standard C++

   fp = static_cast<float *>(vp) ;

   cout<<"Valore = "<<*fp<<endl;
}
```

Una possibile uscita

```
Valore = 1.7236e-043
```

Punta ad un int

RAPPRESENTAZIONE BINARIA

- Si è ottenuto il risultato "1.7236e-043", perché l'*insieme di bit* che rappresentano un tipo `int` di valore 123 sono stati "reinterpretati" come un *insieme di bit* di un tipo `float`, che hanno una diversa rappresentazione.
- La rappresentazione in virgola mobile dei 4 byte (32 bit) che compongono un `float` è composta del bit del segno, dai bit dell'esponente e dai bit della mantissa.

```
S EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM
0 1       8 9                     31
```


RAPPRESENTAZIONE BINARIA

- Vediamo un esempio che stampa la *rappresentazione binaria* dello stesso valore numerico per un tipo `int` e un tipo `float`. Tale esempio usa la seguente funzione che manipola i bit:

```
void printBit(unsigned char val) {
  for(int i = 7; i >= 0; i--)
    if(val & (1 << i))
      std::cout << "1";
    else
      std::cout << "0";
}
```

Operatori bit a bit

RAPPRESENTAZIONE BINARIA

int

float

Una possibile uscita

```
01111011000000000000000000000000
00000000000000001111011001000010
```

```
#include <iostream>
using namespace std;
void printBit(unsigned char val);
int main(){
   int   i=123;
   float x=123;

   unsigned char *bp = reinterpret_cast<unsigned char *>( &i );
   for(int i=0;i<4;i++)
        printBit(bp[i]);
   cout<<endl;

   bp = reinterpret_cast<unsigned char *>( &x );
   for(int i=0;i<4;i++)
        printBit(bp[i]);
   cout<<endl;
}
```

RAPPRESENTAZIONE BINARIA

- Nota: come si può notare il byte meno significativo è quello all'indirizzo più basso, cioè il primo che si incontra, questa è la rappresentazione *little-endian* (utilizzata nei PC).
- Quindi il valore 1025 memorizzato in un intero a 4 byte ha la seguente rappresentazione:

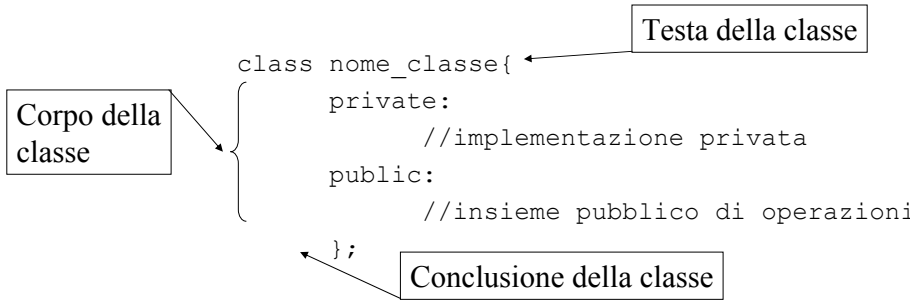
```
00000001 00000100 00000000 00000000   Valori dei bit
01       02       03       04         Indirizzo byte
```

- Se si usasse la rappresentazione *big-endian* si avrebbe:

```
00000000 00000000 00000100 00000001   Valori dei bit
01       02       03       04         Indirizzo byte
```


CLASSI

- Vediamo la forma generale di una *classe*:



- Le operazioni sono chiamate *funzioni membro* o *metodi*.
- I dati sono chiamati *dati membro* o *attributi*.

CLASSI

- I membri pubblici forniscono l'*interfaccia* pubblica della classe (insieme delle operazioni che realizzano il *comportamento* della classe).
- I membri privati rappresentano l'*implementazione* privata, cioè l'insieme dei dati in cui vengono registrate le *informazioni* ed eventuali metodi per la loro manipolazione privata.
- Questa distinzione rispecchia la divisione tra il programmatore della classe e l'eventuale utilizzatore (un altro programmatore).

CLASSI: information hiding

- La divisione tra interfaccia pubblica ed implementazione privata fa riferimento al concetto dell'incapsulamento dei dati (*information hiding*):
  - I programmi che utilizzano la classe non devono essere modificati se viene cambiata (migliorata) l'implementazione privata.
  - I programmi che utilizzano la classe non possono accedere all'implementazione privata, evitando in tal modo di produrre comportamenti indesiderati.

CLASSI: FILE HEADER

- La definizione della classe di solito è posta in un *file header* che ha il nome della classe; le definizioni dei metodi sono poste in un *file sorgente* con lo stesso nome della classe.

- 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.

CLASSI: 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.

CLASSI: ESEMPIO

- Come primo esempio si sviluppa una classe che rappresenta i punti del piano:
  - Sono forniti dei metodi pubblici per modificare lo stato dell'oggetto.
  - Per verificare l'utilizzo del costruttore e del distruttore sono inserite delle stampe a monitor.
- Il test della classe viene fatto utilizzando un terzo file sorgente che contiene il `main()`.

CLASSI: OPERATORE ::

- Poiché le definizioni dei metodi sono al di fuori del corpo della classe, è necessario utilizzare un operatore per individuare a quale classe la funzione membro appartenga.

- Tale compito è eseguito dall'operatore *doppi due punti,* `::`, detto operatore del campo d'azione (*scope resolution*). Si utilizza anteponendo ad esso il nome della classe: `nome_classe::`.

CLASSI: Point.h

```
class Point{
  float x;
  float y;
public:
  Point();
  Point(float, float);
  ~Point();
  void setx(float );
  void sety(float );
  float getx();
  float gety();
};
```

Membri privati

Costruttore (di default): ha lo stesso nome della classe e non può restituire un valore

Overloading di costruttore

Distruttore: ha lo stesso nome della classe preceduto da una tilde, non può restituire un valore né ricevere parametri

CLASSI: Point.cpp

```
#include <iostream>
#include "Point.h"
using namespace std;


Point::Point(){
   x=0; y=0;
   cout<<"Point()"<<endl;
}
Point::Point(float a, float b){
   x=a; y=b;
   cout<<"Point(float, float)"<<endl;
}
Point::~Point(){
   cout<<"~Point()"<<endl;
}
…
```

File header della classe

```
…
void Point::setx(float a){
   x=a;
}
void Point::sety(float b){
   y=b;
}
float Point::getx(){
   return x;
}

float Point::gety(){
   return y;
}
```

Scope resolution

CLASSI: test.cpp

```
#include <iostream>
#include "Point.h"
using namespace std;

int main(){
   Point p1;
   Point p2(3,4.1);

   //p1.x=6.3; //errore: compile-time

   cout<<"p1 "<<p1.getx()<<" " <<p1.gety()<<endl;

   p2.setx(-3.3);
   cout<<"p2 "<<p2.getx()<<" " <<p2.gety()<<endl;

}
```

Una possibile uscita

```
Point()
Point(float, float)
p1 0 0
p2 -3.3 4.1
~Point()
~Point()
```


CLASSI: test1.cpp

```
#include <iostream>
#include "Point.h"
using namespace std;

void print(Point *p);

int main(){
   if(1)
   {
   Point p1(2.3,-4.5);
   print( &p1);
   }
   cout<<"-----------------------"<<endl;
   Point *a = new Point[3];
   cout<<"a[2].getx="<<a[2].getx()<<endl;
   delete  [] a;
}

void print(Point *p){
   cout<<"print(): *p "<<p->getx()<<", " <<p->gety()<<endl;
}
```

Una possibile uscita

```
Point(float, float)
print(): *p 2.3, -4.5
~Point()
-----------------------
Point()
Point()
Point()
a[2].getx=0
~Point()
~Point()
~Point()
```

SOMMARIO

- Costruttori.
  - Lista di inizializzazione dei membri.
- Metodi:
  - Overloading.
  - Argomenti di default.
  - Inline.
- Tipi stringa.

COSTRUTTORI

- I dati membro di una classe sono dichiarati nello stesso modo in cui sono dichiarate le variabili.
- Non è necessario dichiarare separatamente due membri dello stesso tipo.
- Un dato membro può essere di qualsiasi tipo.
- Un dato membro *non* può essere inizializzato esplicitamente nel corpo della classe. A questo scopo è utilizzato il costruttore della classe.

COSTRUTTORI

- Un costruttore ha lo stesso nome della classe e non può restituire un valore, neanche `void`.
- Nel corpo del costruttore sono inizializzati i dati membro.

```
Point::Point(float a, float b){
 x=a; y=b;
}
```

- Una sintassi alternativa è *la lista di inizializzazione dei membri*. Una lista di nomi di attributi e dei loro valori separati da virgole.

COSTRUTTORI

- La lista di inizializzazione dei membri può essere specificata solo nella definizione del costruttore.
- È posta tra la lista dei parametri e il corpo del costruttore ed è preceduta da due punti, **:** .

```
Point::Point(float a, float b): x(a), y(b) {

    }
```

OVERLOADING

- L'*overloading* (sovraccaricamento) delle funzioni e dei metodi consente a più funzioni, che offrono un'operazione comune su tipi di parametri diversi, di condividere un nome comune.
- Se non esistesse l'overloading, ad esempio, si dovrebbe definire un insieme di funzioni `max` con nomi diversi in relazione al tipo dei parametri:
  - `int i_max(int, int);`
  - `int v_max(int *);`
  - `int mat_max(Matrix *);`
- Ma tali funzioni eseguono le *stessa azione generale*: restituire il valore massimo nell'insieme dei valori dei loro parametri.

OVERLOADING

- Dal punto di vista dell'*utente*, esiste una sola operazione, quella di determinare il valore massimo, mentre i dettagli di implementazione sono di scarso interesse.
- Si può dare lo stesso nome a due o più funzioni se le loro *liste di parametri* sono *distinte* per numero o tipo di parametri.

```
void print(string s);
void print(int a);
void print(int a, int b);
```


OVERLOADING

- Il solo tipo di ritorno non basta per distinguere due funzioni.

```
int   val(float x, float y);//errore: compile-
float val(float x, float y);//        time
```

- È necessario fornire una dichiarazione (e definizione) distinta per ogni funzione overloaded.

OVERLOADING

```
#include <iostream>
#include <string>
using namespace std;
int max(int, int);
int max(int *, int);
int main(){
   int ia[]={-1,3,15,-20,1};
   int f=3,g=4;
   cout<<max(f,g)<<endl;
   cout<<max(ia,5)<<endl;
}
int max(int a, int b){
   if (a>b)
        return a;
   else
        return b;
}
int max(int *v, int size){
   int m = v[0];
   for(int i=1;i<size;i++)
        if (v[i]>m) m=v[i];
   return m;}
```

Una possibile uscita

```
4
15
```

OVERLOADING

- Se le funzioni *non* condividono la stessa operazione (anche se operano sullo stesso tipo di dati), allora diversificare i nomi delle funzioni fornisce informazioni che rendono il programma più facile da comprendere.

```
void setDate(Date *, int ,int ,int);
void convertDate(Date *, Date *);
```


ARGOMENTI DI DEFAULT

- Un argomento di default è un *valore* che, sebbene non applicabile in tutte le circostanze, viene giudicato valore appropriato di un parametro nella maggior parte dei casi.

- Una funzione può specificare un argomento di default per uno o più parametri usando la *sintassi di inizializzazione* nella lista dei parametri:

```
int val(int v, int cf=0);
```


ARGOMENTI DI DEFAULT

- Tale funzione può essere chiamata con o senza argomento per tale parametro:

```
val(3);
val(3,3);
```

- I parametri *non inizializzati* devono essere quelli più a *sinistra*. Non si può mescolare parametri inizializzati e parametri non inizializzati:

```
int val2(int v=33, int cf);//errore compile-
                           //         time
```


ARGOMENTI DI DEFAULT

- Gli argomenti di default sono usati soltanto per sostituire gli *ultimi argomenti* mancanti di una chiamata.

- L'argomento di default è specificato nella dichiarazione contenuta in un file header pubblico (che descrive l'interfaccia) e non nella definizione della funzione.

ARGOMENTI DI DEFAULT

Notare la dichiarazione

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

void print(int=1, char='a' );

int main(){

   print();
   print(3);
   print(3,'b');
   //print(,'c');//errore: compile-time

}

void print(int a, char c){
   for(int i=0; i<a;i++)
        cout<<c;
   cout<<endl;
}
```

Una possibile uscita

```
a
aaa
bbb
```


ARGOMENTI DI DEFAULT E OVERLOADING

- In generale quale scelta operare dipende da quale implementazione risulta più chiara:
  - Se il corpo della funzione comprende diverse parti alternative eseguite in relazione al valore dell'argomento di default, allora è più opportuno fare un overloading e lasciare la scelta di quale codice eseguire al compilatore.
  - Se si devono eseguire solo delle inizializzazioni diverse, ma il codice è unico, allora è meglio usare gli argomenti di default.

ARGOMENTI DI DEFAULT E OVERLOADING

- Si deve prestare attenzione a non creare situazioni *ambigue*: un overloading che coincide con una chiamata con argomento di default, in tal caso il compilatore genera un errore.

```
void print(int=1, char='a' );
void print(int );

int main(){
  print(3);//errore: compile-time
}
```


INLINE

- Utilizzare *funzioni* per operazioni piccole, invece del *codice espanso* "nella riga", ha i seguenti vantaggi:
  - Lettura e interpretazione facile.
  - È più semplice cambiare implementazioni localizzate.
  - Possono essere riutilizzate.
  - Permettono il supporto all'*information hiding* (dati privati con accesso attraverso metodi pubblici).

INLINE

- Per esempio, si consideri la seguente funzione:

```
int min(int v1, int v2){
   return (v1<v2 ? v1 : v2);
}
```

- Tuttavia la chiamata di funzione è più *lenta* della valutazione diretta dell'operazione condizionale (vi è la gestione dell'area stack di memoria).
- Le funzioni *inline* forniscono la soluzione: vengono espanse "nella riga", in ogni punto del programma in cui sono chiamate.

INLINE

- Per rendere una funzione *inline* è necessario usare la parola chiave `inline`:

```
inline int min(int v1, int v2){//uguale}
```

- Il meccanismo di inline è in genere pensato per *ottimizzare* funzioni piccole e chiamate molte volte.
- La definizione della funzione inline deve essere visibile, affinché il compilatore sia in grado di espanderla. Pertanto va definita in ogni file in cui è chiamata.

INLINE

- Le definizioni di *funzioni* che sono fornite *nel corpo della classe* sono trattate automaticamente come funzioni inline.

- Le *macro con argomenti* non sono una scelta equivalente:
  - Presentano degli effetti indesiderati (non sono funzioni).
  - Non possono accedere ai membri di una classe.
  - Non eseguono un controllo sui tipi (perché sono gestite dal preprocessore e non dal compilatore).

INLINE E MACRO

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

#define MIN(V1,V2)  (((V1)<(V2)) ? (V1) : (V2))

inline int min(int v1, int v2)
{
       return (v1<v2 ? v1 : v2);
}
int main(){

   int i=3, j=5;

   cout <<min(i,j)<<endl;
   cout <<MIN(i,j)<<endl;
   cout<<"--------"<<endl;
   cout <<min(i++,j++)<<endl;
   i=3; j=5;
   cout <<MIN(i++,j++)<<endl;
}
```

Una possibile uscita

```
3
3
--------
3
4
```

ESEMPIO

- Riscriviamo la classe di esempio Point e utilizziamo lo stesso tipo di test, cioè test1.cpp.

- Si mette in evidenza l'uso della *direttiva condizionale* al fine di evitare errori di inclusione multipla.
- È importante che sia sempre definito un *costruttore di default* (cioè senza argomenti o con argomenti di default per ogni parametro), perché è quello invocato nella creazione degli array.

ESEMPIO: Point.h

Direttiva condizionale

```
#ifndef POINT_H
#define POINT_H

#include <iostream>
using namespace std;
class Point{
   float x;
   float y;
public:
   Point(float a=0, float b=0){
        x=a; y=b;
        cout<<"Point("<<a<<","<<b<<")"<<endl;
   }
   ~Point(){cout<<"~Point()"<<endl;};
   void setx(float a){ x=a;}
   void sety(float b){ y=b;}
   float getx(){return x;}
   float gety(){return y;}
};

#endif
```


ESEMPIO: test1.cpp

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

using namespace std;

void print(Point *p);

int main(){
   if(1)
   {
   Point p1(2.3,-4.5);
   print( &p1);
   }
   cout<<"-----------------------"<<endl;
   Point *a = new Point[3];
   cout<<"a[2].getx="<<a[2].getx()<<endl;
   delete  [] a;
}
void print(Point *p){
   cout<<"print(): *p "<<p->getx()<<", " <<p->gety()<<endl;
}
```

Una possibile uscita

```
Point(2.3,-4.5)
print(): *p 2.3, -4.5
~Point()
-----------------------
Point(0,0)
Point(0,0)
Point(0,0)
a[2].getx=0
~Point()
~Point()
~Point()
```


TIPI STRINGA

- Il C++ fornisce due rappresentazioni delle stringhe: la stringa di caratteri stile C e la classe stringa introdotta nella libreria standard del C++.

<u>Stile C</u>.

- La stringa stile C è memorizzata in un *array di caratteri* ed è in genere manipolata attraverso un puntatore `char *`.
- La libreria standard del C offre una serie di funzioni per manipolare tale tipo di stringhe.

TIPI STRINGA

- In genere si percorre una stringa stile C usando l'aritmetica dei puntatori, fino a quando si raggiunge il *carattere nullo di terminazione.*

```
char str[] = {'p','r','o','v','a','\0'};
char *p=str;
int i=0;
while(*p++)  i++;
cout<<"La stringa "<<str<<" e` lunga "<<i<<" caratteri"<<endl;
```

- Il codice produce il risultato

```
La stringa prova e` lunga 5 caratteri
```


TIPI STRINGA

- Tuttavia a causa della sua rappresentazione a *basso livello* la stringa stile C è usata solo in contesti particolari.
- In genere si usa la classe stringa della libreria standard C++. Per esempio, alcuni comportamenti di tale tipo astratto sono:
  - Inizializzazione con una sequenza di caratteri o un altro *oggetto* stringa.
  - Supporto per la copia (funzione `strcpy()` in C).
  - Supporto per l'accesso in lettura scrittura di singoli caratteri.
  - Supporto per il confronto (funzione `strcmp()` in C).

TIPI STRINGA

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

int main(){
   string st("prova");
   cout<<"La stringa "<<st<<" e` lunga "<<st.size()<<" caratteri"<<endl;
   string s1;
   if (s1.empty()) cout<<"La stringa e` vuota"<<endl;
   string s2(st);
   if (s2==st) cout<<"Le stringhe sono uguali"<<endl;
   s2[0]='P';
   string s3=st+s2;
   cout<<s3<<endl;
   }
```

Una possibile uscita

```
La stringa prova e` lunga 5 caratteri
La stringa e` vuota
Le stringhe sono uguali
provaProva
```


TIPI STRINGA

- Per convertire un oggetto di tipo `string` in una stringa di caratteri stile C è necessario invocare esplicitamente il metodo `c_str()` sull'oggetto.

SOMMARIO

- Tipi stringa: un esempio.
- Qualificatore `const`:
  - Puntatori.
- Tipi riferimento:
  - Parametri formali.
- Tipo `bool`: tipo predefinito.

- Esercitazione: alcune considerazioni.

TIPI STRINGA: un esempio

- Utilizzare un *tipo* di *dato astratto* facilita la scrittura del codice (per es. usare il tipo `string` rispetto al `char *`).

- Tuttavia questo non vuol dire che si ottengano prestazioni inferiori. Confrontiamo due funzioni, semanticamente equivalenti, di cui una fa uso delle stringhe stile C e l'altra del tipo `string`.

TIPI STRINGA: un esempio

```
void stringac(){
   const char *str="Stringa letterale molto lunga: Tecniche\
             Avanzate di Progettazione Software 1";
   int errori=0;

   for(int i=0; i<1000000;i++){
       int len = strlen(str);
       char *str2 = new char[len+1];
       strcpy(str2,str);
       if (strcmp(str2,str))
              errori++;
       delete [] str2;
   }

   cout<<"Stringa stile C: Errori = "<<errori<<endl;
}
```


TIPI STRINGA: un esempio

```
void stringacpp(){
   string str("Stringa letterale molto lunga: Tecniche \
                Avanzate di Progettazione Software 1");
   int errori=0;

   for(int i=0; i<1000000;i++){
       int len = str.size();
       string str2 = str;
       if (str!=str2)
              errori++;
   }

   cout<<"Stringa stile C++: Errori = "<<errori<<endl;
}
```

TIPI STRINGA: un esempio

```
#include <iostream>
#include <string>
#include <ctime>
#include <cstring>
using namespace std;
void stringac();
void stringacpp();
int main(){
   clock_t start,finish;

   start=clock();
   stringac();
   finish=clock();
   cout<<"\tTempo di esecuzione: "<<(finish - start)<<" ms"<<endl;

   start=clock();
   stringacpp();
   finish=clock();
   cout<<"\tTempo di esecuzione: "<<(finish - start)<<" ms"<<endl;
}
```

Una possibile uscita

```
Stringa stile C: Errori = 0
        Tempo di esecuzione: 578 ms
Stringa stile C++: Errori = 0
        Tempo di esecuzione: 484 ms
```


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
```


QUALIFICATORE `const`

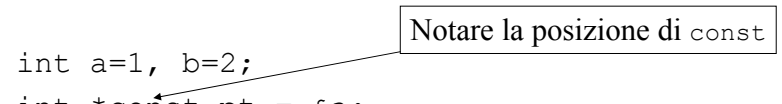
- Per poter utilizzare un puntatore con un oggetto costante, si deve definire un puntatore che punta ad un oggetto costante:

```
//int *pt = & bufsize; //errore:
                              compile-time
const int *pt = & bufsize;

//*pt = 55; //errore: compile-time

int i =0;
pt = &i;
```

Il puntatore non è costante

QUALIFICATORE `const`

- È possibile definire un puntatore costante (esso stesso costante) sia ad un oggetto costante sia ad un oggetto non constante:




```
int a=1, b=2;
int *const pt = &a;
*pt=33; // a vale 33

//pt= &b; // errore: compile-time
```


QUALIFICATORE `const`

- Nei programmi reali si usa molto spesso un puntatore ad un oggetto costante come parametro formale di una funzione, per indicare che l'oggetto effettivo passato alla funzione non è modificato al suo interno:

```
int strcomp(const char *s1, const char *s2)
{
      *s1='a'; //errore: compile-time
…
}
```


QUALIFICATORE `const`

- I dati membro `const` di una classe devono essere sempre inizializzati nella lista di inizializzazione dei membri:

```
class Const{
  //const int size=100;//errore: compile-time
  const int size;
public:
  Const(int i): size(i){
     //size = i; //errore: compile-time
  }
};
```


TIPI RIFERIMENTO

- Un riferimento (*reference*) agisce come un *nome alternativo* per un oggetto: permette la manipolazione indiretta di un oggetto in maniera simile all'uso di un puntatore, ma senza richiedere l'uso della sintassi dei puntatori:


```
double d = 3.3;
double &ref = d;
ref=5; // d vale 5 e ref vale 5
```

- Un riferimento deve essere sempre inizializzato

```
double &r; //errore: compile-time
```

TIPI RIFERIMENTO

- Una volta definito, un riferimento *non* può essere riferito ad un altro oggetto:

```
double x =0;
ref = x; // d vale 0 e ref vale 0
x=123; // d vale 0 e ref vale 0
```

- Tutte le operazioni sui riferimenti sono applicate effettivamente agli oggetti cui si riferiscono, compreso l'operatore *indirizzo-di*:

```
double *p= &ref;
*p=33; // d vale 33 e ref vale 33
```


TIPI RIFERIMENTO

- Le due differenze fondamentali tra puntatori e riferimenti sono: (1) un riferimento deve sempre riferirsi ad un oggetto; (2) l'assegnamento di un riferimento ad un altro cambia il valore dell'oggetto riferito e non il riferimento stesso.

- L'utilizzo più frequente dei riferimenti si ha come *parametri formali* di una funzione: un altro modo per implementare il *passaggio per riferimento* oltre all'uso dei puntatori. Utili per (i) passare come argomenti oggetti molto grandi o (ii) modificare i valori degli argomenti.

TIPI RIFERIMENTO

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

void swap_ptr(int *v1, int *v2){
  int temp=*v2;
  *v2 = *v1;
  *v1 = temp;
}
void swap_ref(int &v1, int &v2){
  int temp=v2;
  v2 = v1;
  v1 = temp;
}
```

riferimenti

Uso semplificato, senza *

TIPI RIFERIMENTO

```
int main(){
  int a=1, b=9;
  swap_ptr( &a, &b);
  cout<<"a="<<a<<"  b="<<b<<endl;

  a=1; b=9;
  swap_ref( a, b);
  cout<<"a="<<a<<"  b="<<b<<endl;
}
```

Chiamata con variabili senza `&`, ma `a` e `b` sono modificate!

Una possibile uscita

```
a=9  b=1
a=9  b=1
```

TIPI RIFERIMENTO

- Prima di dereferenziare un puntatore si dovrebbe verificare che punti effettivamente ad un oggetto, questo non è necessario con un riferimento.
- Se all'interno di una funzione un parametro deve riferirsi a più oggetti o a nessuno allora è necessario usare un puntatore.
- Se si desidera che un parametro riferimento non sia modificato nella funzione, lo si deve dichiarare `const`.

- I parametri riferimento consentono di implementare in modo efficiente gli *operatori sovraccaricati* mantenendone intuitivo l'uso.

TIPO `bool`

- Ad un oggetto di tipo `bool` è possibile assegnare i valori logici `true` e `false`.

- Le variabili di tipo `bool` sono convertire al tipo `int` quando è necessario un valore aritmetico: `false` diventa `0` e `true` diventa `1`.
- Se necessario, un valore zero o un puntatore nullo sono convertiti in `false`, tutti gli altri valori sono convertiti in `true`.

TIPO `bool`

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

int main(){
   bool f = true;
   int i=3;

   if (f){
        i+=f;
        cout<<"i = "<<i<<endl;
   }

   int j = 33;
   if ( j )
        cout <<"Vero"<<endl;
}
```

Una possibile uscita

```
i = 4
Vero
```

SOMMARIO

- Qualificatore `const`:
  - Oggetti.
  - Oggetti con dati membro puntatore.
  - Dati membro `mutable`.
- Il puntatore implicito `this`:
  - Concatenazione delle chiamate ai metodi.
- Parola chiave `friend`:
  - Funzioni
  - Metodi.

OGGETTI E `const`

- Per modificare lo stato di un oggetto è necessario invocare una funzione membro pubblica: per rispettare la caratteristica `const` di un oggetto, è necessario distinguere tra i metodi che possono modificare lo stato dell'oggetto e quelli che non lo modificano.
- Chi progetta la classe indica i metodi che non modificano lo stato come `const`: la parola chiave `const` è inserita tra la lista dei parametri e il corpo della funzione.

OGGETTI E `const`

```
class Point{
  float x;
  float y;
public:
//…
float getx() const {return x;}
float gety() const {return y;}

//void sety(float b) const { y=b;}
//…
};
```

Errore: compile-time

OGGETTI E `const`

```
#include <iostream>
#include "Point.h"
using namespace std;


int main(){

  Point p1;
  p1.setx(33);
  cout<<p1.getx()<<endl;

  const Point p2(-2,5);
  //p2.setx(11); //errore:compile-time
  cout<<p2.getx()<<endl;

}
```

Una possibile uscita

```
Point(0,0)
33
Point(-2,5)
-2
~Point()
~Point()
```

OGGETTI CON PUNTATORI E `const`

- Il qualificatore `const` garantisce che il metodo non modifichi i *dati della classe (stato)*, ma se la classe contiene dei *puntatori*, gli oggetti a cui si riferiscono i puntatori possono essere modificati.
- Questo è un cattivo stile di programmazione e deve essere evitato per non confondere l'utilizzatore della classe.
- Vediamo un esempio in cui si definisce una propria versione di array di float.

OGGETTI CON PUNTATORI E `const`

```
#ifndef MYVECT_H
#define MYVECT_H

class MyVect{
  int size;
  float *data;
public:
  MyVect(int dim, float *v) ;
  ~MyVect() {delete [] data;}
  void Print () const;
  void ReSet (float *v) const ;
};

#endif
```


OGGETTI CON PUNTATORI E `const`

```
#include <iostream>
#include "MyVect.h"
using namespace std;

MyVect::MyVect(int dim, float *v) {
       size=dim;
       data = new float[size];
       for(int i=0;i<size;i++)
              data[i]=v[i];
}
void MyVect::Print() const {
   for(int i=0;i<size;i++)
              cout<<data[i]<<" ";
   cout<<endl;
}
void MyVect::ReSet(float *v) const {
   //data = new …  errore compile-time
   for(int i=0;i<size;i++)
              data[i]=v[i];
}
```


OGGETTI CON PUNTATORI E `const`

```
#include <iostream>
#include "MyVect.h"
using namespace std;

int main(){

   float a[]={1,2,3,4,5};
   const MyVect v(5,a);

   v.Print();

   float a1[]={5,4,3,2,1};
   v.ReSet(a1);
   v.Print();

}
```

Una possibile uscita

```
1 2 3 4 5
5 4 3 2 1
```

DATI MEMBRO `mutable` E `const`

- Per consentire la modifica di un dato membro anche se l'oggetto corrispondente è `const`, si può dichiaralo `mutable` (modificabile): la parola chiave `mutable` deve precedere la dichiarazione nella lista dei membri della classe.

- Modifichiamo l'esempio precedente inserendo un indice che permette lo scorrimento dell'array sottostante: l'indice deve essere modificabile anche quando l'oggetto è `const`.

DATI MEMBRO `mutable` E `const`

NOTA:

- Nell'esempio si fa uso della macro `assert()` fornita nella Libreria Standard del linguaggio C (includere `<cassert>`).

- Tale macro è utile per verificare una condizione: se la condizione risulta falsa, l'asserzione non è valida e viene stampato un messaggio diagnostico e l'esecuzione del programma si interrompe.

DATI MEMBRO `mutable` E `const`

```
#ifndef MYVECT_H
#define MYVECT_H
#include <cassert>
using namespace std;
class MyVect{
   mutable int index;
   int size;
   float *data;
public:
   MyVect(int dim, float *v);
   ~MyVect() {delete [] data;}
   void Print () const;
   void Set (float x) {
        assert(index>=0);
        data[index]=x;}
   float Get () const {
        assert(index>=0);
        return data[index];}
   void Next() const {
        if (++index >=size) index=0;}
};
#endif
```

```
MyVect::MyVect(int dim, float *v)
{
   size=dim;
   data = new float[size];
   for(int i=0;i<size;i++)
        data[i]=v[i];
   index=-1;
}
```


DATI MEMBRO `mutable` E `const`

```
#include <iostream>
#include "MyVect.h"
using namespace std;

int main(){

   float a[]={1,2,3,4,5};
   const MyVect v(5,a);

   //   cout << v.Get() << endl; //errore: run-time


for(int i=0; i<10; i++){
        v.Next();
        cout<<v.Get()<<" ";
   }
   cout<<endl;
}
```

Eventuale messaggio diagnostico

```
Assertion failed: index>=0,
file g:\users\fabio\...\myvect.h, line 19
```

Una possibile uscita

```
1 2 3 4 5 1 2 3 4 5
```

IL PUNTATORE IMPLICITO `this`

- Ogni oggetto mantiene la propria copia dei dati membro della classe.
- Esiste una sola copia di ogni funzione membro di una classe.

- L'associazione del metodo all'oggetto corrente avviene attraverso il puntatore implicito `this`.
- Ogni funzione membro di una classe contiene *un puntatore all'oggetto* tramite il quale il metodo è stato invocato, il puntatore `this`.

IL PUNTATORE IMPLICITO `this`

- Nella definizione di una funzione membro si può fare riferimento esplicitamente al puntatore `this`, anche se in generale *non* è necessario.

- Tuttavia in alcuni casi è necessario ricorrere a tale puntatore: per esempio per definire le funzioni membro di una classe in modo da consentire la *concatenazione delle chiamate* che si riferiscono allo stesso oggetto.

IL PUNTATORE IMPLICITO `this`

- Per esempio sostituire

```
v.Next();
cout<<v.Get()<<" ";
```

dell'esempio precedente con

```
cout<<v.Next().Get()<<" ";
```

- Affinché `v.Get()` sia invocata dopo la chiamata a `v.Next()`, `Next()` deve restituire l'oggetto tramite il quale è stata invocata.
- In particolare deve *restituire un riferimento* all'oggetto.

IL PUNTATORE IMPLICITO `this`

- Il metodo Next() ha la seguente definizione:

```
MyVect&  Next()  {
 if (++index >=size) index=0;
 return *this;
}
```

- Un esempio di uso:

```
float a[]={1,2,3,4,5};
MyVect v(5,a);
for(int i=0; i<10; i++)
  cout<<v.Next().Get()<<" ";
cout<<endl;
for(int i=0; i<5; i++)
  v.Next().Set(i+10);
v.Print();
cout<<endl;
```

Una possibile uscita

```
1 2 3 4 5 1 2 3 4 5
10 11 12 13 14
```

FUNZIONI `friend`

- Il meccanismo delle funzioni `friend` consente ad una classe di accordare ad alcune funzioni l'*accesso* ai suoi *membri non pubblici*.
- La dichiarazione di una funzione `friend` comincia con la parola chiave `friend` e può comparire solo in una definizione di classe.

- Si possono dichiarare `friend` funzioni membro di un'altra classe definita precedentemente o un'intera classe (in tal caso tutti i metodi hanno l'accesso ai membri non pubblici della classe che accorda la qualifica di `friend`).

FUNZIONI `friend`

- In generale, si deve cercare di minimizzare il numero di funzioni che hanno accesso alla rappresentazione interna di una classe.
- L'uso delle dichiarazioni `friend` è comune con gli operatori sovraccaricati.
- Si utilizzano anche se si devono manipolare oggetti di classi distinte o per operazioni di uso generale.

FUNZIONI `friend`

```
class Pixel; // solo la dichiarazione

class Point{
   float x,y;
public:
   Point(float a, float b): x(a),y(b){}
   void Print(){cout<<"x="<<x<<", y="<<y<<endl;}
   friend void MyCopy(Point&, Pixel&);
};

class Pixel{
   int r,c;
public:
   Pixel(int p1, int p2): r(p1),c(p2){}
   void Print(){cout<<"r="<<r<<", c="<<c<<endl;}
   friend void MyCopy(Point&, Pixel&);
};
```


FUNZIONI `friend`

```
void MyCopy(Point& po, Pixel& pi){
   po.x = pi.c;
   po.y = pi.r;
}

int main(){
   Pixel pi1(1,2);
   Point po1(-3.1,0);

   MyCopy(po1,pi1);

   po1.Print();
}
```

Una possibile uscita

```
x=2, y=1
```

- Si potrebbe decidere di rendere la funzione *membro di una classe* e `friend` dell'altra.

METODI `friend`

```
class Pixel; // solo la dichiarazione

class Point{
   float x,y;
public:
   Point(float a, float b): x(a),y(b){}
   void Print(){cout<<"x="<<x<<", y="<<y<<endl;}
   void MyCopy( Pixel&);
};

class Pixel{
   int r,c;
public:
   Pixel(int p1, int p2): r(p1),c(p2){}
   void Print(){cout<<"r="<<r<<", c="<<c<<endl;}
   friend void Point::MyCopy( Pixel&);
};
```

Funzione membro

METODI `friend`

```
void Point::MyCopy(Pixel& pi){
   x = pi.c;
   y = pi.r;
}


int main(){

   Pixel pi1(11,22);
   Point po1(-3.1,0);

   po1.MyCopy(pi1);

   po1.Print();

}
```

Una possibile uscita

x=22, y=11

SOMMARIO

- Parola chiave `extern`:
  - Direttiva di collegamento.
- Parola chiave `static`:
  - Variabili e funzioni.
  - Membri di una classe.
- Array di array: allocazione dinamica.
- Oggetti come dati membro.
- Classi annidate.

PAROLA CHIAVE `extern`

- Gli oggetti globali e le funzioni devono avere una definizione unica (*ODR, one definition rule*).
- In un programma composto da vari file vi è la possibilità di *dichiarare un oggetto* senza definirlo: la parola chiave `extern` assicura che l'oggetto è definito in un altro punto dello stesso file o in un altro file del programma.
- Applicato ad una funzione rende solo esplicito il fatto che la definizione è altrove.

PAROLA CHIAVE `extern`

Test.cpp

```
#include <iostream>
#include "Point.h"
using namespace std;

extern Point p1;

extern int a;

int main(){
  cout<<"------------"<<endl;

  cout<<p1.getx()<<", "<<p1.gety()<<endl;

  cout<<"a="<<a<<endl;

  cout<<"------------"<<endl;
}
```

Def.cpp

```
#include "Point.h"

Point p1(1,2);

int a=12;
```

Una possibile uscita

```
Point(1,2)
-----------
1, 2
a=12
-----------
~Point()
```


PAROLA CHIAVE `extern`

- Se si deve utilizzare una *funzione di libreria scritta in linguaggio C*, allora è necessario indicare al compilatore che la funzione è scritta in un altro linguaggio in modo tale da gestire in modo appropriato le *differenze*. A tal fine si usa una *direttiva di collegamento*:

```
extern "C" {
   int somma(int a, int b);
   float sqr(float x);
}
```

VARIABILI E `static`

- All'interno di una funzione è possibile dichiarare un oggetto locale che duri per tutta la durata del programma. Tale comportamento è ottenuto dichiarando l'oggetto `static`.
- Un oggetto locale statico è inizializzato la prima volta che l'esecuzione del programma passa attraverso la dichiarazione dell'oggetto.

- Oggetti globali a cui è applicato `static` hanno visibilità solo all'interno del file in cui sono definiti.

VARIABILI E `static`

```
#include <iostream>
#include "Point.h"
using namespace std;
void somma_s(){
   static Point p1(2,2);
   p1.setx(p1.getx()+1);
   cout<<p1.getx()<<" ";
}
void somma_n(){
   Point p1(2,2);
   p1.setx(p1.getx()+1);
   cout<<p1.getx()<<" ";
}
int main(){
   for(int i=0; i<5;i++)
        somma_s();
   cout<<endl;
   cout<<"----------------"<<endl;
   for(int i=0; i<5;i++)
        somma_n();
   cout<<endl;
}
```

Una possibile uscita

```
Point(2,2)
3 4 5 6 7
----------------
Point(2,2)
3 ~Point()
Point(2,2)
3 ~Point()
Point(2,2)
3 ~Point()
Point(2,2)
3 ~Point()
Point(2,2)
3 ~Point()

~Point()
```


MEMBRI E `static`

- Talvolta è utile che tutti gli oggetti di una particolare classe accedano ad un *dato comune* (per esempio per tener conto di quanti oggetti sono stati creati). Esiste una sola copia di un *dato membro statico* per tutta la classe, al contrario degli altri dati membro, di cui ogni oggetto ha la propria copia.
- Un dato membro è reso statico inserendo nella dichiarazione la parola chiave `static`.
- Il valore di un dato membro statico può cambiare nel tempo.

MEMBRI E `static`

- In generale, un dato membro statico è *inizializzato all'esterno* della definizione della classe e poiché si può fornire una *sola definizione*, non deve essere inserito nel file header.

- I metodi che accedono solo a dati membro statici dovrebbero essere dichiarati `static`, in tal modo possono non essere riferiti ad una istanza particolare della classe.
- Un metodo statico non ha il puntatore `this`, pertanto qualsiasi riferimento *implicito* (per esempio accedere ad un attributo non statico) od *esplicito* a tale puntatore provoca un errore di compilazione.

MEMBRI E `static`

```
#include <iostream>
using namespace std;
class Num{
   float x;
   static int n;
public:
   Num(float a=0):x(a){n++;}
   ~Num(){n--;}
   //…
   static int GetN(){ return n;}
};
int Num::n = 0;
int main(){
   cout<<Num::GetN()<<endl;
   Num a;
   cout<<a.GetN()<<endl;
   {
        Num v[3];
        cout<<"   "<<Num::GetN()<<endl;
   }
   cout<<Num::GetN()<<endl;
}
```

Una possibile uscita

```
0
1
   4
1
```

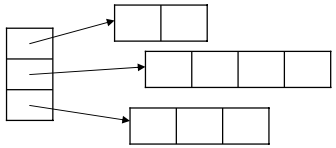
Invocazione attraverso nome qualificato

ARRAY DI ARRAY (DINAMICI)

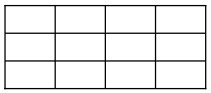
- Il vantaggio principale di un *array allocato dinamicamente* è che non occorre conoscere la sua dimensione al momento della compilazione. Ciò significa allocare tanto spazio di memoria quanto il programma richiede al momento.
- Tale tecnica è *più efficiente* di un array di dimensione prefissata: (1) la dimensione prefissata deve essere abbastanza lunga da contenere il dato che occupa più memoria, inoltre (2) è sufficiente che una istanza di dato sia più grande della dimensione prefissata perché il programma non funzioni.

ARRAY DI ARRAY (DINAMICI)

- Se *l'elemento dell'array* non è un singolo oggetto, ma un *insieme di istanze* di una classe il cui numero *non è noto* al momento della compilazione, allora è necessario allocare un nuovo array per ogni elemento dell'array di partenza.



- Gli *array bidimensionali* hanno invece la *seconda dimensione nota* al momento della compilazione ed *uguale* per ogni riga.



ARRAY DI ARRAY (DINAMICI)

- Vediamo un esempio in cui usiamo oggetti del tipo `Num` precedentemente introdotto.

```
class Num{
       float x;
       static int n;
public:
       Num(float a=0):x(a){n++;}
       ~Num(){n--;}
       float Get()const {return x;}
       void Set(float a){x=a;}
       static int GetN(){ return n;}
};
```

```
int Num::n = 0;
```

ARRAY DI ARRAY (DINAMICI)

```
#include <iostream>
#include "Num.h"
using namespace std;
int main(){
   int r=3, c[]={4,5,3};
   Num **pp;
   //---
   pp = new Num *[r];
   for(int i=0;i<r;i++)
        pp[i] = new Num[c[i]];
   cout<<"Numero oggetti: "<<Num::GetN()<<endl;
   //---
   for(int i=0, h=0;i<r;i++)
        for(int j=0;j<c[i];j++, h++)
                pp[i][j].Set(h);
   for(int i=0;i<r;i++){
        for(int j=0;j<c[i];j++)
                cout<<pp[i][j].Get()<<" ";
        cout<<endl;
   }
   //---
   for(int i=0;i<r;i++)
        delete [] pp[i];
   delete [] pp ;
   cout<<"Numero oggetti: "<<Num::GetN()<<endl;}
```

Una possibile uscita

```
Numero oggetti: 12
0  1  2  3
4  5  6  7  8
9  10  11
Numero oggetti: 0
```


OGGETTI COME ATTRIBUTI

- Per inizializzare in modo appropriato un *dato membro oggetto* di una classe è necessario passare argomenti al costruttore dell'attributo.
- Per fare ciò è necessario utilizzare la lista di inizializzazione dei membri: se il membro è un'istanza di una classe, i valori iniziali sono passati al costruttore appropriato.

OGGETTI COME ATTRIBUTI

```
#include <iostream>
#include "Point.h"
using namespace std;

class Line{
   Point p1,p2;
public:
   Line(float x1, float y1, float x2, float y2) :
   p1(x1,y1), p2(x2,y2)
   {
        cout<<"Line()------"<<endl;
        //p1(x1,y1);//errore: compile-time
   }
};

int main(){

   Line l1(1,1,3,3);

}
```

Una possibile uscita

```
Point(1,1)
Point(3,3)
Line()------
~Point()
~Point()
```


OGGETTI COME ATTRIBUTI

- La differenza tra l'uso della *lista di inizializzazione* dei membri e l'*assegnamento* dei dati membro nel corpo del costruttore è che solo la prima fornisce un'inizializzazione, cioè l'invocazione del costruttore appropriato per il dato membro.
- Nel corpo del costruttore possono avvenire solo assegnazioni, questo può essere fonte di errori e *inefficienza*.
- Prima dell'assegnamento nel corpo del costruttore è implicitamente chiamato il *costruttore di default* del membro istanza di una classe.

OGGETTI COME ATTRIBUTI

```
#include <iostream>
#include "Point.h"
using namespace std;

class Line{
   Point p1,p2;
public:
   Line(float x1, float y1, float x2, float y2){
        cout<<"Line()------"<<endl;
        p1.setx(x1); p1.sety(y1);
        p2.setx(x2); p2.sety(y2);
   }
};

int main(){

   Line l1(1,1,3,3);

}
```

Una possibile uscita

```
Point(0,0)
Point(0,0)
Line()------
~Point()
~Point()
```


CLASSI ANNIDATE

- Si può definire una classe all'interno di un'altra classe, in tal caso si dice classe annidata (*nested class*). Una classe annidata è un membro della classe che la racchiude.
- Una classe che racchiude una nested class non ha alcun privilegio di accesso ai membri privati della classe annidata .
- Nemmeno la classe annidata ha privilegi speciali di accesso ai membri privati della classe che la racchiude.

CLASSI ANNIDATE

- Solitamente le *classi annidate* supportano l'*astrazione* della classe esterna, pertanto non si vuole che siano accessibili (usabili) da tutto il programma.
- Si definisce la classe annidata membro privato: solo i membri della classe che la racchiude possono accedere al tipo della classe annidata.

CLASSI ANNIDATE

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

class Line{
   class Point{
        float x,y;
   public:
        Point(float a=0, float b=0) {
                x=a; y=b;
                cout<<"Point("<<a<<","<<b<<")"<<endl;
        }
        ~Point() {cout<<"~Point()"<<endl;};
        void Setx(float a) { x=a;}
        void Sety(float b){ y=b;}
        float Getx() const {return x;}
        float Gety() const {return y;}
   };
   Point p1,p2;
public:
                        ..↘
```

CLASSI ANNIDATE

```
      ..↖
   Line(float x1, float y1, float x2, float y2) :
     p1(x1,y1), p2(x2,y2){
          cout<<"Line()------"<<endl;
     }
     ~Line(){cout<<"~Line()------"<<endl;}
     void Movep1(float dx, float dy){
          //p1.x=dx;//errore compile-time
          p1.Setx(dx); p1.Sety(dy);
     }
};

int main(){

   Line l1(1,1,3,3);
   //Line::Point p2;//errore compile-time
   //l1.p1.Getx();//errore compile-time

}
```

Una possibile uscita

```
Point(1,1)
Point(3,3)
Line()------
~Line()------
~Point()
~Point()
```


CLASSI ANNIDATE

- Non vi è danno a rendere pubblici tutti i membri della classe annidata.
- Se un metodo della classe annidata non è definito *inline* nella classe stessa, allora deve essere definito *esternamente* alla classe che racchiude la classe annidata. Deve essere *qualificato* in modo opportuno.
- Una classe annidata non può accedere direttamente ai membri non statici della classe che la racchiude, anche se questi sono pubblici. Deve essere fatto *attraverso* un puntatore, riferimento o *oggetto* della classe esterna.

CLASSI ANNIDATE

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

class Line{
   class Point{
   public:
        float x,y;
        Point(float a=0, float b=0);
        ~Point() {cout<<"~Point()"<<endl;};
        void Setx(float a) { x=a;}
        void Sety(float b){ y=b;}
        float Getx() const {return x;}
        float Gety() const {return y;}
        //void ff(){
               //Movep1(2,2);//errore: compile-time
               //p1.x=33;//errore: compile-time
        //}
   };
   Point p1,p2;
public:
                    ..↘
```


CLASSI ANNIDATE

```
..↖  Line(float x1, float y1, float x2, float y2) :
       p1(x1,y1), p2(x2,y2){
            cout<<"Line()------"<<endl;
       }
       ~Line(){cout<<"~Line()------"<<endl;}
       void Movep1(float dx, float dy){
            //x=dx; //errore: compile-time
            p1.x=dx; p1.y=dy;
       }
       void Print(){
            cout<<"("<<p1.x<<","<<p1.y<<")";
            cout<<"-("<<p2.x<<","<<p2.y<<")"<<endl;
       }
};
Line::Point::Point(float a, float b) {
   x=a; y=b;
   cout<<"Point("<<a<<","<<b<<")"<<endl;
       }
int main(){
   Line l1(1,1,3,3);
   l1.Print();
   l1.Movep1(2,2);
   l1.Print();
}
```

Una possibile uscita

```
Point(1,1)
Point(3,3)
Line()------
(1,1)-(3,3)
(2,2)-(3,3)
~Line()------
~Point()
~Point()
```

Nome qualificato

CLASSI ANNIDATE

- Se si desidera che gli *utenti* della *classe esterna* non vedano i dettagli dell'implementazione della classe annidata, allora si deve definire la classe annidata fuori dal corpo della classe che la racchiude.

- Non si definisce la classe annidata nel file header che contiene l'interfaccia della classe esterna, ma nel file sorgente che ne contiene l'implementazione.

CLASSI ANNIDATE

Line.h

```
#ifndef LINE_H
#define LINE_H
class Line{
   class Point;
   //Point p1,p2;//errore: compile-time
   Point *p1,*p2;
public:
   Line(float x1=0, float y1=0, float x2=0, float y2=0);
   ~Line();
   void Movep1(float dx, float dy);
   void Print();
};
#endif
```

Dichiarazione necessaria

CLASSI ANNIDATE

Line.cpp

```
#include "Line.h"
#include <iostream>
using namespace std;

class Line::Point{
public:
   float x,y;
   Point(float a=0, float b=0){
        x=a; y=b;

        cout<<"Point("<<a<<","<<b
   <<")"<<endl;
   }
   ~Point()
   {cout<<"~Point()"<<endl;};
   void Setx(float a) { x=a;}
   void Sety(float b){ y=b;}
   float Getx() const {return x;}
   float Gety() const {return y;}
};
```

```
Line::Line(float x1, float y1,
   float x2, float y2){
   p1 = new Point(x1,y1);
   p2 = new Point(x2,y2);
   cout<<"Line()------"<<endl;
}
Line::~Line(){
   cout<<"~Line()------"<<endl;
   delete p1;
   delete p2;
}
void Line::Movep1(float dx,
   float dy){
   p1->x=dx; p1->y=dy;
}
void Line::Print(){
   cout<<"("<<p1->x<<","<<p1-
   >y<<")";
   cout<<"-("<<p2->x<<","<<p2-
   >y<<")"<<endl;}
```


CLASSI ANNIDATE

Test.cpp

```
#include <iostream>
using namespace std;
#include "Line.h"

int main(){

   Line l1(1,1,3,3);
   l1.Print();
   l1.Movep1(2,2);
   l1.Print();

}
```

Una possibile uscita

```
Point(1,1)
Point(3,3)
Line()------
(1,1)-(3,3)
(2,2)-(3,3)
~Line()------
~Point()
~Point()
```

Attenzione a non omettere i `delete` nel distruttore. Si avrebbe l'uscita visualizzata.

```
Point(1,1)
Point(3,3)
Line()------
(1,1)-(3,3)
(2,2)-(3,3)
~Line()------
```

SOMMARIO

- Costruttore per copia:
  - Dati membro puntatori.
- Namespace: introduzione.
- Operatori sovraccaricati:
  - `operator-()`
  - `operator+()`
  - `operator==()`
  - `operator<<()`

COSTRUTTORE PER COPIA

- L'inizializzazione di un oggetto di una classe con un altro oggetto della stessa classe viene chiamata *inizializzazione di default membro a membro*.
- Tale inizializzazione avviene automaticamente se non è fornito un costruttore esplicito (*copy-constructor*).
- L'operazione eseguita è quella di copia di ogni attributo.

COSTRUTTORE PER COPIA

- L'inizializzazione di un oggetto con un altro della sua classe avviene nelle seguenti situazioni:
  - Inizializzazione esplicita.

```
Point p1(1,2);
Point p2(p1);
```

  - Passaggio di un oggetto come argomento di una funzione.
  - Passaggio di un oggetto come valore di ritorno di una funzione.

COSTRUTTORE PER COPIA

- Il comportamento di default non è adeguato quando:
  - Un attributo è un puntatore a memoria allocata nello *heap*, in tal caso viene copiato solo l'indirizzo e non la memoria: quindi si hanno due oggetti che condividono la stessa area di memoria (*pointer aliasing*).
  - Un attributo deve avere un valore diverso per ogni oggetto (per es. il numero di conto bancario).
- In tali casi è necessario definire un costruttore per copia che implementi la corretta semantica di inizializzazione della classe.

COSTRUTTORE PER COPIA

- Come esempio si implementa una classe che *gestisce* l'allocazione dinamica di un array di `float`.
- Usare una classe (*wrapper*) rende l'utilizzo di certe operazioni più "sicuro" e "semplice": per esempio la deallocazione è automaticamente eseguita dal distruttore; si può evitare che gli indici escano dall'insieme dei valori consentiti; si possono fornire funzionalità di stampa.

COSTRUTTORE PER COPIA

```
#ifndef FLOATARRAY_H
#define FLOATARRAY_H

#include <iostream>
#include <cassert>
using namespace std;
class FloatArray{
   int size;
   float *data;
public:
   FloatArray(int dim=8, float val=0){
        size=dim;
        data = new float[size];
        for(int i=0;i<size;i++)
               data[i]=val;
        cout<<"FloatArray("<<size<<","<<val<<")"<<endl;
   }
   ~FloatArray(){
        delete [] data;
        cout<<"~FloatArray()"<<endl;          ..↘
   }
```


COSTRUTTORE PER COPIA

```
   ..↖
   void Set(int index, float val){
        assert(index>=0 && index<size);
        data[index]=val;
   }
   float Get(int index)const {
        assert(index>=0 && index<size);
        return data[index];
   }
   void Print() const{
        for(int i=0;i<size;i++)
               cout<<data[i]<<" ";
        cout<<endl;
   }
};
#endif
```


COSTRUTTORE PER COPIA

```
#include <iostream>
#include "FloatArray.h"
using namespace std;

int main(){
   FloatArray a(5,1);
   a.Set(0,2);
   //a.Set(10,2);
   a.Print();
}
```

Una possibile uscita

```
FloatArray(5,1)
2 1 1 1 1
~FloatArray()
```

L'oggetto è locale ma i dati sono allocati dinamicamente!

I valori degli indici sono controllati.

COSTRUTTORE PER COPIA

```
int main(){
  FloatArray a(5,1);
  FloatArray b(a);

  b.Set(0,3);
  a.Print();
  b.Print();
}
```

Pointer aliasing

Una possibile uscita

```
FloatArray(5,1)
3  1  1  1  1
3  1  1  1  1
~FloatArray()
~FloatArray()
```

Due chiamate a distruttore e una sola chiamata al costruttore.

COSTRUTTORE PER COPIA

- Deve essere implementato un *copy-constructor*:

```
FloatArray(const FloatArray &o){
    size=o.size;
    data = new float[size];
    for(int i=0;i<size;i++)
           data[i]=o.data[i];
    cout<<"FloatArray(const FloatArray& )"<<endl;
  }
```

```
FloatArray(5,1)
FloatArray(const FloatArray& )
1  1  1  1  1
3  1  1  1  1
~FloatArray()
~FloatArray()
```

Una possibile uscita del programma precedente.

namespace

- Per default ogni entità dichiarata nel *campo d'azione globale* deve avere un *nome unico*. Quindi si ha il rischio di avere una collisione di nomi (per esempio la propria classe ha lo stesso nome di una classe di libreria).
- Si può definire un `namespace` (spazio dei nomi) per *mascherare i nomi*. Si devono *qualificare* i nomi delle entità del `namespace` per poterli usare.

namespace

```
namespace Grafica{
  class Point{
  public:
      void Print(){
            cout<<"Grafica::Point::Print()"<<endl;
      }
  };

  const int size = 123;
}
```

- La definizione di un `namespace` può non costituire un blocco unico e può essere suddivisa in diversi file.

namespace

```
#include "Grafica.h"
class Point{
   public:
        void Print(){
               cout<<"Point::Print()"<<endl;
        }
   };
   const int size = 10;

int main(){

   Point p;
   p.Print();
   cout<<size<<endl;

   Grafica::Point pg;
   pg.Print();
   cout<<Grafica::size<<endl;

}
```

Una possibile uscita

```
Point::Print()
10
Grafica::Point::Print()
123
```

Nomi qualificati

namespace

- Si può usare un *alias di namespace* per associare un sinonimo più breve a un nome di namespace:

```
namespace G = Grafica;
```

- In tal caso si avrebbe il seguente codice:

```
G::Point pg;
pg.Print();
cout<<G::size<<endl;
```


namespace

- Si può utilizzare una *dichiarazione di uso* per accedere ai nomi senza usare la forma *qualificata*:

```
using Grafica::Point;
```

- In tal modo per riferirsi ai nomi dello spazio globale si deve fare uso dell'operatore `::`, perché la *dichiarazione di uso* ha introdotto i nomi nel campo d'azione in cui compare.

namespace

```
int main(){

   using Grafica::Point;
   using Grafica::size;

   ::Point p;
   p.Print();
   cout<<size<<endl;

   Point pg;
   pg.Print();
   cout<<size<<endl;

}
```

Una possibile uscita

```
Point::Print()
123
Grafica::Point::Print()
123
```

namespace globale

namespace

- Una *direttiva di uso* rende tutti i nomi di un namespace disponibili senza la forma qualificata:

```
using namespace Grafica;
```

- Tuttavia in tal modo si può avere nuovamente il problema delle collisioni dei nomi, perché è come se si fosse rimosso il `namespace`. Le collisioni sono rilevate solo quando si usano i nomi.

OPERATORI SOVRACCARICATI

- L'*overloading degli operatori* permette al programmatore di definire versioni specifiche degli operatori predefiniti che usino come operandi oggetti di una classe.
- In tal modo si rende la *manipolazione* degli *oggetti* altrettanto *intuitiva* di quella dei tipi predefiniti.
- Si pensi, per esempio, alla possibilità di sommare due oggetti utilizzando l'operatore `+`, oppure di visualizzare un oggetto attraverso l'operatore `<<`.

```
FloatArray a,b,c;
c=a+b;
cout<<c;
```


OPERATORI SOVRACCARICATI

- Un operatore sovraccaricato è dichiarato nel corpo della classe allo stesso modo di una normale funzione membro, il suo nome è costituito dalla parola chiave `operator` seguito da uno degli operatori del C++.

```
const FloatArray operator+(const FloatArray &o) const;
```

- È possibile sovraccaricare solo operatori predefiniti.

OPERATORI SOVRACCARICATI

- Il significato di un operatore non può essere cambiato per i tipi predefiniti. Si può solo sovraccaricare operatori per operandi di una classe.
- La precedenza predefinita non può essere modificata.
- La *arità* (numero di argomenti) predefinita degli operatori va conservata.
- Non possono essere sovraccaricati i seguenti operatori: `::` `.*` `.` `?:`

OPERATORI SOVRACCARICATI

- Per definire *quali operatori offrire* è buona norma: (1) definire l'*interfaccia pubblica* della classe (*quali operazioni deve offrire la classe ai suoi utenti*); (2) scegliere quali metodi possono essere definiti come operatori.
- Ogni operatore ha associato un *significato* che gli deriva dal suo uso con i tipi predefiniti: è importante che *non sia ambiguo* il suo significato sovraccaricato.
- Per esempio un metodo per la verifica dello stato `isEmpty()` potrebbe essere sostituito da `operator!()`, oppure `isEqual()` con `operator==()` e `copy()` con `operator=()`.

OPERATORI SOVRACCARICATI

- Un *operatore* sovraccaricato *membro* di una classe è considerato solo quando viene usato con un *operando sinistro* oggetto della classe.
- Se l'operando sinistro non è oggetto della classe (si pensi all'operatore <<: per esempio `cout<<c;`) allora si deve dichiarare l'operatore `friend`.
- Devono essere membri i seguenti operatori: `=` `[]` `()` `->`

OPERATORI SOVRACCARICATI

- Introduciamo alcuni operatori che agiscono per una semplice classe `Integer` che rappresenta un numero intero: in tal modo possiamo porre attenzione alle dichiarazioni degli operatori, mentre il loro significato è evidente.

OPERATORI SOVRACCARICATI

- Operatori *unari*:
  - `operator-()`, non modifica l'oggetto quindi è `const` e ritorna un nuovo oggetto della stessa classe con il segno cambiato. L'oggetto ritornato può essere `const` per evitare che vi si invii un messaggio la cui azione potrebbe andare persa.

OPERATORI SOVRACCARICATI

- Operatori *binari*:
  - `operator+()`, crea un nuovo valore modificato, quindi ritorna un nuovo oggetto della stessa classe. Non modifica gli oggetti operandi quindi possono essere `const` (e *reference*).
  - `operator==()`, verifica se due oggetti sono uguali, quindi torna un `bool`. Non modifica gli oggetti operandi quindi possono essere `const` (e *reference*).

OPERATORI SOVRACCARICATI

  - `operator<<()`: deve agire su *reference* dello *stream* di uscita per poterlo modificare, inoltre deve tornare un *reference* allo stesso *stream* per permettere la concatenazione delle espressioni di stampa. Non modifica l'oggetto che si vuole stampare, quindi può essere `const` (e *reference*) .

OPERATORI SOVRACCARICATI

```
class Integer{
   int n;
public:
   Integer(int i=0): n(i){}
   void Set(int val){n=val;}
   int Get() const {return n;}
   const Integer operator-()const{
        cout<<"operator-()"<<endl;
        return Integer(-n);
   }
   friend const Integer operator+(const Integer &l, const Integer &r){
        cout<<"operator+()"<<endl;
        return Integer(l.n + r.n);
   }
   friend bool operator==(const Integer &l, const Integer &r){
        cout<<"operator==()"<<endl;
        return l.n==r.n;
   }
   friend ostream& operator<<(ostream &os, const Integer &r){
        cout<<"operator<<()"<<endl;
        os<<r.n;
        return os;      }};
```


OPERATORI SOVRACCARICATI

```
#include <iostream>
#include "Integer.h"
using namespace std;

int main(){

   Integer a(3),b(5);
   cout<<"------------------"<<endl;
   cout<<-a<<endl;
   cout<<a<<endl;
   cout<<"------------------"<<endl;
   cout<<a+b<<endl;
   cout<<"------------------"<<endl;
   cout<< (a==b) <<endl;
   cout<<"------------------"<<endl;
   cout<<"Oggetto a="<<a<<"."<<endl;
   cout<<"------------------"<<endl;
}
```

Una possibile uscita

```
------------------
operator-()
operator<<()
-3
operator<<()
3
------------------
operator+()
operator<<()
8
------------------
operator==()
0
------------------
Oggetto a=operator<<()
3.
------------------
```

SOMMARIO

- Operatori sovraccaricati:
  - `operator++()`
  - `operator[]()`
  - `operator=()`
- Considerazioni circa l'efficienza.
- Conversioni definite dall'utente.

OPERATORI SOVRACCARICATI : ++

- Per gli operatori di incremento (++) e di decremento (--) esiste la versione *prefissa* e la versione *postfissa*.
- Per distinguere le dichiarazioni degli operatori *postfissi* da quelle degli operatori *prefissi*, le dichiarazioni degli operatori sovraccaricati *postfissi* hanno un parametro aggiuntivo di tipo `int`: tuttavia non è necessario dare un nome a tale parametro perché non viene usato nella definizione dell'operatore.

OPERATORI SOVRACCARICATI: ++

- L'operatore `operator++()` è *unario*:
  - modifica l'oggetto e quindi non è `const`;
  - ha due forme (*prefissa* e *postfissa*) e quindi due implementazioni;
  - ritorna lo stesso oggetto su cui è invocato.
- Si aggiunge tale operatore alla classe `Integer`.

OPERATORI SOVRACCARICATI: ++

```
const Integer& operator++(){//prefisso
      //cout<<"operator++() pre"<<endl;
      n=n+1;
      return *this;
  }

  const Integer operator++(int){//postfisso
      //cout<<"operator++() post"<<endl;
      Integer tmp(n);
      n=n+1;
      return tmp;
  }
```

OPERATORI SOVRACCARICATI: ++

```
#include <iostream>
#include "Integer.h"
using namespace std;

int main(){

   Integer a(0),b(0);
   cout<<"------------------"<<endl;
   cout<<"a="<<a<<",  ";
   cout<<"b="<<b<<endl;
   cout<<"------------------"<<endl;
   cout<<"++a="<<++a<<",  ";
   cout<<"b++="<<b++<<endl;
   cout<<"------------------"<<endl;
   cout<<"++a="<<++a<<",  ";
   cout<<"b++="<<b++<<endl;
   cout<<"------------------"<<endl;
}
```

Una possibile uscita

```
------------------
a=0,  b=0
------------------
++a=1,  b++=0
------------------
++a=2,  b++=1
------------------
```


OPERATORI SOVRACCARICATI: `[]`

- In generale, per le classi che rappresentano l'*astrazione di un contenitore,* in cui si possono leggere i singoli elementi, si può definire un operatore di *subscript*: `operator[]`. Con riferimento alla classe `FloatArray` i metodi `Get()` e `Set()` possono essere sostituiti con `operator[]`.
- Tale operatore deve poter comparire sia sul *lato sinistro* sia su quello *destro* di un operatore di assegnamento.

OPERATORI SOVRACCARICATI: `[]`

- Quindi il suo valore di ritorno deve essere un *lvalue*. Ciò si ottiene specificando il tipo di ritorno come un *riferimento*.

- L'uso del *reference* permette di utilizzare l'operatore di *subscript* in un modo *intuitivo*, come per i tipi predefiniti:

```
int a[3]; …; a[1]=23;
```


OPERATORI SOVRACCARICATI: `[]`

```
float& operator[](int index){
     assert(index>=0 && index<size);
     return data[index];
}

const FloatArray operator+(const FloatArray &r) const{
     assert(size==r.size);
     FloatArray tmp(size);
     for(int i=0;i<size ;i++)
            tmp.data[i]=data[i] + r.data[i];
     return tmp;
}

friend ostream& operator<<(ostream &os, const FloatArray &r) {
     for(int i=0;i<r.size ;i++)
            os<<r.data[i]<<" ";
     os<<endl;
     return os;
}
```

Da inserire in `FloatArray.h`

OPERATORI SOVRACCARICATI: [ ]

```
#include <iostream>
#include "FloatArray.h"
using namespace std;

int main(){
   int dim ;
   cout<<"Inserire dim=";
   cin>>dim;
   FloatArray a(dim,0), b(dim,10), c(dim,10);

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

   cout << a+b+c ;
}
```

Oggetto

Una possibile uscita

```
Inserire dim=7
20 21 22 23 24 25 26
```


OPERATORI SOVRACCARICATI: =

- L'assegnamento di un oggetto ad un altro della sua classe è gestito dall'*assegnamento di default membro a membro*. La meccanica è essenzialmente la stessa dell'inizializzazione di default membro a membro, ma fa uso di un *operatore implicito di assegnamento* per copia al posto del costruttore per copia.
- Se l' inizializzazione di default membro a membro è inadatta per la classe, lo è anche l'*assegnamento di default membro a membro.*

OPERATORI SOVRACCARICATI: =

- Si può definire un operatore esplicito di *assegnamento per copia*, `operator=()`, con cui si implementa la semantica corretta di copia per la classe.
- Si aggiunge l'operatore di assegnamento per copia alla classe `FloatArray`, altrimenti si avrebbe lo stesso problema di *aliasing dei puntatori* incontrato per i costruttori:

```
FloatArray a(5,0), b(5,1);
b=a;
b[0]=3;

cout<<a;
cout<<b;
```

Una possibile uscita

```
3 0 0 0 0
3 0 0 0 0
```


OPERATORI SOVRACCARICATI: =

```
FloatArray& operator=(const FloatArray &r){
    if(this != &r){
          size=r.size;
          delete [] data;
          data = new float[size];
          for(int i=0;i<size;i++)
                data[i]=r.data[i];
          cout<<"operator=(const FloatArray &r)"<<endl;
    }
    return *this;
}
```

Il test condizionale impedisce l'assegnamento di un oggetto a se stesso: è inopportuno nel caso in cui per prima cosa si libera una risorsa associata all'oggetto a sinistra (`delete [] data`) per assegnarvi la corrispondente risorsa (`r.data`) associata allo stesso oggetto a destra dell'assegnamento.

OPERATORI SOVRACCARICATI: =

```
#include <iostream>
#include "FloatArray.h"
using namespace std;

int main(){
  FloatArray a(5,0), b(10,0);
  for(int i=0;i<10;i++)
      b[i]=i;
  b=b;
  a=b;
  cout<<a;
}
```

Una possibile uscita

```
FloatArray(5,0)
FloatArray(10,0)
operator=(const FloatArray &r)
0 1 2 3 4 5 6 7 8 9
~FloatArray()
~FloatArray()
```


OPERATORI SOVRACCARICATI

Note.

- Se per una classe sono definiti gli operatori `operator+()` e `operator=()`, questi *non* supportano implicitamente l'equivalente operatore composto di assegnamento `+=`. È necessario definire esplicitamente l'operatore sovraccaricato `operator+=()`.
- L'implementazione dell'operatore di assegnamento ha la *semantica del valore*, cioè due oggetti dopo l'assegnamento sono distinti e successive modifiche ad un oggetto non hanno effetto sull'altro.

OPERATORI SOVRACCARICATI

- Tuttavia nei casi in cui tale copia comporta un costo eccessivo, si può implementare *la semantica del puntatore*: gli oggetti assegnati condividono i dati e si tiene traccia di tali assegnamenti (*reference counting*). Solo al momento di una modifica l'oggetto viene effettivamente copiato (*copy-on-write*).

CONSIDERAZIONI CIRCA L'EFFICIENZA

- In generale, rispetto al passaggio per valore, è più *efficiente* il passaggio di un oggetto a una funzione mediante un puntatore o un riferimento (è copiato solo l'indirizzo).
- Anche restituire un puntatore o un riferimento è più efficiente. Tuttavia non è semplice trovare una soluzione per tutti i casi.
- L'operatore `+` della classe `FloatArray` torna per valore, se i dati contenuti fossero di grosse dimensioni si potrebbero avere prestazioni inadeguate.

CONSIDERAZIONI CIRCA L'EFFICIENZA

- La soluzione di tornare un *riferimento* non è corretta perché si ritorna un *oggetto locale* che formalmente non esiste più dopo il completamento della funzione.
- Tornare un *puntatore* ad un'area di memoria nello *heap*, produce uno spreco di memoria perché nessuna parte del programma è responsabile per l'applicazione di `delete`.
- Si ottiene l'efficienza richiesta utilizzando un metodo con un *terzo parametro* in cui memorizzare il risultato, ma questo non permette di utilizzare la sintassi dell'operatore:

```
FloatArray c =a+b;
```


CONSIDERAZIONI CIRCA L'EFFICIENZA

- La tecnica di *calcolare il risultato direttamente nell'oggetto bersaglio* è implementata dal compilatore in modo trasparente per l'utente.
- Un ultimo aspetto riguardo all'efficienza: l'inizializzazione di un oggetto, `FloatArray c =a+b` , è più efficiente dell'assegnamento, `FloatArray c; c=a+b`.
- Questo perché non è possibile sostituire direttamente il bersaglio dell'assegnamento all'oggetto locale che viene restituito.

CONSIDERAZIONI CIRCA L'EFFICIENZA

- Vediamo due esempi che mostrano:
  - I metodi invocati nell'effettuare l'inizializzazione e l'assegnamento;
  - La loro differenza in termini di prestazioni.
- Si utilizza la classe `FloatArray`.

CONSIDERAZIONI CIRCA L'EFFICIENZA

```
cout<<"-------------"<<endl;
FloatArray a(10,1), b(10,1), c(10,1);
cout<<"-----"<<endl;
c=a+b;
cout<<"------------"<<endl;
FloatArray d(10,1),e(10,1);
FloatArray f=d+e;
cout<<"------------"<<endl;
```

Operazioni per l'espressione `c=a+b;`

Operazione per l'espressione `FloatArray f=d+e;`

Una possibile uscita

```
--------------------
FloatArray(10,1)
FloatArray(10,1)
FloatArray(10,1)
-----
FloatArray(10,0)
operator=(const FloatArray &r)
~FloatArray()
--------------------
FloatArray(10,1)
FloatArray(10,1)
FloatArray(10,0)
--------------------
~FloatArray()
~FloatArray()
~FloatArray()
~FloatArray()
~FloatArray()
~FloatArray()
```

CONSIDERAZIONI CIRCA L'EFFICIENZA

```
clock_t start,finish;

FloatArray a(100000,1), b(100000,1), c(100000,1);
start=clock();
for(int i=0; i<1000;i++)
     c=a+b;
finish=clock();
cout<<"Tempo di esecuzione  c=a+b             "<<(finish -
                          start)<<" ms"<<endl;

FloatArray d(100000,1), e(100000,1);
start=clock();
for(int i=0; i<1000;i++)
     FloatArray f=d+e;
finish=clock();
cout<<"Tempo di esecuzione  FloatArray f=d+e  "<<(finish -
                                 start)<<" ms"<<endl;
```

Una possibile uscita

Tempo di esecuzione c=a+b	469 ms
Tempo di esecuzione FloatArray f=d+e	219 ms

METODI INDISPENSABILI

- In generale, per un tipo `X`, il costruttore per copia `X(const X&)` si occupa dell'*inizializzazione* per mezzo di un oggetto dello stesso tipo.
- L'*inizializzazione* e l'*assegnamento* sono operazioni *differenti*.
- Se una classe `X` possiede un distruttore che esegue compiti non banali, quali *deallocare memoria dinamica*, tale classe deve implementare i seguenti metodi.

METODI INDISPENSABILI

```
class X{
  //...
  X(...);       //costrutori: creano oggetti

  X(const X&); //costrutore per copia

  X& operator=(const X&);//assegnamento
           //per copia: ripulisce e copia

  ~X();//distruttore: ripulisce
};
```


CONVERSIONI DEFINITE DALL'UTENTE

- Le *conversioni di tipo* sono applicate a operandi di tipo predefinito e agli argomenti di una funzione. Per esempio, `char a; int i; …; a+i;`, in questo caso l'operazione avviene tra interi (il compilatore ha gestito *implicitamente* una promozione).
- Il progettista di una classe può fornire un insieme di conversioni per oggetti della classe (eventualmente invocate implicitamente dal compilatore).

CONVERSIONI DEFINITE DALL'UTENTE

- Se si vuole avere la possibilità di sommare e sottrarre oggetti `Integer` sia con altri oggetti `Integer` sia con oggetti di tipi predefiniti, si deve fornire il supporto a sei funzioni operatore. Per esempio per la somma:

```
friend const Integer operator+(const int &l, const Integer &r);
friend const Integer operator+(const Integer &l, const int &r);
const Integer operator+( const Integer &r)const;
```

- Tuttavia si avrebbe un numero molto elevato di operatori, si preferisce fornire una soluzione per la conversione di tipo.

CONVERSIONI DEFINITE DALL'UTENTE

- La collezione di *costruttori* che prendono *un solo parametro* definisce un insieme di *conversioni* implicite dai valori dei tipi dei parametri al tipo della classe.
- Per la classe `Integer`:

```
Integer(int i=0): n(i){cout<<"Integer()"<<endl;}

friend const Integer operator-(const Integer &l, const Integer &r){
        cout<<"operator-()"<<endl;
        return Integer(l.n - r.n);
   }
const Integer operator+( const Integer &r)const{
        cout<<"operator+()"<<endl;
        return Integer(n + r.n);
   }
```


CONVERSIONI DEFINITE DALL'UTENTE

```
Integer a(11);
cout<<"------------------"<<endl;
cout<< a+3 <<endl;
//cout<< 3+a <<endl;//errore
cout<<"------------------"<<endl;
cout<< a-3 <<endl;
cout<< 3-a <<endl;
```

Una possibile uscita

```
Integer()
------------------
Integer()
operator+()
Integer()
14
------------------
Integer()
operator-()
Integer()
8
Integer()
operator-()
Integer()
-8
```

Conversione implicita

Operatore `friend`

CONVERSIONI DEFINITE DALL'UTENTE

- Se tale conversione implicita al tipo della classe non corrisponde a ciò che si desidera, si deve dichiarare *esplicito* il costruttore.

```
explicit Integer(int i=0): n(i){cout<<"Integer()"<<endl;}
```

- Si consideri il seguente codice:

```
Integer a(11);
cout<<"------------------"<<endl;
//cout<< a+3 <<endl;//errore
cout<< a+Integer(3) <<endl;
```

Una possibile uscita

```
Integer()
------------------
Integer()
operator+()
Integer()
14
```

CONVERSIONI DEFINITE DALL'UTENTE

- È possibile definire dei metodi che convertono un oggetto della classe in qualche altro tipo: *funzione di conversione*.
- Tali funzioni devono essere membri della classe. La loro dichiarazione non deve specificare un tipo di ritorno né una lista di parametri.

```
operator int() {
 cout<<"operator int()"<<endl;
 return n;
}
```


CONVERSIONI DEFINITE DALL'UTENTE

```
Integer a(11),b(7);
cout<<"------------------"<<endl;
cout<< a+b <<endl;
cout<<"------------------"<<endl;
cout<< 3+a <<endl;
cout<< a+3 <<endl;
cout<<"------------------"<<endl;
cout<< a-3 <<endl;
cout<< 3-a <<endl;
```

Una possibile uscita

```
Integer()
Integer()
------------------
operator+()
Integer()
18
------------------
operator int()
14
operator int()
14
------------------
operator int()
8
operator int()
-8
```

SOMMARIO

- Contenitori e algoritmi:
  - Libreria Standard del C++.
  - Iteratori.
  - Oggetti funzione.
- Esempi di implementazione:
  - Iteratori.
  - Oggetti funzione.

CONTENITORI E ALGORITMI

- La Libreria Standard del C++ fornisce molti strumenti per *manipolare* oggetti: contenitori e algoritmi.
- I principali contenitori sono:
  - *Contenitori sequenziali*: i principali sono `vector`, `list` e `deque` (simile al `vector`, ma specializzato nell'inserimento e rimozione del suo primo elemento).
  - *Contenitori associativi*: efficienti per la interrogazione e la rimozione di un elemento.

CONTENITORI E ALGORITMI

I principali contenitori associativi sono: `map` e `set`. Una `map` è una *coppia chiave/valore*: la chiave è usata per la ricerca e il valore contiene i dati che si ha intenzione di usare.

  - Per avere più occorrenze della stessa chiave si utilizzano `multimap` e `multiset`.
- Tali contenitori forniscono una *interfaccia minima*, che non contiene gli algoritmi (`find()`, `sort()`,…). Tali *operazioni comuni* a tutti i contenitori sono *fattorizzate* in una collezione di funzioni, gli *algoritmi generici.*

CONTENITORI E ALGORITMI

- L'algoritmo è *indipendente* dal tipo di *contenitore* a cui è applicato e dal tipo (e confronto) di eventuali argomenti.
- L'algoritmo deve, senza conoscere la sottostante struttura dati del contenitore:

  (1) scandire la collezione, cioè avanzare all'elemento successivo e riconoscere quando è finita la collezione;

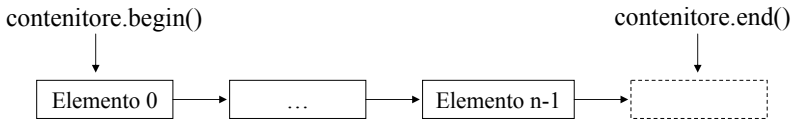
  (2) confrontare ogni elemento del contenitore (con l'eventuale argomento) per produrre il risultato della manipolazione.

SCANDIRE UN CONTENITORE

- La scansione avviene attraverso un ***iteratore*** che *astrae* il concetto di *puntatore a un elemento di una sequenza*:
  - Elemento puntato (dereferenziazione: operatori sovraccaricati `*` e `->`);
  - Spostarsi all'elemento successivo (incremento: operatore sovraccaricato `++`);
  - Confronto (operatori sovraccaricati `==` e `!=`).

SCANDIRE UN CONTENITORE

- L'intervallo di elementi che l'algoritmo deve scandire è marcato da una *coppia di iteratori* forniti dal contenitore:



- Il primo iteratore (`begin()`) punta al primo elemento su cui operare, il secondo (`end()`) punta ad una posizione *oltre* l'ultimo elemento su cui operare, agisce come una "sentinella".

SCANDIRE UN CONTENITORE

- Dato che gli algoritmi sono indipendenti dal tipo di contenitore si applicano anche agli array, in tal caso *come iteratori* sono utilizzati i *puntatori*.
- Come esempio di contenitore si utilizza il `vector`. È un array che può crescere dinamicamente e il *tipo degli elementi* è *scelto* al momento della *definizione*: si specifica il tipo desiderato inserendolo tra i simboli `< >`. Questa classe implementa il meccanismo dei `template`.

CONFRONTARE ELEMENTI DI UN CONTENITORE

- Il confronto può avvenire in due modi:
  - Utilizzando l'*operatore di uguaglianza* (in generale, `operator==()`) del tipo dell'elemento;
  - Utilizzando un ***oggetto funzione*** che implementa uno *specifico confronto* per il tipo (classe) dell'elemento. Cioè un confronto diverso dall'uguaglianza: per esempio ordinare oggetti di una classe `Image` in base o al nome, o alla dimensione, o alla profondità del colore.

- Un oggetto funzione si ha per una classe che sovraccarica l'operatore di chiamata di funzione, `operator()()`. L'*operatore incapsula* quello che normalmente sarebbe *implementato* come una funzione.
- Vi sono molti *oggetti funzione predefiniti*: si dividono in operazioni aritmetiche, relazionali e logiche. Inoltre vi sono degli adattatori di funzione per associare un operando di un oggetto funzione a un valore.

- Per confrontare due oggetti, per esempio, si potrebbe utilizzare una classe predefinita come `equal_to`, che fornisce oggetti funzione.
- Tale classe è implementata con il meccanismo dei `template`: è un predicato binario (un oggetto funzione che restituisce un bool si dice predicato) che ritorna `true` se i due oggetti operandi soddisfano il criterio di uguaglianza ==.

CLASSE `Point`

```
class Point{
  float x;
  float y;
public:
  Point(float a=0, float b=0){
      x=a; y=b;
  }
  //...
  bool operator==(const Point &r) const{
      return x==r.x && y==r.y;
  }
  friend ostream& operator<<(ostream &os, const
  Point &r){
      os<<"("<<r.x<<","<<r.y<<")"<<endl;
      return os;
  }};
```

Nel caso in cui si utilizzano oggetti (al posto dei tipi predefiniti) come elementi dei contenitori si considera la classe `Point`.

CONFRONTO

- Esempio di uso di un oggetto funzione di tipo `equal_to` per verificare se due punti sono uguali:

```
int main(){
  Point p1(2,3), p2(2,3);

  equal_to<Point> comp;

  bool out = comp(p1,p2);

  cout<<boolalpha;
  cout<<"Il confronto e` "<<out<<endl;
}
```

E necessario includere `<functional>`

Un manipolatore, come `endl`, che modifica lo stato di formattazione dell'oggetto `cout`

Una possibile uscita

```
Il confronto e` true
```

CONTENITORI E ALGORITMI

- Per trovare un particolare *oggetto* all'interno di un `vector` (un esempio di contenitore) si utilizza l'algoritmo generico della Libreria Standard `find()`, un tipico uso è il seguente:

```
iteratore_out = find(iteratore_begin, iteratore_end, oggetto);
```

- Gli elementi nell'intervallo `[begin, end)` sono confrontati usando l'operatore di uguaglianza del tipo dell'oggetto, quindi, per esempio, la classe `Point` deve sovraccaricare l'operatore ==.
- Se si trova una corrispondenza, la ricerca finisce e si ritorna un iteratore all'elemento, altrimenti si ritorna `iteratore_end`.

CONTENITORI E ALGORITMI: esempio

Esempio di ricerca di un punto all'interno di un `vector`

```
#include "Point.h"
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main(){
   vector<Point> v;
   v.push_back(Point(0.3,-0.9));v.push_back(Point(1,2));
   v.push_back(Point(3,6));v.push_back(Point(-1,-2));

   vector<Point>::iterator out;

   out=find(v.begin(), v.end(), Point(3,6));

   if(out!=v.end())
        cout<<"Punto trovato "<< *out <<endl;
}
```

Un `vector` di `Point`

Un iteratore per `vector`

Una possibile uscita

```
Punto trovato (3,6)
```


CONTENITORI E ALGORITMI : esempio

Esempio di ricerca con array

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

int main(){
   int val=16;
   int a[6];

   for(int i=0;i<6;i++){
        a[i]=i*i;
   }

   int *res=find(&a[0], &a[6], val);

   if(res == &a[6])
        cout<<"Valore non presente."<<endl;
   else
        cout<<"Valore presente."<<endl;

   cout<<"----------------------"<<endl;
```

Algoritmo generico con array

Elemento non valido

CONTENITORI E ALGORITMI : esempio

Esempio di ricerca con vector

```
   vector<int> v;

   for(int i=0;i<6;i++){
        v.push_back(i*i);
   }
   vector<int>::iterator iter = v.begin();
   vector<int>::iterator iter_end = v.end();

   for( ; iter!=iter_end; iter++)
        cout<< *iter <<" ";
   cout<<endl;

   vector<int>::iterator out;

   out = find(v.begin(), v.end(), val);

   if(out == v.end())
        cout<<"Valore non presente."<<endl;
   else
        cout<<"Valore presente."<<endl;
}
```

Un `vector` di `int`

Tipica iterazione su un contenitore

Una possibile uscita

```
Valore presente.
----------------------
0 1 4 9 16 25
Valore presente.
```

CONTENITORI E ALGORITMI

- Per verificare una condizione più generale si può utilizzare l'algoritmo generico della Libreria Standard `find_if()`, un tipico uso è il seguente:

```
iteratore_out = find_if(iteratore_begin, iteratore_end, predicato);
```

- Gli elementi nell'intervallo `[begin, end)` sono esaminati ordinatamente e l'oggetto funzione `predicato` è applicato ad ognuno. Se l'oggetto funzione risulta `true`, la ricerca finisce e si ritorna un iteratore all'elemento, altrimenti si ritorna `iteratore_end`.

CONTENITORI E ALGORITMI : esempio

Esempio con oggetti funzione

```
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>

using namespace std;
int main(){
   int a[]={14,2,3,3,1,25};

   vector<int> v(&a[0],&a[6]);

   for(int i=0;i<6;i++){
        cout<<v[i]<<" ";
   }
   cout<<"\n----------------------"<<endl;
```

Una possibile uscita

```
14 2 3 3 1 25
----------------------
Non vi e` valore uguale a 10
Valore minore di 10 e` 2
Valore maggiore di 10 e` 14
```


CONTENITORI E ALGORITMI : esempio

```
   vector<int>::iterator out;

   out = find_if(v.begin(), v.end(),  bind2nd(equal_to<int>(),10));

   if(out != v.end())
        cout<<"Valore uguale a "<<10<<" e` " <<*out <<endl;
   else
        cout<<"Non vi e` valore uguale a "<<10<<endl;

   out = find_if(v.begin(), v.end(),  bind2nd(less_equal<int>(),10));

   if(out != v.end())
        cout<<"Valore minore di "<<10<<" e` " <<*out <<endl;

   out = find_if(v.begin(), v.end(),  bind2nd(greater<int>(),10));

   if(out != v.end())
        cout<<"Valore maggiore di "<<10<<" e` " <<*out <<endl;

}
```


IMPLEMENTAZIONE DI ITERATORI

- Vediamo i *meccanismi base* per implementare un iteratore.
- Si fornisce un *insieme minimo* di metodi e attributi.
- Il contenitore è la classe `FloatArray`. In generale, si pensi ad un *tipo qualsiasi* (anche definito dall'utente) e non solo un `float` come nei prossimi esempi.
- La classe che rappresenta l'iteratore, `Iterator`, è *interna* perché *specifica* del contenitore su cui si vuole iterare. I costruttori sono privati.
- Non vi è il costruttore di default perché un iteratore deve essere associato ad un contenitore.

IMPLEMENTAZIONE DI ITERATORI

```
#ifndef FLOATARRAY_H
#define FLOATARRAY_H

#include <iostream>
#include <cassert>

using namespace std;

class FloatArray{
   int size;
   float *data;
public:
   FloatArray(int dim=8, float val=0);
//...
   class Iterator;
   friend class Iterator;
   Iterator begin();
   Iterator end();

};
```

Classe interna

IMPLEMENTAZIONE DI ITERATORI

```
class FloatArray::Iterator{
   friend class FloatArray;
   FloatArray *fa;
   int index;

   Iterator(FloatArray *o): fa(o),index(0){}
   Iterator(FloatArray *o,int): fa(o),index(o->size){}
public:
   Iterator & operator++(){
        assert(index<fa->size);
        index++;  return *this;
   }
   float operator*(){
        assert(index<fa->size);
        return fa->data[index];
   }
   bool operator!=(const Iterator &l)const{
        return index!=l.index;
}};
```

Costruttore *privato* che inizializza l'iteratore un elemento oltre la fine dell'array.

IMPLEMENTAZIONE DI ITERATORI

```
FloatArray::Iterator FloatArray::begin(){
        return Iterator(this);
   }
FloatArray::Iterator FloatArray::end(){
        return Iterator(this,1);
   }
```

```
int main(){

        FloatArray a(10,0);
        for(int i=0;i<10;i++)
                a[i]=i;

        FloatArray::Iterator iter = a.begin();
        FloatArray::Iterator iter_end = a.end();
        for(iter; iter!=iter_end; ++iter)
                cout<< *iter <<" ";
        cout<<endl;     }
```

Una possibile uscita

```
0 1 2 3 4 5 6 7 8 9
```


ALGORITMO GENERICO

- Come esempio di *algoritmo* si sceglie la *ricerca* di un valore all'interno di un contenitore sequenziale non ordinato.
- Quindi devo percorre *linearmente* tutti gli elementi senza conoscere la struttura dati sottostante: sfrutto il *comportamento* (*interfaccia*) *comune* degli iteratori.
- In questo caso uso l'operatore di uguaglianza ==.

ALGORITMO GENERICO

```
FloatArray::Iterator trova(FloatArray::Iterator first,
                           FloatArray::Iterator last,float val){
   for( ; first!=last; ++first)
        if(val == *first)
               return first;
   return last;
}
int main(){
   FloatArray a(10,0);
   for(int i=0;i<10;i++)
        a[i]=i;
   cout<<a;
   FloatArray::Iterator out = trova(a.begin(), a.end(),10);
   if(out != a.end())
        cout<<"Valore uguale a "<<10<<" e` " <<*out <<endl;
   else
        cout<<"Non vi e` valore uguale a "<<10<<endl;

   out = trova(a.begin(), a.end(),5);
   if(out != a.end())
        cout<<"Valore uguale a "<<5<<" e` " << *out <<endl;
   else
        cout<<"Non vi e` valore uguale a "<<5<<endl;}
```

Una possibile uscita

```
0 1 2 3 4 5 6 7 8 9
Non vi e` valore uguale a 10
Valore uguale a 5 e` 5
```


OGGETTO FUNZIONE

- Un limite dell'implementazione dell'algoritmo è l'uso dell'operatore ==: può non essere supportato dal tipo dell'elemento oppure, in generale, si può voler fare una ricerca secondo un criterio di confronto diverso.
- La soluzione tradizionale è di *parametrizzare* il confronto utilizzando un *puntatore a funzione*.
- Una soluzione migliore è utilizzare un *oggetto funzione*: una classe che sovraccarica l'operatore di chiamata di funzione, `()`.

OGGETTO FUNZIONE

- Se una classe è definita in modo da rappresentare un'operazione, si può sovraccaricare l'operatore di chiamata di funzione per invocare l'operazione stessa.
- L'operatore, `operator()()`, deve essere dichiarato come membro. Può avere una lista di parametri e ritornare un tipo, come una funzione.
- L'*operatore* è invocato *applicando* una lista di *argomenti* a un *oggetto* della sua classe.

OGGETTO FUNZIONE

- L'oggetto funzione ha prestazioni migliori perché può essere espanso *inline*.
- Inoltre essendo un oggetto dispone di un proprio stato, in cui memorizzare dati, e di metodi per l'inizializzazione e l'estrazione di tali dati.
- La classe `Comparatore` implementa il criterio di confronto per mezzo dell'operatore di chiamata.

OGGETTO FUNZIONE: esempio

```
class Comparatore{
   float val;
public:
   Comparatore(float i=0):val(i){}

   bool operator()(float x){
        return x<=val;
   }
};


FloatArray::Iterator TrovaSe(FloatArray::Iterator first,
             FloatArray::Iterator last,Comparatore comp){
   for( ; first!=last; ++first)
        if(comp(*first))
             return first;
   return last;
}
```

Usa lo stato per memorizzare il valore di confronto.

Criterio di confronto implementato nell'oggetto `comp`.

OGGETTO FUNZIONE : esempio

```
int main(){
   float val=7;
   FloatArray a(6,0);
   for(int i=0,j=5;i<6;i++,j--)
        a[i]=j*j;
   cout<<a;

   FloatArray::Iterator out = TrovaSe(a.begin(),
                                      a.end(),Comparatore(val));
   if(out != a.end())
        cout<<"Valore minore di "<<val<<" e` " <<*out <<endl;
   else
        cout<<"Non vi e` valore minore di "<<val<<endl;

   out = find_if(a.begin(), a.end(),Comparatore(val));

   if(out != a.end())
        //...
}
```

Una possibile uscita

```
25 16 9 4 1 0
Valore minore di 7 e` 4
Valore minore di 7 e` 4
```

Oggetto temporaneo

Algoritmo della Libreria Standard

OGGETTO FUNZIONE

- Si può utilizzare la nostra implementazione di iteratore (con opportune modifiche) e oggetto funzione con gli algoritmi della Libreria Standard.
- È anche possibile sviluppare algoritmi per i contenitori della Libreria Standard:

Test.cpp

```
//...#include <vector>
vector<float>::iterator TrovaSe(vector<float>::iterator first,
               vector<float>::iterator last,Comparatore comp){
       for( ; first!=last; ++first)
               if(comp(*first))
                       return first;
       return last;}
//...main(){
       vector<float> v;
       for(int i=0;i<10;i++)
               v.push_back((10-i));
       vector<float>::iterator out;
       out=TrovaSe(v.begin(),v.end(),Comparatore(6));
//...
```


OGGETTO FUNZIONE : esempio

```
class Somma{
   float res;
public:
   Somma(float a=0):res(a){}
   void operator()(float x){
        res+=x;
   }
   float Risultato() const{
        return res;
   }
};

Somma PerOgni(FloatArray::Iterator first, FloatArray::Iterator
   last,Somma op){
   for( ; first!=last; ++first)
        op(*first);
   return op;
}
```

Esempio di oggetto funzione che implementa un operatore matematico: una sommatoria.

Esempio di algoritmo che applica ad ogni elemento del contenitore un oggetto funzione.

OGGETTO FUNZIONE : esempio

```
int main(){

   FloatArray a(10,0);
   for(int i=0;i<10;i++)
        a[i]=i;
   cout<<a;
   Somma s;
   s = PerOgni(a.begin(), a.end(),s);

   cout<<"La somma dell'array e` "<<s.Risultato()<<endl;
}
```

Una possibile uscita

```
0 1 2 3 4 5 6 7 8 9
La somma dell'array e` 45
```


OGGETTO FUNZIONE : esempio

```
#include "Point.h"
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;


class Comparatore{
   float raggio;
public:
   Comparatore(float x=0):raggio(x){}

   bool operator()(Point &p){
        float r = sqrt(p.getx()*p.getx() + p.gety()*p.gety() );
        return r < raggio;
   }
};
```

Esempio per verificare *se un punto nel contenitore è all'interno del cerchio unitario*: si dovrebbe utilizzare un diverso criterio di confronto rispetto a <= il cui significato per la classe `Point` non è chiaro.

OGGETTO FUNZIONE : esempio

```
int main(){
   vector<Point> v;
   v.push_back(Point(1,2));v.push_back(Point(0.3,-0.9));
   v.push_back(Point(3,6));v.push_back(Point(-1,-2));

   vector<Point>::iterator out;

   out=find_if(v.begin(), v.end(), Comparatore(1));

   if(out!=v.end())
        cout<<"Punto trovato "<< *out <<endl;
}
```

Una possibile uscita

```
Punto trovato (0.3,-0.9)
```

SOMMARIO

- Gestione memoria:
  - Espressione `new` di piazzamento.
  - Overloading `new` e `delete`.

GESTIONE MEMORIA

- Per default l'*allocazione* e la *deallocazione* di un oggetto sullo *heap* è eseguita dagli operatori globali `new()` e `delete()` definiti nella Libreria Standard del C++.
- È possibile modificare tale comportamento globale, ma solitamente si esegue *l'overloading per singole classi*.

- Vediamo per primo un uso diverso dell'allocazione dinamica di oggetti: l' espressione `new` di piazzamento.

ESPRESSIONE `new` DI PIAZZAMENTO

- Il programmatore può richiedere che un oggetto sia *creato* (inizializzato) in una zona di *memoria già allocata*. In questo caso si usa l'espressione `new` di piazzamento: si specifica l'indirizzo di memoria in cui deve essere creato l'oggetto,

```
new (indirizzo) tipo
```

- In tal modo posso creare un oggetto sempre nella stessa zona di memoria.
- Questo meccanismo risulta importante nella programmazione di *dispositivi hardware* in cui un *oggetto* deve utilizzare specifici indirizzi di memoria.

ESPRESSIONE `new` DI PIAZZAMENTO

- Non vi è un'espressione `delete` corrispondente, in quanto tale istruzione non è necessaria: le espressioni `new` di piazzamento non allocano memoria.
- Tuttavia può risultare necessario *distruggere* (rilasciare risorse) *l'oggetto creato* (prima di crearne uno nuovo): in tal caso si deve *invocare esplicitamente il distruttore.*
- Porre *attenzione* ad invocare esplicitamente un distruttore, ciò avviene la maggior parte delle volte *solo* in congiunzione con una espressione `new` di piazzamento.

ESPRESSIONE `new` DI PIAZZAMENTO

```
#include "Point.h"
#include <iostream>
#include <new>
using namespace std;

char *buf = new char[sizeof(Point)]; //simula uno specifico
                                     //indirizzo
int main(){

   Point *p=0;

   p = new (buf) Point(1,3.3);
   cout<< *p ;
   p->~Point();

   p = new (buf) Point(2, -5.1);
   cout<< *p ;
   p->~Point();

   delete [] buf;
}
```

Alloca memoria "grezza", ma nessun oggetto `Point`

Crea un oggetto `Point` nella memoria puntata da `buf`

Si distrugge l'oggetto lasciando la memoria sottostante disponibile

Una possibile uscita

```
Point(1,3.3)
(1,3.3)
~Point()
Point(2,-5.1)
(2,-5.1)
~Point()
```


GESTIONE MEMORIA

- Una classe può assumere la gestione della propria memoria fornendo metodi chiamati `operator new()` e `operator delete()`.
- Gli operatori globali `new()` e `delete()` sono pensati per un *uso generale*, che può non essere adeguato a particolari necessità della propria classe:
  - La principale ragione è l'*efficienza*: per esempio, si devono creare e distruggere un tale elevato numero di oggetti che è necessario implementare un proprio meccanismo di allocazione ottimizzato.

GESTIONE MEMORIA

- Un'altra ragione è la *frammentazione dello heap*: implementando un proprio schema di allocazione si può prevenire tale problema.
- Oppure in *sistemi embedded* e in *sistemi real-time* in cui un'applicazione deve essere eseguita per molto *tempo* con *risorse limitate*. Per esempio l'allocazione deve avvenire sempre nella stessa quantità di tempo, oltre che essere efficiente ed evitare frammentazione.

OVERLOADING `new` & `delete`

- Quando si usa un'espressione `new` avvengono due cose: (1) prima si *alloca memoria* usando l'operatore `new` e poi (2) il *costruttore* è chiamato.
- Analogamente con un'espressione `delete`: prima (1) è chiamato il *distruttore* e poi (2) l'operatore `delete` *dealloca la memoria*.
- Le chiamate al costruttore e al distruttore *non* sono mai controllate dal programmatore della classe.
- Ciò che si cambia è la *strategia dell'allocazione* della memoria "grezza" (*raw storage*).

OVERLOADING `new` & `delete`

- L'operatore `new()`:
  - Ha un parametro di tipo `size_t` (definito in `<cstddef>`) inizializzato automaticamente con un valore che rappresenta la dimensione dell'oggetto in byte.
  - Ritorna un `void *` alla zona di memoria dove costruire l'oggetto. Notare che non è un puntatore ad un particolare tipo, perché deve solo produrre memoria e non un oggetto finito.
  - Se non vi è memoria deve essere lanciata un'*eccezione*.

OVERLOADING `new` & `delete`

- L'operatore `delete()`:
  - Ha un parametro di tipo `void *` inizializzato automaticamente con il puntatore fornito nell'espressione. Notare che non è un puntatore ad un particolare tipo, perché deve solo deallocare (il distruttore è già stato chiamato).
  - Ritorna un `void`.
- In generale, l'operatore `delete()` usato dovrebbe corrispondere all'operatore `new()` usato per allocare.
- Gli operatori `new()` e `delete()` sono membri statici.

OVERLOADING `new` & `delete`

- La classe di esempio è `MisureVI` dove si implementa solo la parte specifica alla gestione di memoria.
- La strategia di allocazione della memoria è gestire una *lista collegata di memoria* ad oggetti `MisureVI` disponibili, puntati dal puntatore `freeStore`.
- Se la lista è vuota viene chiamato l'operatore `new` globale per allocare un blocco di spazio che possa contenere un certo numero di oggetti. Tale indirizzo deve essere memorizzato, `begin`.
- Si tiene conto del numero di oggetti creati, `num`, per non allocarne un numero superiore a quello consentito, `numMax`.

OVERLOADING `new` & `delete`

MisureVI.h

```
#ifndef MISUREVI_H
#define MISUREVI_H
#include <cstddef>
#include <iostream>


class MisureVI{
   float tensioni[10];
   float correnti[10];
   //...
   //specifici per gestione memoria
   static MisureVI *freeStore;
   static int num;
   static int numMax;
   static char *begin;
   MisureVI *next;
```

Sono attributi statici perché devono essere comuni a tutti gli oggetti (il loro valore è unico per tutti gli oggetti).

OVERLOADING `new` & `delete`

MisureVI.h

```
public:
   MisureVI(){
        num++;
        std::cout<<"MisureVI()"<<std::endl;
   }
   ~MisureVI(){
        num--;
        std::cout<<"~MisureVI()"<<std::endl;
   }
   static int Num(){
        std::cout<<num<<"/"<<numMax<<" oggetti"<<std::endl;
        return num;}

   //...
   //specifici per gestione memoria
   void *operator new(std::size_t);
   void operator delete(void *);
};
#endif
```


OVERLOADING `new` & `delete`

MisureVI.cpp

```
#include "MisureVI.h"
#include <cstddef>
#include <cassert>
#include <iostream>

using namespace std;

MisureVI *MisureVI::freeStore=0;
char *MisureVI::begin=0;
int MisureVI::num=0;
int MisureVI::numMax=32;
```

Gli attributi statici sono inizializzati all'esterno della classe.

`numMax` indica il numero massimo di oggetti allocabili dinamicamente (per esempio 32).

OVERLOADING `new` & `delete`

MisureVI.cpp

```
void * MisureVI::operator new(size_t size){
   cout<<"\toperator new() "<<size<<" byte"<<endl;

   assert(num<numMax);

   MisureVI *p;

   if(!num){
        cout<<"\tbegin= new char[dim];"<<endl;
        size_t dim = size*numMax;
        begin= new char[dim];
        p=freeStore= reinterpret_cast<MisureVI *>(begin);
        for( ; p != &freeStore[numMax-1]; p++)
               p->next = p+1;
        p->next = 0;
   }

   p = freeStore;
   freeStore = freeStore->next;
   return p;
}
```

Se non vi sono ancora oggetti, si deve allocare la memoria "grezza" comune.

Si percorre la memoria allocata per creare la lista.

Ritorna il primo blocco di memoria libera.

OVERLOADING `new` & `delete`

MisureVI.cpp

```
void MisureVI::operator delete(void * p){
   cout<<"\toperator delete()"<<endl;

   static_cast<MisureVI*>(p)->next = freeStore;
   freeStore = static_cast<MisureVI*>(p);

   if(!num){
        cout<<"\tdelete [] begin"<<endl;
        delete [] begin;
   }
}
```

Si dealloca la memoria relativa ad un oggetto: la si *inserisce* nuovamente nella lista.

Si deve deallocare la memoria "grezza" comune a tutti gli oggetti.

OVERLOADING `new` & `delete`

MisureVI_test.cpp

```
#include "MisureVI.h"
#include <iostream>
#include <ctime>

using namespace std;


void test1(){
   MisureVI::Num();
   MisureVI *o1 = new MisureVI;
   MisureVI *o2 = new MisureVI;
   MisureVI::Num();
   cout<<"-----------"<<endl;
   delete o1;
   delete o2;
   MisureVI::Num();
}

int main(){
   test1();
}
```

Una possibile uscita

```
0/32 oggetti
      operator new() 84 byte
      begin= new char[dim];
MisureVI()
      operator new() 84 byte
MisureVI()
2/32 oggetti
----------------------
~MisureVI()
      operator delete()
~MisureVI()
      operator delete()
      delete [] begin
0/32 oggetti
```


OVERLOADING `new` & `delete`

- Per vedere le prestazioni si scrivono due *funzioni di test*: in una si allocano e deallocano tutti gli oggetti permessi dall'implementazione utilizzando l'operatore `new()` sovraccaricato e nell'altra si esegue lo stesso compito ma utilizzando l'operatore `new()` globale.
- Si verifica il tempo impiegato dalle due funzioni per ripetere tale compito un certo numero di volte.

OVERLOADING `new` & `delete`

```
void test2(){
   clock_t start,finish;
   MisureVI *o[32];
   start=clock();
   for(int i=0; i<100000;i++){
        for(int i=0;i<32;i++)
                o[i]= new MisureVI;
        for(int i=0;i<32;i++)
                delete o[i];
   }
   finish=clock();
   cout<<"Tempo di esecuzione  operator new() "<<(finish - start)<<"
   ms"<<endl;
}
void test3(){
   clock_t start,finish;
   MisureVI *o[32];
   start=clock();
   for(int i=0; i<100000;i++){
        for(int i=0;i<32;i++)
                o[i]= ::new MisureVI;
        for(int i=0;i<32;i++)
                ::delete o[i];
   }
   finish=clock();
   cout<<"Tempo di esecuzione  global new() "<<(finish - start)<<"
   ms"<<endl;}
```


OVERLOADING `new` & `delete`

```
int main(){
   test2();
   test3();
}
```

Una possibile uscita

```
Tempo di esecuzione  operator new() 109 ms
Tempo di esecuzione  global new() 703 ms
```

- Se si vuole gestire la memoria per array, devono essere sovraccaricati in modo analogo gli operatori `new[]()` e `delete[]()`.

SOMMARIO

- Programmazione orientata agli oggetti, OOP.
- Composizione.
- Ereditarietà:
  - Sintassi.
  - Metodi.
  - Upcasting.
  - Metodi speciali.
  - Upcasting e tipi dinamici (parola chiave `virtual`).

OOP

- La programmazione orientata agli oggetti *estende* la programmazione semplicemente basata sugli oggetti in modo da sfruttare le relazioni tipo/sottotipo. Questo si ottiene con il meccanismo dell'*ereditarietà* (*inheritance*).
- Anziché *reimplementare* le caratteristiche condivise, una classe *eredita* attributi e metodi della classe madre.
- La classe da cui si eredita è chiamata *classe base*, mentre la nuova classe è detta *classe derivata*.
- L'insieme di classi base e derivate è detta *gerarchia di ereditarietà di classi*.

OOP

- Se la classe base e derivata condividono la *stessa interfaccia pubblica*, la classe derivata è detta *sottotipo* (*subtype*).
- Un *puntatore* o un *riferimento* a una classe base possono *riferirsi* a qualunque *sottotipo derivato* senza l'intervento esplicito del programmatore.
- Manipolare più di un tipo con un puntatore o riferimento a una classe base è detto *polimorfismo*.
- Il *polimorfismo* dei sottotipi permette di scrivere la parte centrale e più importante di un'applicazione *senza curarsi dei singoli tipi* che si vogliono manipolare.

OOP

- Si *programma l'interfaccia* della classe base della nostra gerarchia mediante puntatori o riferimenti alla classe base. Il *tipo effettivo* cui essi si riferiscono è *risolto durante l'esecuzione* (*run-time*) quando viene chiamata l'istanza appropriata dell'interfaccia pubblica.
- La risoluzione durante l'esecuzione del metodo da invocare è detta *legame dinamico*, *dynamic binding* (o *late binding*). Per default i metodi sono risolti *staticamente* a tempo di compilazione, *static binding* (o *early binding*).
- Il legame dinamico è supportato dal meccanismo delle *funzioni virtuali* (*virtual functions*).

COMPOSIZIONE

- Una possibilità per *riusare il codice* già scritto da altri è implementare una classe che ha come *dati membro* degli *oggetti* di altre classi, cioè la composizione (*composition*).
- Il *vantaggio* è che *non* si accede direttamente al *codice*, ma se ne usano le *funzionalità*: in tal modo non si rischia di introdurre nuovi errori in codice che è corretto (è stato verificato da chi ha scritto le classi che si usano per implementare la propria classe).
- Un esempio di composizione può essere la classe che rappresenta una retta che usa due `Point` come attributi privati.

COMPOSIZIONE

```
#include "Point.h"
#include <iostream>
using namespace std;

class Retta{
   Point p1,p2;
public:
   Retta(Point &o1, Point &o2): p1(o1),p2(o2){}
   float pendenza(){
        //p2.x;//errore
        return (p2.gety() - p1.gety())/(p2.getx() - p1.getx());
   }
//...
};

int main(){
   Point p1(1,1), p2(2,2);
   Retta r(p1,p2);
   cout<<"Pendenza "<<r.pendenza()<<endl;
}
```

Lista di inizializzazione dei membri: i valori iniziali sono passati al costruttore appropriato

Una possibile uscita

```
Pendenza 1
```


COMPOSIZIONE

- I sotto-oggetti (*subobject*) `p1` e `p2` rappresentano l'implementazione della retta e quindi sono *privati*: in tal modo potrei *cambiare l'implementazione* utilizzando l'equazione della retta in forma normale (distanza della retta dall'origine e angolo tra la normale e il semiasse positivo delle ascisse) e lasciare la stessa interfaccia pubblica.

- In altri casi un sotto-oggetto potrebbe essere lasciato pubblico in modo da poter utilizzare la sua interfaccia.

COMPOSIZIONE

```
class DVD{
   //...
public:
   //..
   DVD& open(){cout<<"DVD aperto"<<endl; return *this;}
   DVD& close(){cout<<"DVD chiuso"<<endl; return *this;}
   void play(){cout<<"DVD..."<<endl;}
};

class Desktop{
   //...
public:
   //..
   DVD dvd;
};

int main(){
    Desktop pc;

    pc.dvd.open().close().play();
}
```

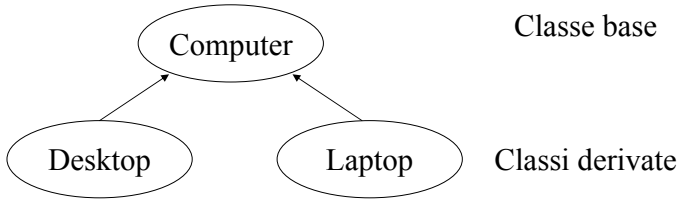
Una possibile uscita

```
DVD aperto
DVD chiuso
DVD...
```

COMPOSIZIONE ED EREDITARIETÀ

- Un altro modo di *riusare il codice* è attraverso il meccanismo dell'ereditarietà: in tal caso la nuova classe *eredita* attributi e metodi della classe base.
- In generale, la composizione implica una relazione del tipo "*has-a*", mentre l'ereditarietà implica una relazione del tipo "*is-a*": un Desktop *ha un* DVD e non *è un* DVD. Mentre un Desktop *è un* Computer.
- La *classe base* rappresenta una *astrazione* più *generale*, mentre le *classi derivate* ne sono una *specializzazione*.

EREDITARIETÀ



- La definizione della gerarchia di classi mostrata è la seguente:

```
class Computer{…};
class Desktop: public Computer{…};
class Laptop: public Computer{…};
```


EREDITARIETÀ: sintassi

- La parola chiave `public` indica che i membri pubblici della classe base rimangono pubblici per la classe derivata, altrimenti sarebbero privati.
- La classe derivata contiene un *sotto-oggetto* della classe base.
- Per inizializzare un sotto-oggetto si deve chiamare l'opportuno costruttore nella *lista di inizializzazione dei costruttori*.
- I distruttori sono automaticamente chiamati per l'intera gerarchia.

EREDITARIETÀ : sintassi

```
class Base{
   int b;
public:
   Base(int i=0):b(i){cout<<"Base("<<b<<")"<<endl;}
   ~Base(){cout<<"~Base()"<<endl;}
};
class Derivata : public Base{
   int d;
public:
   Derivata(int i=0): Base(i) , d(i){
        cout<<"Derivata("<<d<<")"<<endl;}
   ~Derivata(){cout<<"~Derivata()"<<endl;}
};
int main(){
   Base bobj(1);
   cout<<"-----------"<<endl;
   Derivata dobj(10);
   cout<<"-----------"<<endl;
   cout<<"Base "<<sizeof(bobj)<<" byte"<<endl;
   cout<<"Derivata "<<sizeof(dobj)<<" byte"<<endl;
   cout<<"-----------"<<endl;
}
```

Costruttore del sotto-oggetto della classe base

Una possibile uscita

```
Base(1)
-----------
Base(10)
Derivata(10)
-----------
Base 4 byte
Derivata 8 byte
-----------
~Derivata()
~Base()
~Base()
```

EREDITARIETÀ : metodi

- Se nella classe derivata si ridefinisce un metodo della classe base vi sono due possibilità:
  - Se il metodo ha la stessa firma e tipo di ritorno, si dice che si è eseguito un *redefining*; tuttavia se il metodo è virtuale (`virtual`) allora si è eseguito un *overriding*.
  - Se cambia la firma o il tipo di ritorno allora tutti i corrispondenti metodi sovraccaricati della classe base sono nascosti.

EREDITARIETÀ : metodi

```
class Base{
   //...
   void Set(int ){cout<<"Base::Set(int )"<<endl;}
   void Set(int , char){cout<<"Base::Set(int, char )"<<endl;}
   int Get(int ){cout<<"Base::Get(int )"<<endl;}
   void Print(){cout<<"Base::Print()"<<endl;}
};
class Derivata : public Base{
   //...
   void Set(int ){
        Base::Set(3);
        cout<<"Derivata::Set(int )"<<endl;}
   void Get(int ){cout<<"Derivata::Get(int )"<<endl;}
};
int main(){
   Derivata dobj(10);
   cout<<"-----------"<<endl;
   dobj.Set(3);
   //dobj.Set(3,'a');//errore
   cout<<"-----------"<<endl;
   dobj.Get(3);
   //int a=dobj.Get(3);//errore
   cout<<"-----------"<<endl;
   dobj.Print();
   cout<<"-----------"<<endl;
}
```

Diverso il tipo ritornato

Metodo presente solo nella classe base

Una possibile uscita

```
Base(10)
Derivata(10)
-----------
Base::Set(int )
Derivata::Set(int )
-----------
Derivata::Get(int )
-----------
Base::Print()
-----------
~Derivata()
~Base()
```


EREDITARIETÀ E COMPOSIZIONE

- Tuttavia cambiare l'interfaccia della classe base (modificando la firma o il tipo di ritorno dei suoi metodi) non è una prassi comune per l'ereditarietà.
- L'*ereditarietà* è pensata per supportare il *polimorfismo*: la possibilità di manipolare più di un tipo con un puntatore o riferimento a una classe base (se la classe base e le classi derivate condividono la stessa interfaccia pubblica).
- Se si *cambia l'interfaccia* allora forse è più opportuno utilizzare la *composizione*.

EREDITARIETÀ

- In generale, si vuole creare un *nuovo tipo* (*subtyping*) da *uno esistente,* in modo tale che il tipo derivato abbia la stessa interfaccia del tipo base (più i propri membri specifici) e quindi possa essere usato nello stesso modo del tipo base.

- Per esempio si vuole creare un *tipo numero complesso* che presenti un metodo per fornire la sua rappresentazione polare, ma che possa essere usato come un *numero complesso predefinito*: in questo caso è utile l'ereditarietà.

EREDITARIETA

Una possibile uscita

```
mod=1.41421, phase=0.785398
Re=1, Im=1
(1,-1)
```

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

class MyComplex: public complex<float>{
public:
   MyComplex(float a=0,float b=0):complex<float>(a,b){}
   //...
   void Polar(float &r, float &ph){
        r  = abs(*this);
        ph = arg(*this);
   }
};
int main(){
   MyComplex c(1,1);
   float mod=0, phase=0;

   c.Polar(mod,phase);
   cout<<"mod="<<mod<<", phase="<<phase<<endl;

   cout<<"Re="<<c.real()<<", Im="<<c.imag()<<endl;
   cout<<conj(c)<<endl;}
```

Metodo specifico di `MyComplex`

Metodi e funzioni standard di `complex`

EREDITARIETÀ PRIVATA

- Se una classe eredita in modo `private` allora la nuova classe ha tutte le funzionalità e dati della classe base ma in forma privata. L'eventuale utente non può accedervi e la classe derivata non può essere usata come un tipo della classe base.
- Quando si usa l'eredità privata si dice che "*si implementa in termini di*".
- Se si vuole rendere pubblico qualche membro, allora lo si deve dichiarare nella sezione pubblica della classe derivata.

EREDITARIETÀ: `protected`

- In alcuni casi è utile fornire l'accesso a una parte dell'implementazione della classe base, in modo che lo sviluppatore delle classi derivate la possa sfruttare. Tuttavia per gli altri utenti deve essere privata. Tale tipo di *accesso differenziato* è permesso dalla parola chiave `protected`.
- Una classe derivata può accedere direttamente ad un membro `protected`, mentre il resto del programma deve utilizzare una funzione pubblica di accesso. Tuttavia l'*accesso* diretto per la classe derivata è relativo solo al proprio *sotto-oggetto* e non ad un oggetto indipendente.

EREDITARIETÀ: `protected`

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

class Base{
protected:
   int b;
public:
   Base(int i=0):b(i){}
   void Set(int i){ b=i;}
   int Get() const{return b;}
};

class Derivata : public Base{
   int d;
public:
   Derivata(int i=0): Base(i) , d(i){}
   void Set(int i){ d=i; b=2*i;}
   void Get() const{cout<<"d="<<d<<", b="<<b<<endl;}
   int Elab(Base &bobj){
        //d=bobj.b;//errore
        return d=bobj.Get();
}};
```

Membro del proprio sotto-oggetto

Membro di un oggetto indipendente

EREDITARIETÀ: `protected`

```
int main(){
   Base bobj(10);
   Derivata dobj(0);

   //cout<<bobj.b;//errore

   dobj.Set(3);
   dobj.Get();

   cout<<"-----------"<<endl;

   cout<<dobj.Elab(bobj)<<endl;

}
```

Una possibile uscita

```
d=3, b=6
-----------
10
```


EREDITARIETÀ: upcasting

- Un aspetto importante dell'ereditarietà non è solo il fatto che le classi derivate hanno nuove funzionalità, ma è la *relazione di tipo* che esiste tra classe base e derivate: la nuova classe è un tipo della classe esistente.
- In tal modo tutte le *funzionalità* della classe *base* sono *presenti* nella classe *derivata* e quindi qualsiasi messaggio che può essere spedito ad un oggetto della classe base può essere spedito ad un oggetto della classe derivata.

EREDITARIETÀ: upcasting

- La *conversione* di un *puntatore* o *riferimento* ad un oggetto di una classe derivata in un puntatore o riferimento alla classe base è detto *upcasting* ed è supportato dal compilatore, perché risulta sicuro convertire un tipo specifico/derivato a uno più generale/base (fornisce le *funzionalità* minime e quelle *comuni*)

```
Derivata dobj;        Base *p = &dobj;
```

- Il termine *upcasting* si riferisce a come normalmente vengono disegnate le gerarchie di classi derivate: la classe base è in alto.

EREDITARIETÀ: metodi speciali

- Vi sono alcuni metodi speciali che non sono ereditati ma devono essere direttamente *implementati*: i *costruttori* e il *distruttore,* in particolare il *costruttore per copia* e *l'operatore di assegnamento*.

- Vediamo un semplice esempio generale che fornisce uno *schema* da seguire quando è *necessario* implementare specifiche soluzioni di costruzione per copia e assegnamento.

EREDITARIETA: metodi speciali

```
class Base{
public:
   Base(){cout<<"Base()"<<endl;}

   Base(const Base &){cout<<"Base(const Base &)"<<endl;}

   ~Base(){cout<<"~Base()"<<endl;}

   Base& operator=(const Base &r){
        if (&r != this) {
               //...
        }
        cout<<"Base::operator=()"<<endl;
        return *this;
   }
};
```


EREDITARIETA: metodi speciali

```
class Derivata : public Base{
public:
   Derivata(): Base() {cout<<"Derivata()"<<endl;}

   Derivata(const Derivata &o):Base(o){
   //Derivata(const Derivata &o){
        cout<<"Derivata(const Derivata &o)()"<<endl;
   }

   ~Derivata(){cout<<"~Derivata()"<<endl;}

   Derivata& operator = (const Derivata &r){
        if( &r != this){
               this->Base::operator = (r);
        }
        cout<<"Derivata::operator=()"<<endl;
        return *this;
   }
};
```

Costruttore per copia classe base: upcasting

Chiamata esplicita operatore = della classe base: upcasting

EREDITARIETÀ: metodi speciali

```
int main(){
   Derivata d1;
   cout<<"-----------"<<endl;
   Derivata d2(d1);
   cout<<"-----------"<<endl;
   d2=d1;
   cout<<"-----------"<<endl;

}
```

Una possibile uscita

```
Base()
Derivata()
-----------
Base(const Base &)
Derivata(const Derivata &o)()
-----------
Base::operator=()
Derivata::operator=()
-----------
~Derivata()
~Base()
~Derivata()
~Base()
```


EREDITARIETÀ: metodi speciali

- È importante seguire lo schema descritto, altrimenti si ottiene un comportamento non corretto, come si può verificare apportando le seguenti modifiche al codice del precedente esempio:

```
//Derivata(const Derivata &o):Base(o){
Derivata(const Derivata &o){
...
// this->Base::operator = (r);
```

Una possibile uscita

```
Base()
Derivata()
-----------
Base()
Derivata(const Derivata &o)()
-----------
Derivata::operator=()
-----------
~Derivata()
~Base()
~Derivata()
~Base()
```

Costruttore di default della classe base invece del costruttore per copia

Nessuna chiamata di default all'operatore di assegnamento della classe base

EREDITARIETÀ: upcasting e tipi dinamici

- Per default le funzioni membro sono invocate in base al *tipo statico* del riferimento o puntatore attraverso il quale sono invocate: in tal modo l'invocazione è determinata a tempo di compilazione.
- Questa scelta è stata fatta per *efficienza*.
- Per ottenere un comportamento dinamico, cioè la funzione membro invocata è determinata dal tipo dinamico (*run-time*) dell'oggetto cui fa riferimento il puntatore o reference attraverso il quale è invocata, tale funzione membro deve essere dichiarata `virtual`.

EREDITARIETÀ: upcasting e tipi dinamici

```
class Base{
   int b;
public:
   Base(int i=0):b(i){}
   void Get() const{cout<<"b="<<b<<endl;}
};

class Derivata : public Base{
   int d;
public:
   Derivata(int i=0): Base(i) , d(i){}
   void Get() const{cout<<"d="<<d<<" ";  Base::Get();}
   int Elab(){/*...*/}
};

void  Print(Base &o){
   //o.Elab();//errore
   o.Get();
}
```

Comportamento di default: legame statico

Si invoca il metodo della classe base

Il parametro è un riferimento quindi è supportato l'upcasting

È disponibile solo l'interfaccia della classe base

EREDITARIETÀ: upcasting e tipi dinamici

```
int main(){
   Base bobj(10);
   Derivata dobj(20);
   bobj.Get();
   dobj.Get();
   cout<<"-----------"<<endl;
   Print(bobj);
   Print(dobj);
}
```

upcasting

Una possibile uscita

```
b=10
d=20 b=20
----------
b=10
b=20
```

È invocato il metodo `Get()` della classe base: infatti è stampato solo il sotto-oggetto

EREDITARIETÀ: upcasting e tipi dinamici

- Per avere un *comportamento polimorfico* è necessario dichiarare `virtual` il metodo `Get()` nella classe base:

```
virtual void Get() const{cout<<"b="<<b<<endl;}
```

- In tal caso si ottiene la seguente uscita:

```
b=10
d=20 b=20
----------
b=10
d=20 b=20
```

È invocato il metodo `Get()` della classe derivata, cioè il tipo dinamico del riferimento

SOMMARIO

- Polimorfismo:
  - Parola chiave `virtual`.
  - Overloading e overriding.
- Classi base astratte: funzioni virtuali pure.
- Costruttori e distruttori: distruttori virtuali.
- Metodi: `Clone()` e <<.
- Esempi:
  - Libreria grafica.
  - Gerarchia di classi Object.

POLIMORFISMO

- Si consideri l'esempio visto nei lucidi precedenti:

```
class Base{
  //...
  virtual void Get() const{cout<<"Base"<<endl;}
};
class Derivata : public Base{
  //...
  void Get() const{cout<<"Derivata"<<endl;}
};
```

overriding

- E il seguente codice :

```
Base bobj;
Derivata dobj;

Base *p;

p = &bobj;
p->Get();

p = &dobj;
p->Get();
```

upcasting

dynamic binding

Una possibile uscita

```
Base
Derivata
```


POLIMORFISMO

- In corrispondenza di ogni *invocazione* durante l'*esecuzione* del programma, viene determinato l'*effettivo tipo* di classe a cui punta il riferimento o il puntatore e viene chiamata l'*istanza appropriata* del metodo, che deve essere dichiarato *virtuale nella classe base*. Il meccanismo che supporta tale comportamento è il *dynamic binding*.

- Nel *paradigma orientato agli oggetti* il programmatore manipola oggetti principalmente attraverso puntatori e riferimenti alla classe base.
- Nel *paradigma basato sugli oggetti* si manipola un'istanza di un singolo tipo fissato al momento della compilazione.

POLIMORFISMO

- Il polimorfismo esiste soltanto all'interno di singole gerarchie di classi.
- I puntatori `void *` possono essere considerati polimorfici, ma sono privi di un supporto *esplicito* da parte del linguaggio: devono essere gestiti mediante *cast espliciti* e qualche forma di *discriminante* che tenga traccia del tipo effettivo puntato.
- Per default, le funzioni membro di una classe *non* sono *virtuali* (per ragioni di efficienza): in tal caso la funzione invocata è quella definita dal *tipo statico* (quello a tempo di compilazione).

POLIMORFISMO

- Quando una funzione membro è dichiarata *virtuale*, allora la funzione invocata è quella definita dal *tipo dinamico* (quello a tempo di esecuzione).
- Questo permette di *estendere* la propria applicazione con nuovi tipi ed utilizzare sempre le funzionalità fornite dalla classe base (si programma l'interfaccia della classe di base).
- Il meccanismo delle funzioni virtuali è supportato solo con puntatori o riferimenti alla classe base.
- Nel caso degli *oggetti* si utilizza il tipo *statico*, cioè non vi è ambiguità sul tipo.

OBJECT SLICING

- Inizializzare un oggetto della classe base con uno di una classe derivata è legale. Tuttavia ciò che si ottiene è che solo il sotto-oggetto viene trasferito, la parte specifica della classe derivata è scartata (*object slicing*).
- Con riferimento al codice precedente:

```
Base bobj;
Derivata dobj;

bobj=dobj;

bobj.Get();
```

Una possibile uscita

```
Base
```

object slicing

OVERLOADING E OVERRIDING

- Quando si esegue un *overriding* di una funzione virtuale, eventuali metodi sovraccaricati della classe base sono nascosti.
- Non si può cambiare il tipo di ritorno di una funzione *overridden*. È possibile quando il tipo ritornato è un riferimento o puntatore ad una classe derivata dal tipo base.
- Le funzioni virtuali disponibili per un riferimento o puntatore della classe base sono quelle della classe base e non eventuali nuove funzioni definite nelle classi derivate.

OVERLOADING E OVERRIDING

```
class Base{
   //...
   virtual void Set(){cout<<"Base::Set()"<<endl;}
   virtual void Set(int){cout<<"Base::Set(int)"<<endl;}
   virtual Base* Get(){cout<<"Base::Get()"<<endl; return this;}
};
class Derivata : public Base{
   //...
    void Set(){cout<<"Derivata::Set()"<<endl;}
    //int Set(){cout<<"Derivata::Set()"<<endl; return 1;}//errore
    Derivata* Get(){cout<<"Derivata::Get()"<<endl; return this;}
    virtual void Print(){cout<<"Derivata::Print()"<<endl;}
};
int main(){
   Derivata dobj;
   Base *p = &dobj;
   dobj.Set();
   //dobj.Set(1);//errore
   cout<<"----------"<<endl;
   p->Set();
   p->Set(1);
   //p->Print();//errore
   p->Get();
}
```

Una possibile uscita

```
Derivata::Set()
----------
Derivata::Set()
Base::Set(int)
Derivata::Get()
```

CLASSI ASTRATTE

- Quando si vuole che la *classe base* rappresenti solo un'*interfaccia* per le sue classi derivate, ma che *non* sia *istanziata*, in questo caso si rende la *classe astratta*. Una classe è astratta quando contiene almeno una *funzione virtuale pura*: una dichiarazione di membro `virtual` seguita da `=0`.
- Quando una classe astratta è ereditata, la *classe derivata* deve fornire le *implementazioni* delle funzioni virtuali pure, altrimenti diventa essa stessa astratta.
- È utile quando la classe base rappresenta azioni che si applicano a tutte le classi derivate, ma non ha senso creare un oggetto della classe base.

CLASSI ASTRATTE

- È possibile fornire una definizione di una funzione virtuale pura: per esempio, per contenere codice utile alle classi derivate.
- Quando si invoca una funzione virtuale usando l'operatore di risoluzione di scopo di classe, si supera il meccanismo delle funzioni virtuali, forzando la risoluzione statica. È utile per invocare funzioni virtuali pure.

CLASSI ASTRATTE

```
class Base{
   //…
   virtual void Get()=0;
   virtual void Set()=0;
};
void Base::Set(){cout<<"Base::Set()"<<endl;};

class Derivata : public Base{
   //…
   void Get(){cout<<"Derivata::Get()"<<endl;}
   void Set(){
        Base::Set();
        cout<<"Derivata::Set()"<<endl;}
};
int main(){
   //Base bobj;//errore
   Derivata dobj;
   Base &r = dobj;
   r.Get();
   cout<<"----------"<<endl;
   r.Set();}
```

Definizione della funzione virtuale pura

Invocazione statica della funzione virtuale pura

Una possibile uscita

```
Derivata::Get()
----------
Base::Set()
Derivata::Set()
```


COSTRUTTORI E DISTRUTTORI

- I costruttori *non* possono essere virtuali, perché creano oggetti di una *classe specifica* e non puntatori. I puntatori possono essere usati per manipolare oggetti già creati.
- I costruttori della classe base sono chiamati per primi nella lista di inizializzazione dei costruttori.
- Eventuali funzioni virtuali invocate nel costruttore fanno riferimento alle *versioni locali*, cioè al tipo "corrente".
- Lo stesso comportamento si ha per i distruttori: eventuali funzioni virtuali invocate nel distruttore fanno riferimento alle *versioni locali*.
- I *distruttori* sono chiamati in *ordine inverso* rispetto ai *costruttori*.

COSTRUTTORI E DISTRUTTORI

- I *distruttori* possono essere *virtuali*: in generale, se una classe presenta una funzione virtuale *dovrebbe fornire* anche il distruttore virtuale.
- Poiché è frequente *manipolare* un *oggetto* attraverso un *puntatore* della sua classe *base*, quando è necessario utilizzare `delete`, allora dovrebbe essere invocato il distruttore appropriato. Ciò accade solo se il distruttore della classe base è dichiarato virtuale.
- Se il distruttore non è virtuale, vi è il rischio che non vengano rilasciate le risorse della classe derivata.

COSTRUTTORI E DISTRUTTORI

```
class Base1{
public:
   ~Base1(){cout<<"~Base1()"<<endl;}
};
class Base2{
public:
   virtual ~Base2(){cout<<"~Base2()"<<endl;}
};
class Derivata1 : public Base1{
public:
   ~Derivata1(){cout<<"~Derivata1(): rilascio risorse"<<endl;}
};
class Derivata2 : public Base2{
public:
   ~Derivata2(){cout<<"~Derivata2(): rilascio risorse"<<endl;}
};
int main(){

   Base1 *p1 = new Derivata1;
   delete p1;
   cout<<"----------"<<endl;
   Base2 *p2 = new Derivata2;
   delete p2;
}
```

Una possibile uscita

```
~Base1()
----------
~Derivata2(): rilascio risorse
~Base2()
```


COSTRUTTORI E DISTRUTTORI

- I distruttori virtuali puri sono legali, ma deve essere fornita anche una definizione.
- Tuttavia, in una classe derivata non è richiesto di fornirne esplicitamente una definizione.

- Dal momento che gli *operatori di output* sono membri della classe `ostream`, non è possibile fornire direttamente un operatore di output virtuale. Occorre fornire una *funzione virtuale indiretta* come nel seguente esempio.

OUTPUT VIRTUALE

```
class Base{
   //...
   virtual ostream& Print(ostream &os) const{
        os<<"Base::operator<<()"<<endl;
        return os;
   }
    friend ostream& operator<<(ostream &os, const Base &r){
         return r.Print(os);
   }

};

class Derivata : public Base{
   //...
   virtual ostream& Print(ostream &os) const{
        os<<"Derivata::operator<<()"<<endl;
        return os;
   }
};
```

Operatore << non virtuale

Invocazione virtuale di Print()

OUTPUT VIRTUALE

```
int main(){
  Base bobj;
  Derivata dobj;

  Base *p = &bobj;
  cout<<*p;
  cout<<"----------"<<endl;
  p = &dobj;
  cout<<*p;
}
```

Una possibile uscita

```
Base::operator<<()
----------
Derivata::operator<<()
```


METODO `clone()`

- Se si dispone di un puntatore alla classe base, come si può *allocare* una *copia* dell'oggetto puntato a tempo di esecuzione ?
- Per esempio, se si dispone di un puntatore `Derivata *d` inizializzato, allora si può utilizzare `new Derivata(*d)`. Ma se si ha un `Base *b` inizializzato, il comportamento *corretto* dipende dal tipo dinamico.
- L'operatore `new` non può essere reso virtuale perché si tratta di una funzione membro statica.
- Si può fornire un metodo virtuale per allocare e copiare oggetti sullo *heap*, tale metodo è generalmente chiamato `clone()`.

METODO `clone()`

- Il metodo *virtuale* `clone()` invoca il *costruttore* per copia *appropriato* per la classe e la *risoluzione dinamica* di invocazione dei metodi permette di attuare la corretta allocazione.
- Vediamo alcuni frammenti di codice.
- Nella classe `Base`:

```
virtual Base* clone() {return new Base(*this);}
virtual ~Base(){cout<<"~Base()"<<endl;}
```

- Nella classe `Derivata`:

```
virtual Derivata* clone() {
       return new Derivata(*this);
  }
```


METODO `clone()`

- Un esempio di utilizzo :

```
Base *p = new Base(1);
cout<<"----------"<<endl;
Base *pp=p->clone();
cout<<"----------"<<endl;
delete pp;
delete p;
cout<<"=========="<<endl;
p = new Derivata(10);
cout<<"----------"<<endl;
pp=p->clone();
cout<<"----------"<<endl;
delete pp;
delete p;
```

```
Base()
----------
Base::clone()
Base(const Base &)
----------
~Base()
~Base()
==========
Base()
Derivata()
----------
Derivata::clone()
Base(const Base &)
Derivata(const Derivata &)
----------
~Derivata()
~Base()
~Derivata()
~Base()
```

ESEMPIO: libreria grafica

- Lo sviluppo di un programma per disegnare forme geometriche bidimensionali può essere composto da tre parti: (i) un *gestore dello schermo*: funzioni e strutture dati a basso livello; (ii) una *libreria di forme*: forme generali complesse (rettangoli, cerchi) e funzioni per la loro gestione; (iii) l'*applicazione* finale.

- La libreria di forme definisce il *concetto generale di forma*, in modo tale che possa essere condiviso (come classe base `Shape`) da tutte le forme particolari (ad esempio, cerchi e quadrati) e che ogni *forma* possa essere *gestita* esclusivamente attraverso l'*interfaccia* fornita dalla classe base `Shape`.

ESEMPIO: libreria grafica

- Vediamo un semplice esempio (non è gestita l'allocazione e la deallocazione di forme, la cancellazione di forme, …).
- Il costruttore `Shape()` inserisce la forma in cima/fondo a una lista di forme `list`. Questa lista viene gestita attraverso l'elemento `next` in ogni oggetto `Shape`.
- Poiché la creazione di un generico oggetto `Shape` non ha senso, la classe è resa astratta.
- Si dichiara un metodo `draw()` per disegnare la forma.

ESEMPIO: libreria grafica

```
#ifndef GRAFICA2D_H
#define GRAFICA2D_H
#include <iostream>
using namespace std;

class Shape{
   static Shape *list ;
   Shape *next;
public:
   Shape(){
        cout<<"Shape() ";
        next=list;
        list = this;
   }
   virtual ~Shape(){cout<<"~Shape()"<<endl;}

   virtual void draw()=0;

   friend void shape_refresh();
};

Shape * Shape::list=0;
```


ESEMPIO: libreria grafica

- Ogni forma particolare, `Line`, `Circle`, definisce la propria struttura e il modo in cui viene disegnata (si deve implementare la funzione virtuale pura `draw()`).
- Oltre alle definizioni delle forme, una libreria di forme contiene *le funzioni per gestirle*.
- Con la funzione `shape_refresh()` vengono ridisegnate tutte le forme. Si osservi che non si conosce il *tipo effettivo* delle forme che vengono disegnate.

ESEMPIO: libreria grafica

```
..↖
void shape_refresh(){
   for(Shape *p=Shape::list; p ; p =  p->next)
        p->draw();
}

class Line: public Shape{
   //...
public:
   Line(/*...*/){cout<<"Line()"<<endl;}
   ~Line(){cout<<"~Line() ";}
   void draw(){cout<<"Line::draw()"<<endl;};
};

class Circle: public Shape{
   //...
public:
   Circle(/*...*/){cout<<"Circle()"<<endl;}
   ~Circle(){cout<<"~Circle() ";}
   void draw(){cout<<"Circle::draw()"<<endl;};
};
#endif
```


ESEMPIO: libreria grafica

- Tale libreria (compilata e distribuita) è la base per lo sviluppo di un'applicazione.
- In particolare un utente può definire proprie forme (derivate da `Shape`) e quindi estendere la libreria. È importante evidenziare che si può continuare ad utilizzare le funzioni di libreria per la gestione delle forme, comprese quelle definite dall'utente.

- Vediamo un esempio di applicazione.

ESEMPIO: libreria grafica

```
class MyShape: public Shape{
   //...
public:
   MyShape(/*...*/){
        cout<<"MyShape()"<<endl;}
   ~MyShape(){cout<<"~MyShape() ";}
   void draw(){
        cout<<"MyShape::draw()"<<endl;};
};

int main(){
   //Shape s;//errore
   Line l1, l2;
   Circle c1 ;
   MyShape m1;

   cout<<"-----------------"<<endl;
   shape_refresh();
   cout<<"-----------------"<<endl;
}
```

Una possibile uscita

```
Shape() Line()
Shape() Line()
Shape() Circle()
Shape() MyShape()
-----------------
MyShape::draw()
Circle::draw()
Line::draw()
Line::draw()
-----------------
~MyShape() ~Shape()
~Circle() ~Shape()
~Line() ~Shape()
~Line() ~Shape()
```


ESEMPIO: `PointStack (Object)`

- Nello sviluppo di classi e funzioni è sempre necessario dichiarare tipi specifici.
- Quindi se si vuole sviluppare, per esempio, uno *stack*, deve essere *implementato* per un *tipo specifico*. Di conseguenza non può essere usato in modo *generico*, cioè con un altro tipo.

- Vediamo un esempio di contenitore: uno *stack* per oggetti `Point`, in particolare si gestiscono puntatori.
- Per semplicità non si considerano i problemi relativi alla memoria.

ESEMPIO: `PointStack (Object)`

```
class PointStack{
   static const int size = 10;
   Point *data[size];
   int index;
public:
   PointStack(): index(-1){}
   ~PointStack(){ clear();}
   void push(Point *p){
        assert(index<(size-1));
        data[++index]=p;
   }
   Point * pop(){
        assert(index>=0);
        return data[index--];
   }
   bool isEmpty(){return index==-1;}
   void clear(){
        while(!isEmpty())
                delete pop();
   }
};
#endif
```

```
#ifndef POINTSTACK_H
#define POINTSTACK_H
#include "Point.h"
#include <iostream>
#include <cassert>
using namespace std;
```


ESEMPIO: `PointStack (Object)`

```
#include "PointStack.h"
#include "Point.h"
#include <iostream>
using namespace std;

void test(){
   PointStack st;
   cout<<boolalpha;
   cout<<st.isEmpty()<<endl;
   for(int i=0; i<10; i++)
        st.push(new Point(i,i));
   cout<<st.isEmpty()<<endl;
   for(int i=0; i<10; i++){
        Point *p = st.pop();
        cout<<*p;
        delete p;
   }
   cout<<st.isEmpty()<<endl;
}

int main(){ test();}
```

Una possibile uscita

```
true
false
(9,9)
(8,8)
(7,7)
(6,6)
(5,5)
(4,4)
(3,3)
(2,2)
(1,1)
(0,0)
true
```


ESEMPIO: `PointStack (Object)`

- In particolare, il seguente codice evidenzia la *corretta invocazione dei distruttori* degli oggetti `Point` da parte del contenitore `PointStack`:

```
void test2(){
   PointStack st;
   cout<<st.isEmpty()<<endl;
   for(int i=0; i<4; i++)
        st.push(new Point);
   cout<<st.isEmpty()<<endl;
   st.clear();
   cout<<st.isEmpty()<<endl;
}
int main(){ test2();}
```

Una possibile uscita

```
1
Point(0,0)
Point(0,0)
Point(0,0)
Point(0,0)
0
~Point()
~Point()
~Point()
~Point()
1
```


ESEMPIO: `Object`

- Per rendere generico tale contenitore non si può usare un puntatore `void`, perché non sarebbe corretto usare `delete` nella sua implementazione (non è un tipo specifico). Il compito della distruzione degli oggetti dovrebbe essere lasciato all'utente. Ciò fa nascere il cosiddetto "*ownership problem*".
- Alcuni linguaggi (per esempio Java) risolvono il problema utilizzando il *polimorfismo*: tutti gli oggetti sono *derivati* da una classe comune denominata `Object` (*singly-rooted hierarchy*): utilizzando puntatori alla classe base è possibile distruggere correttamente gli oggetti (distruttore virtuale).

ESEMPIO: `Object`

- Vediamo come implementare un *comportamento come quello* di Java in C++.
- Si implementa una semplice classe `Object` *astratta*.

```
#ifndef OBJECT_H
#define OBJECT_H

class Object{
public:
  virtual ~Object() = 0;
};
Object::~Object(){}
#endif
```


ESEMPIO: `Object`

- Si derivano tutte le classi da `Object`. Quindi anche `Point`:

```
#ifndef OPOINT_H
#define OPOINT_H

#include "Object.h"
#include <iostream>
using namespace std;

class OPoint: public Object{
//…
};
#endif
```

- Ora si può implementare il contenitore *stack* in modo *generico* sfruttando il *tipo base* `Object`.

ESEMPIO: `Object`

```
class OStack{
   static const int size = 10;
   Object *data[size];
   int index;
public:
   OStack(): index(-1){}
   ~OStack(){ clear();}
   void push(Object *p){
        assert(index<(size-1));
        data[++index]=p;
   }
   Object * pop(){
        assert(index>=0);
        return data[index--];
   }
   bool isEmpty(){return index==-1;}
   void clear(){
        while(!isEmpty())
                delete pop();
   }
};
#endif
```

```
#ifndef OSTACK_H
#define OSTACK_H
#include "OPoint.h"
#include <iostream>
#include <cassert>
using namespace std;
```


ESEMPIO: `Object`

```
#include "OStack.h"
#include "OPoint.h"
#include <iostream>
using namespace std;

void test(){
   OStack st;
   cout<<st.isEmpty()<<endl;
   for(int i=0; i<4; i++)
        st.push(new OPoint);
   cout<<st.isEmpty()<<endl;
   st.clear();
   cout<<st.isEmpty()<<endl;
}

int main(){ test();}
```

Una possibile uscita

```
1
OPoint(0,0)
OPoint(0,0)
OPoint(0,0)
OPoint(0,0)
0
~OPoint()
~OPoint()
~OPoint()
~OPoint()
1
```

SOMMARIO

- Template di funzione e template di classe:
  - Argomenti espliciti.
  - Requisiti sui tipi.
  - Specializzazione esplicita.
  - Overloading.
  - Algoritmi generici.
  - Parametri valore.
  - Funzioni membro.

TEMPLATE

- Per implementare un comportamento tipo Java in C++ (`Object`) si è utilizzato la soluzione nota come *singly-rooted hierarchy* o *object-based hierarchy*.
- Per ragioni di efficienza la soluzione del C++ è il meccanismo dei `template`. Tuttavia, è possibile utilizzare la soluzione vista per singoli progetti.

- Un template fornisce un meccanismo per *generare automaticamente* istanze particolari di funzioni o classi che si differenziano per il tipo. Il programmatore *parametrizza* tutti o una parte dei tipi dell'interfaccia.

TEMPLATE DI FUNZIONE

- Un *linguaggio fortemente tipizzato* può sembrare un ostacolo all'implementazione di funzioni altrimenti immediate. Ad esempio, una funzione per comparare due argomenti e restituire il minimo deve avere una implementazione diversa per ogni coppia di tipi:

```
int Min(int a, int b){
  return a < b ? a : b;
}

double Min(double a, double b){
  return a < b ? a : b;
}
```


TEMPLATE DI FUNZIONE

- Il meccanismo dei template permette di *parametrizzare* il tipo e creare automaticamente un'*istanza specifica* per ogni tipo.
- La parola chiave `template` è posta all'inizio sia di una definizione sia di una dichiarazione di funzione ed è seguita da una *lista di parametri* separati da virgole e racchiusi dai simboli < > .
- Un *parametro di tipo* consiste nella parola chiave `typename` (o `class`) seguita da un *identificatore* che rappresenta un tipo predefinito o definito dall'utente.

TEMPLATE DI FUNZIONE

- Quando il template è *istanziato*, un tipo effettivo è sostituito al parametro tipo del template.

```
template<typename T>
T Min(T a, T b){
   return a < b ? a : b;
}

int main(){
   int i=1, j=2;
   cout<<Min(i,j)<<endl;
   double x=2.3, y=2.5;
   cout<<Min(x,y)<<endl;;
}
```

Una possibile uscita

```
1
2.3
```


TEMPLATE DI FUNZIONE

- Un template di funzione specifica *come costruire* le singole funzioni dato un insieme di uno o più tipi effettivi.
- Questo processo di costruzione, detto *istanziazione (instantiation) del template*, avviene automaticamente alla chiamata del template funzione.
- Il processo di determinazione dei tipi degli argomenti di un template a partire da tipo di argomenti di funzione è chiamato *deduzione degli argomenti del template*.

TEMPLATE DI FUNZIONE

- Le parole chiave `typename` e `class` sono intercambiabili. L'identificatore del parametro di tipo è scelto dal programmatore. Il precedente esempio è equivalente al seguente:

```
template<class Tipo>
Tipo Min(Tipo a, Tipo b){
  return a < b ? a : b;
}

int main(){
  cout<<Min(1,2)<<endl;
  cout<<Min(2.3,2.5)<<endl;
}
```

Esempio precedente

```
template<typename T>
T Min(T a, T b){
        return a < b ? a : b
}
```


TEMPLATE DI FUNZIONE: argomenti espliciti

- Se la deduzione degli argomenti del template rileva due tipi diversi allora viene generato un errore:

```
int a=1;
short b=2;
//cout<<Min(a,b)<<endl;//errore
```

- In tali casi si può *specificare esplicitamente* il tipo degli argomenti. Si inseriscono i tipi, separati da virgole e racchiusi dai simboli < >, dopo il nome della funzione:

TEMPLATE DI FUNZIONE: requisiti sui tipi

- L'implementazione vista introduce un *requisito nascosto* sui tipi con cui viene istanziato il template: il tipo usato deve essere in grado di usare *l'operatore predefinito minore-di* oppure deve essere una classe che implementa `operator<()`.

```
string s1("primo"), s2("secondo");
cout<<Min(s1,s2)<<endl;
cout<<"--------------------"<<endl;
Point p1, p2(1,1);
//cout<<Min(p1,p2)<<endl;//errore
```

Una possibile uscita

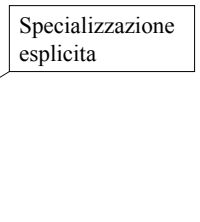
```
primo
--------------------
```


TEMPLATE DI FUNZIONE: specializzazione esplicita

- Non sempre si può utilizzare un solo template per qualsiasi tipo: la conoscenza del tipo può essere utile a scrivere una funzione più *efficiente* o con il *significato corretto*.
- Per esempio, il confronto di due stringhe stile C, `const char *`, è fatto sui puntatori e non sulle stringhe.
- Se si desidera un confronto tra stringhe, è necessario scrivere una *definizione esplicita di specializzazione* per il template.

TEMPLATE DI FUNZIONE: specializzazione esplicita

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

template<typename T>
T Min(T a, T b){
   return a < b ? a : b;
}

template<> const char * Min<const char *>(const char * a, const char *b){
   return strcmp(a, b) ? a : b;
}

int main(){
   string s1("primo"), s2("secondo");
   cout<<Min(s1,s2)<<endl;
   cout<<"--------------"<<endl;
   cout<<Min("primo","secondo")<<endl;

}
```



Una possibile uscita

```
primo
--------------
primo
```


TEMPLATE DI FUNZIONE: sovraccaricamento

- In generale, può essere necessario *sovraccaricare il template* di funzione affinché produca il risultato desiderato: per esempio, differenziare la funzione template in relazione al fatto che l'argomento del template sia un puntatore o non lo sia. Il template di funzione `T Min(T a, T b)` produce il risultato confrontando gli indirizzi.

```
int a=5, b=1;
cout<<Min(a,b)<<endl;
cout<<"-----------------"<<endl;
int *p1=&a, *p2=&b;
cout<<Min(p1,p2)<<endl;
```

Una possibile uscita

```
1
-----------------
0012FF54
```

TEMPLATE DI FUNZIONE: sovraccaricamento

```
template<typename T>
T Min(T a, T b){
   return a < b ? a : b;
}

template<typename T>
T Min(T *a, T *b){
   return *a < *b ? *a : *b;
}

int main(){
   int a=5, b=1;
   cout<<Min(a,b)<<endl;
   cout<<"-----------------"<<endl;
   int *p1=&a, *p2=&b;
   cout<<Min(p1,p2)<<endl;
}
```

Overloading del template di funzione

Una possibile uscita

```
1
-----------------
1
```


TEMPLATE DI FUNZIONE: algoritmo generico

- Utilizzando il meccanismo dei template è possibile implementare un *algoritmo generico*.
- Per esempio, si era preso in considerazione un algoritmo per la ricerca di un valore all'interno di un contenitore sequenziale non ordinato.
- Per ottenere un *comportamento generico* si deve poter percorre linearmente tutti gli elementi senza conoscere la struttura dati sottostante: a tal fine si sfrutta il comportamento (*interfaccia*) comune degli *iteratori*.

TEMPLATE DI FUNZIONE: algoritmo generico

- Il limite era che l'implementazione era fatta per *tipi specifici*, per esempio

```
FloatArray::Iterator   trova(FloatArray::Iterator   first,
                    FloatArray::Iterator last,float val){
```

- Quindi non si può utilizzare con un `vector` o, in generale, con un tipo diverso da `FloatArray`. Il meccanismo dei template permette di superare questo limite, si consideri la seguente funzione:

```
template<typename IT, typename V>
IT trova(IT first, IT last,V val){
      for( ; first!=last; ++first)
            if(val == *first)
                  return first;
      return last;}
```


TEMPLATE DI FUNZIONE: algoritmo generico

```
int main(){
   FloatArray a(10,0);
   vector<int> v(10);

   for(int i=0;i<10;i++){
        a[i]=i*i;
        v[i]=i*i;
   }
   cout<<a;

   FloatArray::Iterator out = trova(a.begin(), a.end(),25);

   if(out != a.end())
        cout<<"Trovato"<<endl;
   cout<<"---------------"<<endl;

   vector<int>::iterator outv = trova(v.begin(), v.end(),25);

   if(outv != v.end()) cout<<"Trovato"<<endl;

}
```

Una possibile uscita

```
0 1 4 9 16 25 36 49 64 81
Trovato
------------------------
Trovato
```

Template di funzione

TEMPLATE DI FUNZIONE

- La definizione di un template deve essere presente per permetterne l'istanziazione: cioè deve essere definito in file *header*.
- Il modello di compilazione per separazione (parola chiave `export`) non è supportato da tutti i compilatori.

- Oltre ai parametri di tipo esistono i *parametri valori*: i parametri valori sono normali dichiarazioni e indicano che il nome del parametro rappresenta un *valore costante* nella definizione di template.

TEMPLATE DI CLASSE

- Un *template di classe* è un *modello* per creare una classe in cui uno o più tipi o valori sono *parametrizzati*.
- Come per i template di funzione si ha: parametro tipo, parametro valore, istanziazione del template.
- I *parametri valore* rappresentano delle *costanti* nella definizione del template di classe: possono rappresentare la dimensione di array.
- I parametri di un template di classe possono avere degli *argomenti di default*.

TEMPLATE DI CLASSE

- Al contrario degli argomenti dei template di funzione, gli *argomenti* dei template di *classe non* sono mai *dedotti* dal contesto in cui si usano, pertanto il *nome* di un'istanza di un template di classe deve sempre specificare esplicitamente gli argomenti (per esempio, `vector<int>`).
- Con riferimento all'esempio visto di *singly-rooted hierarchy* (comportamento tipo Java), in C++ si rende *generico* il contenitore *stack* utilizzando il *meccanismo dei template* di classe e non il polimorfismo.

TEMPLATE DI CLASSE: esempio

```
template<typename T, int size = 10>
class Stack{
   T *data[size];
   int index;
public:
   Stack(): index(-1){}
   ~Stack(){ clear();}
   void push(T *p){
        assert(index<(size-1));
        data[++index]=p;
   }
   T * pop(){
        assert(index>=0);
        return data[index--];
   }
   bool isEmpty(){return index==-1;}
   void clear(){
        while(!isEmpty())
                delete pop();
   }
};
```

Parametro valore con argomento di default

Tipo generico

Quando il template è istanziato per un tipo specifico, si può correttamente applicare `delete`.

TEMPLATE DI CLASSE : esempio

```
#include "Stack.h"
#include "Point.h"
#include <iostream>
using namespace std;

void test1(){
   Stack<Point,5> st;

   cout<<boolalpha;
   cout<<st.isEmpty()<<endl;
   for(int i=0; i<5; i++)
        st.push(new Point(i,i));
   cout<<st.isEmpty()<<endl;

   st.clear();
   cout<<st.isEmpty()<<endl;
}

int main(){ test1(); }
```

Una possibile uscita

```
true
Point(0,0)
Point(1,1)
Point(2,2)
Point(3,3)
Point(4,4)
false
~Point()
~Point()
~Point()
~Point()
~Point()
true
```


TEMPLATE DI CLASSE : esempio

- L'istruzione `Stack<Point,5> st;` provoca l'istanziazione di uno *stack specifico* (di 5 elementi di tipo `Point`) a partire dal *template generico*.
- In particolare, il precedente codice evidenzia la *corretta invocazione dei distruttori* degli oggetti `Point` da parte del contenitore generico `Stack`.
- Il contenitore è in grado di gestire anche i tipi predefiniti: l'esempio seguente istanzia uno *stack* di 10 elementi di tipo intero (`Stack<int> st;`).

TEMPLATE DI CLASSE : esempio

```
void test2(){
   Stack<int> st;

   cout<<boolalpha;
   cout<<st.isEmpty()<<endl;

   for(int i=0; i<10; i++)
        st.push(new int(i));

   cout<<st.isEmpty()<<endl;

   for(int i=0; i<10; i++){
        int *p = st.pop();
        cout<<*p<<endl;
        delete p;
   }
   cout<<st.isEmpty()<<endl;
}

int main(){ test2(); }
```

Una possibile uscita

```
true
false
9
8
7
6
5
4
3
2
1
0
true
```


TEMPLATE DI CLASSE: funzioni membro

- Ogni occorrenza del nome del template di classe `Stack` nella definizione del template è una notazione abbreviata per `Stack<T,size>`. La notazione abbreviata può essere usata solo nella definizione del template, all'esterno si deve usare il nome completo.
- Vediamo come definire una funzione membro all'esterno del template.
- In particolare, una funzione membro di un template di classe è essa stessa un template.

TEMPLATE DI CLASSE : funzioni membro

- Per esempio, si consideri di definire le funzioni membro `pop()` e `push()` all'esterno del template, invece che internamente come visto nell'esempio precedente.

```
template<typename T, int size>
//void Stack::push(T *p){//errore
void Stack<T,size>::push(T *p){
          assert(index<(size-1));
          data[++index]=p;
}

template<typename T, int size>
T * Stack<T,size>::pop(){
          assert(index>=0);
          return data[index--];
}
```


TEMPLATE DI CLASSE

- Un template di classe o di funzione può essere membro di una classe ordinaria o di un template di classe.
- Quindi è possibile implementare un iteratore generico.
- Tuttavia, per essere compatibile con la Libreria Standard deve essere un iteratore derivato da iteratori standard contenuti in `<iterator>`.

TEMPLATE DI CLASSE

- In generale, valgono le considerazioni fatte per i template di funzione a proposito dei *requisiti nascosti sui tipi* dei parametri e sulla *specializzazione esplicita*.
- Per esempio, poiché i *tipi* possono essere *classi* è preferibile (i) utilizzare i riferimenti come parametri dei membri funzione e (ii) utilizzare la lista di inizializzazione dei membri del costruttore.
- La *specializzazione esplicita o parziale* va usata quando non si può utilizzare un solo template per qualsiasi tipo: per esempio, differenziare la classe template in relazione al fatto che l'argomento del template sia un puntatore o non lo sia (cambia l'implementazione).

ECCEZIONI

- La gestione delle eccezioni è un meccanismo che permette a due componenti separate di un programma di *comunicare* quando viene rilevata durante l'esecuzione un'*anomalia*. Tale comunicazione avviene con una *modalità standard*.
- Attraverso l'uso dei blocchi `try` (insieme delle istruzioni che possono lanciare un'eccezione) e delle clausole `catch` (lista di procedure per la gestione delle eccezioni) è possibile mantenere *separato* il codice che gestisce le anomalie da quello di normale elaborazione.

ECCEZIONI

- Utilizzare le eccezioni è una *scelta progettuale*, non tutti i programmi devono farne uso: dovrebbero essere usate per comunicare anomalie di un programma fra parti sviluppate indipendentemente, perché sollevare un'eccezione *non* è un'operazione *veloce*.
- Normalmente i programmi dovrebbero usare altre tecniche più appropriate di gestione delle anomalie e risolverle *localmente*.
- Principalmente si utilizzano le eccezioni quando si implementano sistemi ad alto grado di affidabilità.

ECCEZIONI: ESEMPIO

```
class MyErr{
   string msg;
public:
   MyErr(string str):msg(str){cout<<"MyErr()"<<endl;}
   MyErr(const  MyErr  &o){cout<<"MyErr(const  Myerr  &)"<<endl;
                                              msg=o.msg;}
   ~MyErr(){cout<<"~MyErr()"<<endl;}
   void Get()const {cout<<msg<<endl;}
};

class Prova{
public:
   Prova(){cout<<"Prova()"<<endl;}
   ~Prova(){cout<<"~Prova()"<<endl;}
   void Metodo() const{
        cout<<"Metodo()"<<endl;
        throw MyErr("Eccezione!");
        cout<<"Fine Metodo()"<<endl;
   }
};
```


ECCEZIONI: ESEMPIO

```
int main(){
   cout<<"----------------"<<endl;
   try{
        Prova p;
        cout<<"---prima"<<endl;
        p.Metodo();
        cout<<"---dopo"<<endl;
   }
   catch(MyErr &e){
        cout<<"---catch"<<endl;
        e.Get();
   }
   cout<<"----------------"<<endl;
}
```

Una possibile uscita

```
----------------------
Prova()
---prima
Metodo()
MyErr()
MyErr(const Myerr &)
~MyErr()
~Prova()
---catch
Eccezione!
~MyErr()
----------------------
```

SROTOLAMENTO DELLA PILA

- L'espressione `throw` crea un *oggetto temporaneo* della classe che utilizziamo per memorizzare *informazioni* relative all'anomalia. Un corrispondente *oggetto eccezione* è passato alla procedura di gestione dell'eccezione.
- La ricerca delle clausole `catch` avviene nella funzione che lancia l'eccezione e se non vengono trovate continua nella funzione chiamante.
- Tale processo si chiama *srotolamento della pila* (*stack unwinding*).
- Se un'eccezione non viene intercettata il programma termina in modo *anomalo*.

SROTOLAMENTO DELLA PILA

- Durante il processo di *srotolamento della pila* è garantito che vengano chiamati i *distruttori* per gli oggetti locali (utilità della tecnica: "l'acquisizione di una risorsa è l'inizializzazione; il rilascio di una risorsa è la distruzione").
- Altrimenti è possibile eseguire qualche azione prima di uscire a causa di un'eccezione, anche se non si può gestire l'eccezione stessa, attraverso l'uso di una clausola generica:

```
catch(…){
  //rilascio risorse
  throw; //rilancio
}
```


SPECIFICAZIONE DI ECCEZIONE

- È possibile specificare quale tipo di eccezioni una funzione può sollevare attraverso l'uso di una *specificazione di eccezione* (solo accettata da .NET): dopo la lista dei parametri va inserita la parola chiave `throw` seguita da una lista di tipi di eccezioni racchiusa fra parentesi.
- La *specificazione di eccezione* garantisce che la funzione non lanci altri generi di eccezioni.
- Per indicare che una funzione non lancia eccezioni si utilizza una *specificazione di eccezione* vuota: per esempio, `void funzione() throw();`.

ECCEZIONI: NOTE

- La libreria standard fornisce una gerarchia di classi eccezione che possono essere usate direttamente o ulteriormente derivate. La classe radice nella gerarchia delle eccezioni della libreria standard è chiamata `exception` ed è definita nel file header `<exception>`.

Note:

- Non utilizzare il meccanismo delle eccezioni come un'alternativa a normali strutture di controllo.
- Non utilizzare le eccezioni nei distruttori.
- Si possono utilizzare le eccezioni nei costruttori.
- Un'eccezione che è istanza di una classe può essere intercettata da una clausola `catch` riferita ad una sua classe base pubblica.

Programmazione Windows con Microsoft Foundation Classes

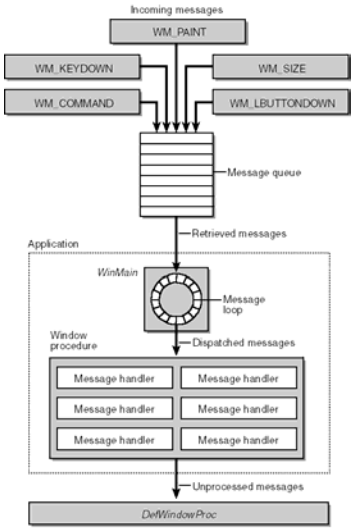
Programmazione Windows

- Le applicazioni Windows operano utilizzando un modello di programmazione “event-driven”.
- Si gestisce la reazione del programma di fronte a eventi (pressione del mouse, di un tasto, chiusura di una finestra,...) che vengono segnalati all’applicazione tramite messaggi inviati dal S.O.
- I messaggi diretti ad una applicazione vengono accodati in attesa dell’elaborazione, elaborati e cosi’ via fino alla richiesta di terminazione del programma (`WM_QUIT`).

Programmazione Windows

- Un’applicazione grafica in ambiente Windows può essere creata effettuando chiamate alle API del sistema operativo da un programma C/C++.
- L’entry-point di un’applicazione Windows è la funzione `WinMain`, al suo interno vi è il loop dei messaggi che riceve e consegna i messaggi per la procedure window.
- La procedure window processa i messaggi attraverso uno switch.
- I messaggi sono contenuti in una coda fino a che non vengono processati.
- Il loop dei messaggi (e quindi l’applicazione) termina con il messaggio `WM_QUIT`

Programmazione Windows

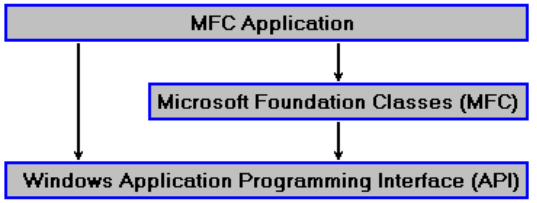


MFC - Introduzione

- MFC (Microsoft Foundation Classes) e' una libreria nata per semplificare il compito di gestire finestre e eventi di Windows incapsulando porzioni delle API di Windows in classi C++.
- Anziché effettuare chiamate dirette alle API di Windows è possibile creare oggetti di queste classi ed effettuare chiamate alle loro funzioni membro.
- MFC contiene inoltre librerie utili alla gestione di database, alla comunicazione fra processi, alla gestione della memoria, dei threads, ecc.

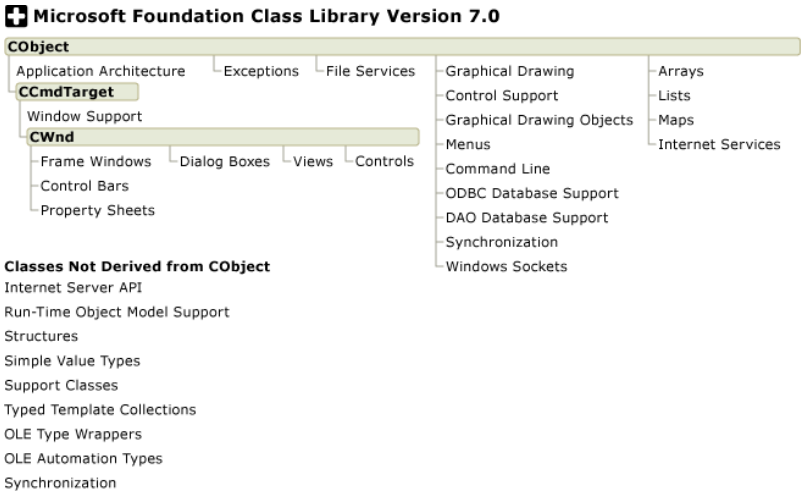
MFC - Introduzione

- Da un'applicazione MFC è comunque possibile effettuare chiamate dirette alle API di Windows



- MFC è organizzata come un albero gerarchico di classi derivate. La maggior parte delle classi deriva da `CObject`.

MFC - Introduzione



MFC - Introduzione

- Gli ambienti di sviluppo Visual Studo e .NET mettono a disposizione uno strumento wizard per la creazione di applicazioni MFC
- Le applicazioni MFC possono essere
  - Dialog based
  - Single Document Interface (SDI)
  - Multiple Document Interface (MDI)
- Gli esempi seguenti sono basati su un'applicazione MFC Dialog Based

MFC – Dialog based

Utilizzando il wizard .NET genera in automatico 2 classi

- Application Class: *CLezioneMFCApp*
- Main Dialog Class: *CLezioneMFCDlg*

Application Class

- Rappresenta l'applicazione MFC stessa e il suo thread principale.
- Deriva da `CWinApp` (è necessario includere "Afxwin.h") che a sua volta deriva da CWinThread.
- Fornisce il loop di messaggi che riceve e distribuisce i messaggi alla finestra dell'applicazione.
- Un'applicazione MFC ha un (e uno solo) oggetto applicazione. Il wizard crea quindi un oggetto globale (derivato da `CWinApp`) chiamato `theApp` che esiste per tutto il tempo di esecuzione del programma.

Application Class

```
class CLezioneMFCApp : public CWinApp
{
public:
	CLezioneMFCApp();

// Overrides
	public:
	virtual BOOL InitInstance();

// Implementation

	DECLARE_MESSAGE_MAP()
};
```


Application Class – InitInstance()

- L'override del metodo `InitInstance()` è obbligatorio, in quanto esso è richiamato per creare l'applicazione. Esso torna TRUE se l'applicazione è creata con successo altrimenti ritorna FALSE.
- In un'applicazione MFC dialog-based nel metodo `InitInstance()` è istanziato (e reso visibile) un oggetto della classe dialog principale.
- L'override del metodo `ExitInstance()` è necessario per deallocare risorse eventualmente allocate durante l'inizializzazione dell'applicazione.

Application Class – InitInstance()

```
BOOL CLezioneMFCApp::InitInstance(){
	...
	CLezioneMFCDlg dlg;
	m_pMainWnd = &dlg;
	INT_PTR nResponse = dlg.DoModal();
	if (nResponse == IDOK){

	}
	else if (nResponse == IDCANCEL){

	}
	return FALSE;
}
CLezioneMFCApp::CLezioneMFCApp()
{
}

CLezioneMFCApp theApp;
```


Application Class – Run()

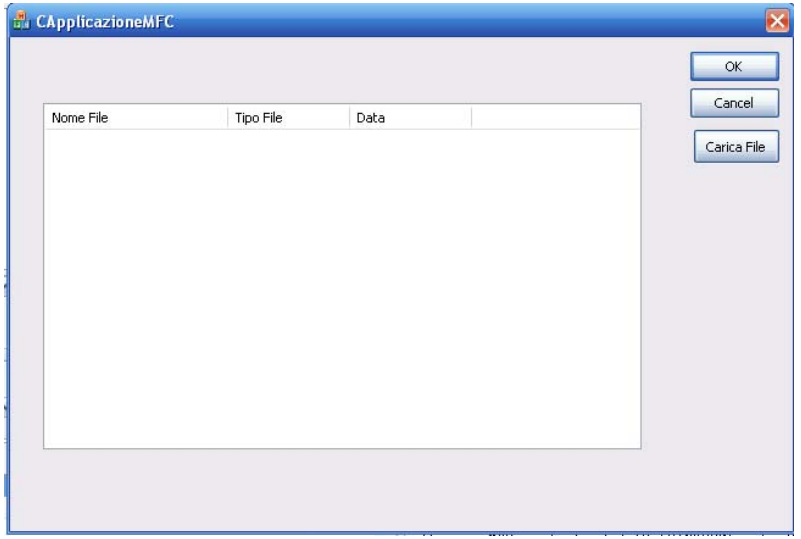
Il metodo virtuale `Run()` si occupa di ricevere i messaggi dalla coda dei messaggi dell'applicazione e processarli fino a che non viene ricevuto il messaggio `WM_QUIT`.

- se la coda dei messaggi dell'applicazione non contiene alcun messaggio `Run()` chiama `OnIdle()`.
- non appena arriva un messaggio esso viene pre processato dalla funzione `PreTranslateMessage`, dopodiché viene chiamata la funzione Windows `TranslateMessage` ed infine la funzione `DispatchMessage`.

Classe CDialog

- E' la classe usata per mostrare dialog boxes sullo schermo.
- Deriva dalla classe `CWnd`.
- Può essere modal o non-modal; nel primo caso l'esecuzione della funzione chiamante rimane sospesa fino a che la dialog non viene chiusa, nel secondo caso i due task procedono in parallelo.
- Una dialog riceve messaggi da Windows e dai controlli presenti in essa.

Dialog



Dialog

- Alla Dialog è assegnato un identificatore: IDD_LEZIONEMFC_DIALOG
- Ad essa è associata la classe *CLezioneMFCDlg.*
- Nella dialog sono inseriti i seguenti controlli:
  - 1 List Control IDC_LISTAFILE a cui è associata la variabile CListCtrl m_listafile;
  - 3 Button Control a cui sono associate variabili della classe CButton

Dialog

- La classe *CLezioneMFCDlg,* generata in automatico dal wizard, e tutte le classi delle dialog presenti nelle applicazioni MFC, sono derivate da CDialog.
- Il metodo `virtual void DoDataExchange(CDataExchange* pDX);` gestisce la relazione tra i controlli presenti nella dialog e i corrispondenti membri della classe.
- Il metodo virtual `BOOL OnInitDialog()` è usato per effettuare le inizializzazioni della dialog.

Dialog

```
class CLezioneMFCDlg : public CDialog{
public:
	CLezioneMFCDlg(CWnd* pParent = NULL);
	enum { IDD = IDD_LEZIONEMFC_DIALOG };
protected:
	virtual void DoDataExchange(CDataExchange* pDX);
	HICON m_hIcon;
	virtual BOOL OnInitDialog();
	afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	DECLARE_MESSAGE_MAP()
public:
	CButton m_button;
	afx_msg void OnBnClickedFile();
	CListCtrl m_listafile;
	afx_msg void OnNMDblclkListafile(NMHDR *pNMHDR,
					LRESULT *pResult);
};
```


Dialog

```
CLezioneMFCDlg::CLezioneMFCDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CLezioneMFCDlg::IDD, pParent)
{
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

```
void CLezioneMFCDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	DDX_Control(pDX, IDC_FILE, m_button);
	DDX_Control(pDX, IDC_LISTAFILE, m_listafile);
}
```

Dialog

```
BOOL CLezioneMFCDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  //altre inizializzazioni del wizard
  ...

  //ulteriori  inizializzazioni
  int nItem=0;
  m_listafile.InsertColumn(nItem++,L"Nome File",LVCFMT_LEFT,150);
  m_listafile.InsertColumn(nItem++,L"Tipo File",LVCFMT_LEFT,100);
  m_listafile.InsertColumn(nItem,L"Data", LVCFMT_LEFT,100);

  return TRUE;
}
```


Message Map

- Il loop dei messaggi e le funzioni di callback sono stati incapsulate all'interno di MFC.
- La macro DECLARE_MESSAGE_MAP() deve essere inserita alla fine della definizione della classe di cui si devono gestire i messaggi.
- Nell'implementazione della classe è necessario utilizzare le macro BEGIN_MESSAGE_MAP e END_MESSAGE_MAP per specificare quali saranno i messaggi gestiti dall'applicazione.
- I messaggi non gestiti dalla classe verranno gestiti dalle classi a livello superiore nella gerarchia.

Message Map

- La macro BEGIN_MESSAGE_MAP ha due argomenti: il nome della classe e di quella da cui essa deriva.
- Tra le due macro viene inserito l'elenco dei messaggi che verranno gestiti dall'applicazione.
- E' possibile inserire una entry per ogni messaggio che vogliamo gestire nella dialog (ad esempio click su un button, doppio click in una list control, ecc..)
- La message map è essenzialmente una lookup table che mette in relazione un messaggio con una funzione membro di una classe.

Message Map - Esempio

La message map della classe *CLezioneMFCDlg* è:

```
BEGIN_MESSAGE_MAP(CLezioneMFCDlg, CDialog)
  ON_WM_SYSCOMMAND()
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  //}}AFX_MSG_MAP
  ON_BN_CLICKED(IDC_FILE, &CLezioneMFCDlg::OnBnClickedButton1)
  ON_NOTIFY(NM_DBLCLK, IDC_LISTAFILE,
                          &CLezioneMFCDlg::OnNMDblclkListafile)
END_MESSAGE_MAP()
```

Message Map

- E' stata definita una firma standard per i metodi della classe che gestiscono i messaggi. Le firme possono essere trovate nel file `AFXWIN.H` e nella dichiarazione di `CWnd`.
- La dichiarazione di queste funzioni è preceduta dalla parola **afx_msg.**

```
ON_BN_CLICKED(IDC_FILE, &CLezioneMFCDlg::OnBnClickedButton1)

afx_msg void OnBnClickedFile();
```


Message Map

- Dato che ogni dialog box deriva da CDialog, molti comportamenti vengono ereditati dalla classe CDialog (ad esempio la chiusura della finestra premendo sulla x in alto a destra)
- E' comunque possibile prevedere comportamenti specifici, definendo nella message map l'opportuna entry relativa al messaggio che si vuole intercettare.
- Ad esempio se si vuole definire un comportamento personalizzato alla chiusura della finestra, si aggiungerà `ON_WM_CLOSE()`

Message Map

- Il metodo `OnClose()` verrà chiamato alla generazione di un messaggio di chiusura.

```
void CLezioneMFCDlg::OnClose()
{
   if(IDYES == MessageBox(_T("Sei sicuro di voler
      chiudere?"),_T("Attenzione"),MB_ICONQUESTION | MB_YESNO))

            CDialog::OnClose();
}
```

MFC - Esempio

- MFC Application – Dialog based
- Nella Dialog principale sono presenti
  - 1 ListCtrl (`IDC_LISTAFILE`)
  - 1 Button (`IDC_FILE`)
- Si vogliono gestire i seguenti eventi:
  - `WM_CLOSE` per chiedere conferma della chiusura
  - `ON_BN_CLICKED(IDC_FILE, &CLezioneMFCDlg::OnBnClickedButton1)`
  - `ON_NOTIFY(NM_DBLCLK, IDC_LISTAFILE, &CLezioneMFCDlg::OnNMDblclkListafile)`

MFC - Esempio

- Per quanto riguarda l'evento `ON_BN_CLICKED` si vuole riempire la ListCtrl con l'elenco dei files presenti in una determinata directory (es. c:\temp)
- Classe `CFileFind`: utile per eseguire ricerche all'interno del file-system e per ottenere informazioni sui files (nome, data di modifica,...)

MFC - Esempio

- Per quanto riguarda l'evento `NM_DBLCLK` si prevedono due diversi comportamenti:
  - Doppio-click su un file di tipo PGM
  - Doppio-click su un altro tipo di file
- Nel primo caso verrà creato un oggetto della classe `CMyImage` (vedi esercitazioni) e alcune informazioni relative all'oggetto verranno visualizzate in una Dialog.
- Nel secondo caso verrà eseguita l'applicazione Windows associata con il file selezionato.

MFC - Esempio

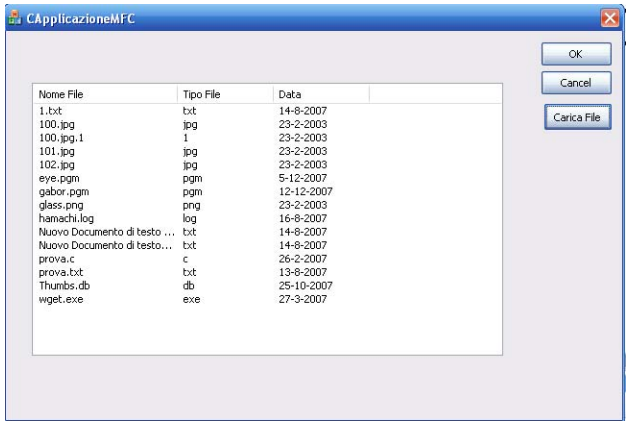
```
void CLezioneMFCDlg::OnBnClickedButton1()
{
  CFileFind mFileFind;
  LPCTSTR   Path=_T("c:\\temp\\*");
  TCHAR dummy[256];
  CString nomefile;
  CString estensione;
  CTime tmod;
  m_listafile.DeleteAllItems();
  int nItem = 0;
  bool result = true;

       ...
```


MFC - Esempio

```
...
if(mFileFind.FindFile(Path)){
  while(result){
  result=mFileFind.FindNextFile();
  if(!mFileFind.IsDirectory()){
    nomefile=mFileFind.GetFileName();
    mFileFind.GetLastWriteTime(tmod);
    m_listafile.InsertItem(nItem, nomefile.GetBuffer(256));
    m_listafile.SetItemText(nItem,0, nomefile.GetBuffer(256));
    estensione=nomefile.Right(nomefile.GetLength()-
                                nomefile.ReverseFind('.')-1);
    m_listafile.SetItemText(nItem,1, estensione.GetBuffer(256));
   wsprintf(dummy,_T("%d-%d-%d"), tmod.GetDay(),tmod.GetMonth(),
                                                tmod.GetYear());
    m_listafile.SetItemText(nItem,2, dummy);
    nItem++; }}}
```

MFC - Esempio

Dopo la pressione del Button “Carica File” si avrà:



MFC - Esempio

- E’ necessario creare una nuova Dialog, i controlli necessari e la classe `CMyImageDlg` ad essa associata.
- E’ necessario passare le informazioni opportune tra una classe e l’altra.
- Ad esempio il nome del file selezionato nella Dialog principale dovrà essere utilizzato nella Dialog MyImage per poter creare l’oggetto di tipo `CMyImage`.

MFC - Esempio

MFC - Esempio

```
class CMyImageDlg : public CDialog{
private:
   CMyImage* PGMimage;
   CString nomePGM;
public:
      ...
   void SetImage(CString nome){nomePGM=nome;}
protected:
   virtual void DoDataExchange(CDataExchange* pDX);
   DECLARE_MESSAGE_MAP()
public:
   CEdit m_nomefile;
   CEdit m_width;
   CEdit m_height;
   CEdit m_maxlevel;
   virtual BOOL OnInitDialog();
   afx_msg void OnClose();
};
```

override del metodo OnInitDialog()

MFC - Esempio

```
void CLezioneMFCDlg::OnNMDblclkListafile(NMHDR *pNMHDR, LRESULT
                                                    *pResult){
      CString nomefile; CString estensione;
      int pos; CString Path("c:\\temp\\");
      POSITION sel=m_listafile.GetFirstSelectedItemPosition();
      if(sel){
         pos=m_listafile.GetNextSelectedItem(sel);
         nomefile=m_listafile.GetItemText(pos,0);
         nomefile = Path + nomefile;
         if(m_listafile.GetItemText(pos,1)=="pgm"){
             CMyImageDlg PGMdlg;
             PGMdlg.SetImage(nomefile);
             PGMdlg.DoModal();
         }
      ...
```


MFC - Esempio

```
...
else

      ShellExecute(0,_T("Open"),nomefile,0,0,SW_SHOW);
      }
      else
        MessageBox(_T("Nessun File Selezionato",
                                  T("Errore"),MB_ICONSTOP);
      *pResult = 0;
}
```


MFC - Esempio

```
BOOL CMyImageDlg::OnInitDialog()
{
       CDialog::OnInitDialog();
       char nome[256];
       wcstombs(nome, nomePGM,256);
       PGMimage = new CMyImage(nome);

       TCHAR dummy[256];
       wsprintf(dummy,_T("%d"),PGMimage->GetMaxVal());
       m_maxlevel.SetWindowTextW(dummy);
       wsprintf(dummy,_T("%d"),PGMimage->GetRow());
       m_height.SetWindowTextW(dummy);
       wsprintf(dummy,_T("%d"),PGMimage->GetCol());
       m_width.SetWindowTextW(dummy);
       m_nomefile.SetWindowTextW(nomePGM);

       return TRUE;
}
```


MFC - Esempio

- Dato che nel metodo `OnInitDialog()` è stato creato un oggetto `CMyImage` è necessario distruggerlo alla chiusura della dialog. Pertanto si intercetta il messaggio `WM_CLOSE`

```
void CMyImageDlg::OnClose()
{

       delete PGMimage;
       CDialog::OnClose();
}
```

MFC - Conclusioni

- Sfruttando il meccanismo dell'ereditarietà è possibile derivare dalle classi contenute nella libreria MFC per ottenere comportamenti personalizzati.
- E' possibile inserire un messaggio nella coda dei messaggi con la funzione PostMessage().

```
PostMessage(WM_CLOSE,0,0);
```


Single Document Interface

- Un'applicazione SDI è composta da:
  - Application class, derivata da `CWinApp` (analogamente a quanto visto per le applicazioni dialog-based).
  - `CMainFrame`
  - Classe Document, derivata `CDocument`
  - Classe View, derivata da `CView`

Single Document Interface

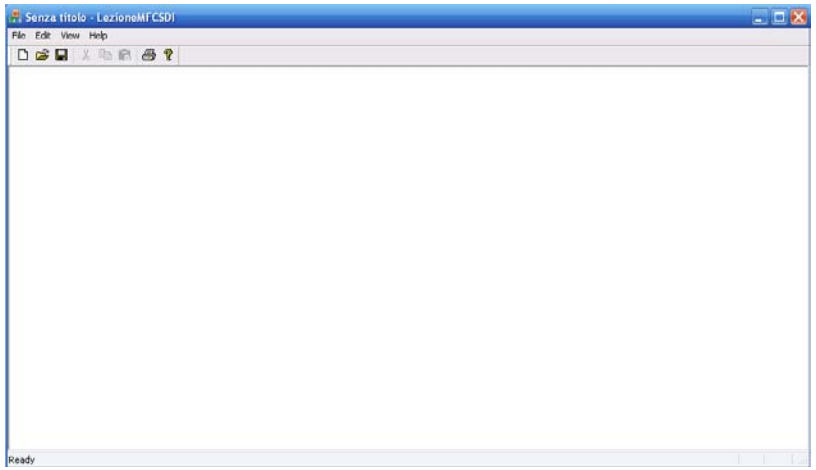
- La classe `CDocument` fornisce il supporto per le operazioni standard, come creare, caricare e salvare i documenti.
- `CMainFrame` è derivata da `CFrameWnd`. E' la classe che mantiene le variabili utili all'applicazione, implementa la message-map.
- La classe "view" derivata da CView è direttamente associata a un Document e agisce come intermediario tra il Document e l'utente.

Single Document Interface

- La relazione tra la classe view, la classe frame window e la classe document è stabilita da un oggetto della classe `CDocTemplate`:

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
       IDR_MAINFRAME,
       RUNTIME_CLASS(CLezioneMFCSDIDoc),
       RUNTIME_CLASS(CMainFrame),
       RUNTIME_CLASS(CLezioneMFCSDIView));
```

Single Document Interface



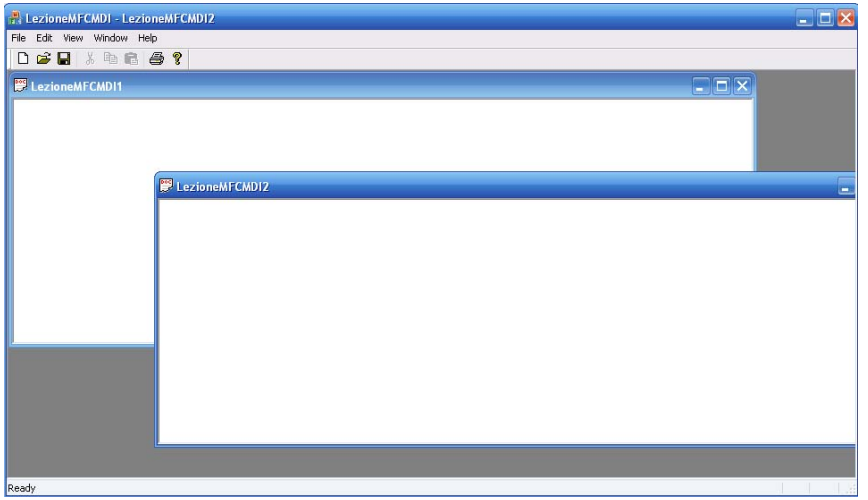
Multiple Document Interface

- Un'applicazione MDI è composta da:
  - Application class, derivata da `CWinApp`
  - `CMainFrame`
  - Classe Document, derivata `CDocument`
  - Classe View, derivata da `CView`
  - Classe CChildFrame, derivata da `CMDIChildWnd`

Multiple Document Interface

- Un'applicazione MDI conterrà una frame window "padre", e al suo interno potranno essere create più MDI child window.
- Una MDI child window è del tutto simile a una frame windows, a parte il fatto che deve essere contenuta in una MDI frame window.
- Non possiede funzionalità quali un suo menu, ma le condivide con la MDI frame window.
- Il framework cambia automaticamente il menu del MDI frame per rappresentare la MDI child window correntemente attiva.

Multiple Document Interface



Librerie multipiattaforma

- Esistono librerie per sviluppare applicazioni grafiche multipiattaforma. Ad esempio:
  - Qt
  - GTK
  - wxWindows
  - Java?
- Forniscono insiemi di classi (non solo C++) per gestire l'interfaccia grafica, gli eventi, i threads, applicazioni grafiche 3D, accesso a database).