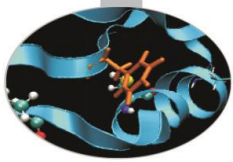
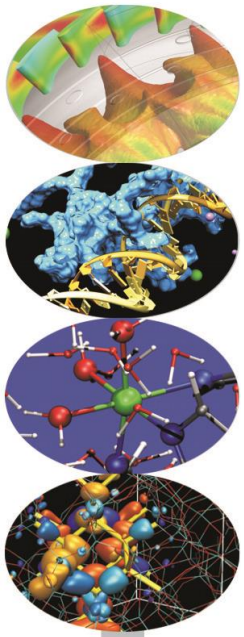
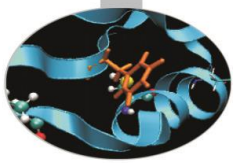


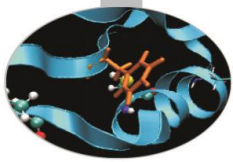
Overloading degli operatori





Indice

- L'overloading degli operatori
- Restrizioni sull'overloading degli operatori
- L'overloading degli operatori: le funzioni membro
- L'overloading degli operatori: le funzioni friend
- L'overloading degli operatori unari
- L'overloading dell'operatore ++
- Il costruttore di copia e l'overloading dell'operatore =
- L'overloading degli operatori di cast

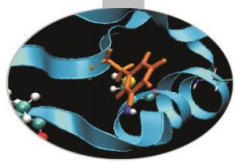


L'overloading degli operatori

- In tutti i linguaggi, gli **operatori** sono dei simboli convenzionali che rendono più agevole la presentazione e lo sviluppo di concetti di uso frequente:

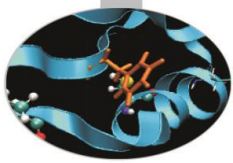
```
int  a,b,c;
int d=a+b*c;
```

- Il C++ supporta, come ogni altro linguaggio, un insieme di **operazioni** per i suoi **tipi *nativi***
- E' pertanto necessario che anche le **operazioni** fra tipi astratti (classi) possano essere descritte tramite simboli convenzionali
- Il C++ consente di soddisfare questa esigenza tramite **l'overloading** degli operatori



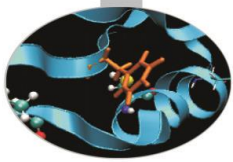
Esempio

```
#include <iostream>
using namespace std;
class Test
{
        public:
                Test();
                Test(int a[]);
                friend Test somma(Test t1, Test t2);
                friend int scalar_multiply(Test t1, Test t2);
                friend Test somma( Test t1, int c);
                void print();
        private:
                int data[3];
};
```



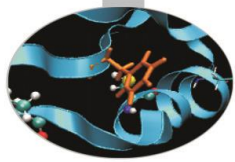
Esempio

```
Test::Test()
{
  for(int i=0;i<3;i++) data[i]=0;
}

Test::Test(int a[])
{
  for(int i=0;i<3;i++) data[i]=a[i];
}

void Test::print()
{
  for(int i=0;i<3;i++) cout<<data[i]<<" ";
  cout <<endl;
}

Test somma(Test t1, Test t2)
{
  Test t3;
  for( int i=0;i<3;i++)t3.data[i]=t1.data[i]+t2.data[i];
  return t3;
}
```



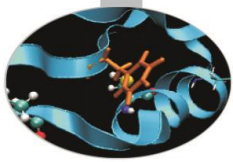
Esempio

```
Test somma (Test t1, int c)
{
        Test t3;
        for( int i=0;i<3;i++) t3.data[i]=t1.data[i]+c;
        return t3;
}

int scalar_multiply(Test t1, Test t2)
{
        int sum=0;
        for(int i=0;i<3;i++) sum+= t1.data[i]*t2.data[i];
        return sum;
}
```



Esempio

```
int main()
{
        int aa[3]={1,3,5};
        int bb[3]={2,4,6};
        int cc[3]={1,1,1};
        Test a(aa);
        Test b(bb);
        Test c(cc);
        cout<<"a "<<endl;
        a.print();
        cout<<"b "<<endl;
        b.print();
        cout<<"c "<<endl;
        c.print();
        // a+b
        Test t3;
        t3= somma(a,b);
        cout<<" operazione a+b "<<endl;
        t3.print();
        // a+b*c
        Test t4;
        cout<<"operazione a+b*c "<<endl;
        t4 = somma(a, scalar_multiply(b,c));
        t4.print();

        return 0;
}
```

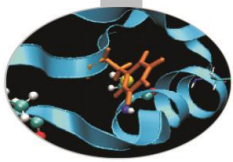


L'overloading degli operatori

- In C++ *non* è possibile creare nuovi operatori, ma è possibile eseguire l'*overloading* della maggior parte degli operatori esistenti così che possano agire nel modo più appropriato sugli *oggetti* delle classi definite dall'utente.
- La sintassi dell'overloading degli operatori ricalca quella della dichiarazione e della definizione delle funzioni ove, al posto del nome della funzione, compare la parola ***operator*** seguita dal simbolo dell'operatore che si vuole ridefinire:

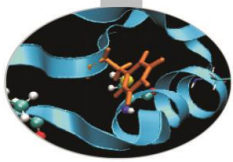
```
type operator operator_symbol(list_of_args)
{
   overloading definition
}
```

L'overloading degli operatori



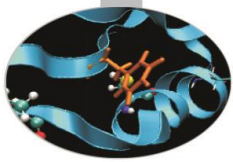
- Solo due operatori possono operare sugli oggetti di una classe *senza* dover essere necessariamente ridefiniti. Essi sono: l'operatore di assegnamento **=** e l'operatore indirizzo ***&***.
- L'overloading degli operatori è un utile strumento per scrivere espressioni che contengono gli oggetti con la stessa *concisione* delle espressioni che contengono variabili associate ai tipi predefiniti.
- Dovendo operare con le classi e con gli attributi delle classi, gli operatori ridefiniti sono, generalmente, *funzioni membro* o *friend* delle classi stesse (se devono operare su dati privati).

Restrizione sull'overloading degli operatori

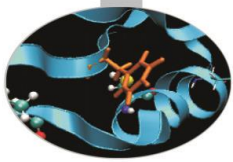


- I seguenti operatori *non* possono essere ridefiniti:
- **. :: ?: sizeof() typeid()**
- Il numero degli operandi di un operatore *non* può essere modificato.
- La precedenza degli operatori viene mantenuta (è legata al simbolo non al significato)
- Come già accennato, *non* è possibile implementare nuovi operatori. Per es. non è permesso costruire un operatore ** che, come in FORTRAN, esegua l'elevamento a potenza.
- **Non** è permesso ridefinire l'azione di un qualsiasi operatore sui tipi predefiniti.
- Se, per esempio, viene effettuato l'overloading dell'operatore somma ***+***, questo *non* implica che gli operatori correlati, come ***+=,*** vengano automaticamente ridefiniti dal compilatore. Affinché si abbia un comportamento coerente di tutti gli operatori correlati all'operatore somma si rende, dunque, necessario implementare il loro overloading a partire da quello dell'operatore somma stesso.

L'overloading degli operatori: funzioni membro



- Nel fare l'overloading, gli operatori vengono trattati come metodi di una classe o come funzioni esterne (eventualmente *friend*)
- Quando un operatore coincide con una funzione non membro, è preferibile che tale funzione sia dichiarata *friend* della classe (se deve accedere ai membri private di una classe).
- I metodi della classe possono fare utilizzo del puntatore implicito *this* che punta allo stesso oggetto della classe in cui il metodo è definito.
- Di default il primo argomento di ogni metodo è implicitamente il puntatore nascosto *this* (viene gestito dal compilatore)
- Ne consegue che se il *left operand* è un oggetto o un riferimento ad un oggetto della classe, l'operatore può essere convenientemente implementato come metodo della classe.
- Nell'overloading degli operatori **()**, **[]**, **->** e di ogni operatore di assegnamento, la funzione operatore **deve** essere dichiarata come **metodo della classe**.



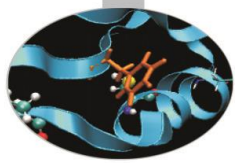
Operatori binari

- Nel caso di operatori binari dichiarati come metodi il numero di argomenti è 1 (il puntatore nascosto *this*) viene utilizzato come *left operand*

esempio: definiamo all'interno della classe Time l'overloading dell'operatore di uguaglianza **==** in modo da determinare se due date siano o meno uguali.

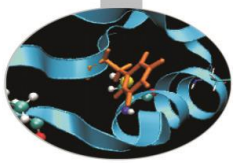
```
// header file time.h
#ifndef TIME_H
#define TIME_H
class Time{
private:
    int hour, minute,second;
public:
    Time(int, int, int);
     ~Time();
     bool operator==(const Time&);
     void printTime();
};
#endif
```

Esempio



// time_fun.cc, contiene le definizioni delle funzioni

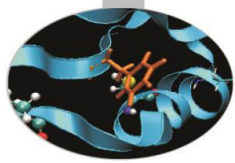
```
#include<iostream.h>
#include"time.h"
Time::Time(int hr, int min, int sec) {
  hour = (hr < 24 && hr >=0) ? hr : 0;
  minute = (min < 60 && min >=0) ? min  : 0;
  second = (sec < 60 && sec >=0) ? sec : 0;}
Time::~Time(){ }
void Time::printTime() {
   cout << "Time is " << hour << ":" << minute
       <<  ":" << second << endl;}
bool Time::operator==(const Time& tm){
   if ( hour == tm.hour && minute == tm.minute
       && second == tm.second)
           return true;
   else
           return false;
}
```



Esempio

```
// time_prg.cc, contiene il programma vero e proprio
#include<iostream.h>
#include"time.h"
int main(){
   Time sveglia_a(6,45,30), sveglia_b(7,0,0);
   Time sveglia_c(6,45,30);
   sveglia_a.printTime(); sveglia_b.printTime();
   if(sveglia_a == sveglia_b)
    cout << " The times are the same. " << endl;
   else
       cout << " The times are different. " << endl;
   sveglia_a.printTime();
   sveglia_c.printTime();
   if(sveglia_a == sveglia_c)
    cout << " The times are the same. " << endl;
   else
       cout << " The times are different. " << endl;
return 0; }
```

Esempio



- Infatti come output otteniamo:

```
Time is 6:45:30
Time is 7:0:0
The times are different.
Time is 6:45:30
Time is 6:45:30
 The times are the same.
```

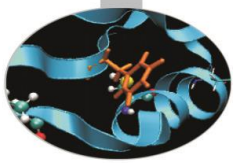
Dunque l'espressione

sveglia_a ***==*** *sveglia_b*

è trattata come una chiamata a:

*sveglia_a.****operator ==*** ***(****sveglia_b****);***

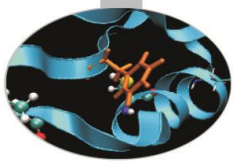
ove *sveglia_a* è acceduto, implicitamente, tramite il puntatore *this* mentre *sveglia_b* è l'argomento della funzione.



Esempio

Operatore binario come funzione friend

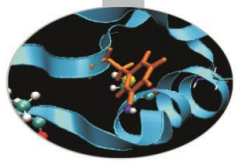
```
// header file time.h
#ifndef TIME_H
#define TIME_H
class Time{
private:
    int hour, minute,second;
public:
    Time(int, int, int);
     ~Time();
     friend bool operator==(const Time &t1,const
  Time&t2);
     void printTime();
};
#endif
```



Esempio

```
bool operator==(const Time& t1, const Time &t2){
    if ( t1.hour == t2.hour && t1.minute == t2.minute
        && t1.second == t2.second)
            return true;
    else
            return false;
}
```

L'overloading degli operatori: le funzioni friend

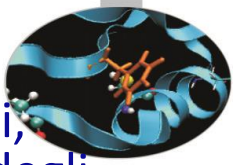


- Se l'operando di sinistra (nel caso di un operatore binario) o l'unico previsto (operatore unario) *non* deve essere un oggetto della classe in cui si sta lavorando, allora la funzione operatore *non* deve essere una funzione membro e sarà, generalmente, dichiarata come *friend* della classe.

- Nel caso di un operatore binario, la corrispondente funzione operatore *friend*, non potendo far uso del puntatore *this*, deve ricevere due argomenti.

- Una funzione non membro che non sia dichiarata friend, può accedere ai membri *private* di una classe solo attraverso eventuali funzioni membro di tipo *set* e *get*. Queste funzioni membro dovrebbero essere dichiarate *inline* per migliorare l'efficienza del codice.

Esempio

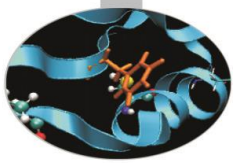


Esempio: scriviamo un programma che riceva come input, e quindi stampi, l'ora su cui si vuole puntare una sveglia, facendo uso dell'overloading degli operatori **>>** e **<<** :

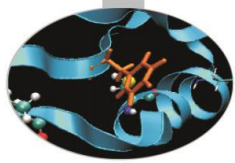
// header file time2.h

```
#ifndef TIME2_H
#define TIME2_H
#include<iostream>
using namespace std;
class Time{
   friend istream& operator>>(istream&, Time&);
   friend ostream& operator<<(ostream&,
                              const Time&);
   private:
      int hour, minute,second;
   public:
      Time(int=0, int=0, int=0);
      ~Time();
};
#endif
```

Esempio



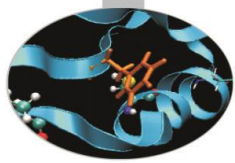
```
// time_fun2.cc, contiene le definizioni delle funzioni
#include<iostream.h>
#include"time2.h"
Time::Time(int hr, int min, int sec){
   hour=(hr < 24 && hr >=0) ? hr : 0;
   minute=(min < 60 && min >=0) ? min : 0;
   second=(sec < 60 && sec >=0) ? sec : 0;}
Time::~Time(){ }
istream& operator>>(istream& input, Time &sveglia){
   input >> sveglia.hour;
   if( sveglia.hour > 24 || sveglia.hour < 0 )
         sveglia.hour=0;
   input >> sveglia.minute;
   if( sveglia.minute > 60 || sveglia.minute < 0 )
          sveglia.minute=0;
   input >> sveglia.second;
   if( sveglia.second > 60 || sveglia.second < 0 )
     sveglia.second=0;
   return input;}
```



Esempio

// time_fun2.cc, contiene le definizioni delle funzioni

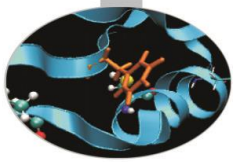
```
// … segue
ostream& operator<<(ostream& output,
                    const Time &sveglia){
   output << sveglia.hour << ":"
     << sveglia.minute << ":"
                    << sveglia.second << endl;
return output;}
```



Esempio

```
// time2_prg.cc, contiene il programma vero e proprio
#include<iostream.h>
#include"time2.h"
int main(){
   Time alarm1, alarm2;
   cout << "Insert time: " << endl;
   cin >> alarm1;
   cout << "Insert time: " << endl;
   cin >> alarm2;
   cout << "The first time is: " << alarm1
        << "and the second one is: " << alarm2;
return 0; } // Un possibile run del programma è il seguente:

Insert time:
7 12 23
Insert time:
8 25 43
The first time is: 7:12:23
and the second one is: 8:25:43
```

Esempio



Ad es. l'espressione:

```
cin >> alarm1;
```

viene trattata come una chiamata a

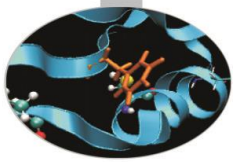
```
operator >> (cin, alarm1);
```

La chiamata a cascata, ad es., nell'espressione:

```
cout << "The first time is " << alarm1 <<  "and the
  second one is: " << alarm2;
```

è invece permessa dal fatto che la funzione operator<< restituisce *sempre* un *reference* a cout (output) sia quando viene mandata in stampa una stringa (come in cout << " The first time is:") che quando invia su standard output un oggetto di tipo Time (es.: *cout << alarm1).*

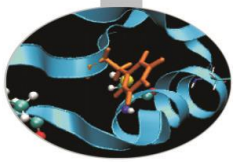
L'overloading degli operatori unari



Esempio1: funzione *membro, usa il puntatore esplicito this*

```
class String{
   public:
   bool operator!() const;
   …};
// nel main file:
String stg;
if ( !stg ) {…} // l'espressione !stg viene trattata come
                // una chiamata a stg.operator!()
```

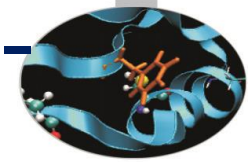
L'overloading degli operatori unari



esempio2: funzione *friend*

```
class String{
friend bool operator!(const String &);
…};
// nel main file
String stg;
if ( !stg )_{…} // l'espressione viene
 trattata come una
        // chiamata a operator!(str)
```

L'overloading dell'operatore di pre-incemento ++



- L'operatore di incremento **++** può essere prefisso o suffisso. Nel farne l'overloading è necessario che le due versioni siano ben distinte per evitare ambiguità in fase di compilazione.
- L'operatore di *pre-incremento* può essere dichiarato come funzione *membro* o *friend* di una classe ed avrà come prototipo:

```
nome_classe& operator++();
```

oppure

```
friend nome_classe& operator++(nome_classe&);
```

Se *var_c* è un oggetto di una qualsiasi classe definita dall'utente, l'espressione ***++var_c*** viene valutata dal compilatore come

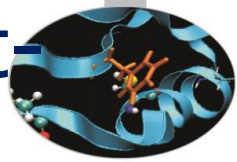
```
var_c.operator++();
```

oppure

```
operator++(var_c);
```

a seconda del prototipo dell'operatore di pre-incremento.

L'overloading dell'operatore di post-incemento ++



- Anche l'operatore di *post-incremento* può essere dichiarato come funzione membro o friend di una classe, ma il suo *prototipo* deve essere scritto in maniera *differente* da quello dell'operatore di pre-incremento: questo si ottiene inserendo nella lista degli argomenti un parametro fittizio di tipo ***int***. Avremo dunque:

```
nome_classe operator++( int );
```

oppure

```
friend nome_classe operator++( nome_classe&, int );
```

L'espressione var_c**++** è ora valutata dal compilatore come:

```
var_c.operator++( 0 );
```

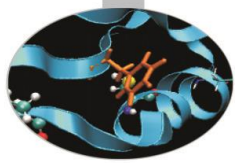
oppure

```
operator++( var_c, 0 );
```

a seconda della natura del prototipo dell'operatore di post-incremento.

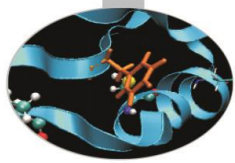
Naturalmente, tutto quanto detto sull'overloading dell'operatore di incremento **++** vale anche per l'overloading dell'operatore di decremento **- -**.

Esempio



- Esempio: incremento e decremento di oggetti della classe Time

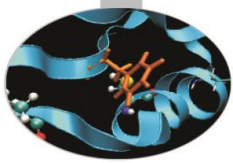
```
// header file time3.h
#ifndef TIME3_H
#define TIME3_H
#include<iostream.h>
class Time{
friend ostream& operator <<(ostream&, const Time&);
private:
    int hour, minute,second;
    void increment();            // funzione di utilità
public:
    Time(int=0, int=0, int=0);
     ~Time();
    Time& operator++();   // operatore di pre-incremento
    Time operator++( int ); //operatore di post-incremento

};
#endif
```



Esempio

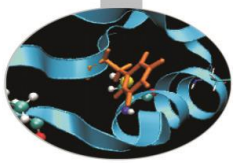
```
// time_fun3.cc, contiene le definizioni delle funzioni
#include"time3.h"
Time::Time(int hr, int min, int sec){
  hour=(hr < 24 && hr >=0) ? hr : 0;
  minute=(min < 60 && min >=0) ? min  : 0;
  second=(sec < 60 && sec >=0) ? sec : 0;}

Time::~Time(){}
void Time::increment(){
   ++second;
   if (second==60){
       second=0;
       ++minute;
   }
   if (minute==60) {
        minute=0;
       ++hour;
   }
    if (hour == 24) hour=0;
}
```

Esempio



```
// segue

Time& Time::operator++(){
   increment();
   return *this;}    //Ritorna una reference, può essere
                     //usato come left -value
Time Time::operator++( int ){
   Time temp= *this;
   increment();
   return temp;}   // restituisce un valore
                   // non un reference, non può essere un
                   //left-value


ostream& operator<<(ostream& output, const Time &tempo){
output << tempo.hour << ":" << tempo.minute << ":"
          << tempo.second << endl;  return output;}
```

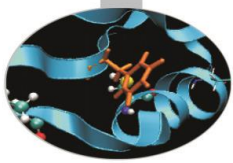
Esempio



- **// time3_prg.cc, contiene il programma vero e proprio**

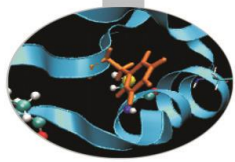
```
#include<iostream.h>
#include"time3.h"
int main(){
Time s_time(23,59,56);
cout << "Starting time is: " << s_time;
cout << "Using the preincrement operator" << endl;
for(int i=0; i<74; i++)
{
    ++s_time;
    cout << i << "  " << s_time;
}
cout << "Using the postincrement operator" << endl;
cout << "s_time++ is: " << s_time++;
cout << "s_time is: " << s_time;
return 0; }
```

Esempio



Come output otteniamo:

```
Starting time is: 23:59:56
Using the preincrement operator
0   23:59:57
1   23:59:58
2   23:59:59
3   0:0:0
4   0:0:1
5   0:0:2
6   0:0:3
…
71   0:1:8
72   0:1:9
73   0:1:10
Using the postincrement operator
s_time++ is: 0:1:10
s_time is: 0:1:11
```

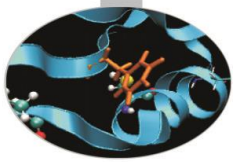


Operatori come funzioni friend

Una funzione operatore non membro permette di avere operatori commutativi.

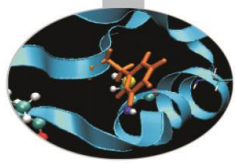
```
class MyInt{
MyInt operator+(int x);
public:
        MyInt(int usr_dat=0){dat=usr_dat;}
private:
        int dat;
};
MyInt MyInt::operator+(int x){
        MyInt temp;
        temp.dat = this->dat + x;
        return temp;
}
```

Operatori come funzioni friend



```
int main()
{
   MyInt a(10);
   a+1; // OK
   1+a; //Error stiamo utilizzando un operatore
        //del tipo predefinito int
   return 0;
}
```

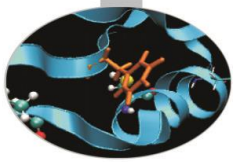
Operatori come funzioni friend



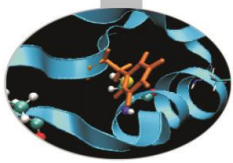
```
class MyInt{
MyInt operator+(int);
friend MyInt operator+(int, const MyInt &obj1);
public:
        MyInt(int usr_dat=0){dat=usr_dat;}
private:
        int dat;
};
MyInt MyInt::operator+(int x){
    MyInt temp;
    temp.dat = this->dat + x;
    return temp;
}
MyInt operator+(int x, const MyInt &obj1){
    MyInt temp;
    temp.dat = x + obj1.dat;
    return temp;
}
```

```
int main()
{
MyInt a(10);
a+1; // OK
1+a; //OK
return 0;
}
```

Esempio di overloading degli operatori

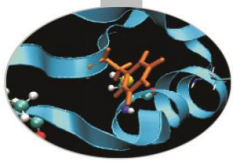


- La libreria standard del C++ mette a disposizione la classe *string* per la rappresentazione delle stringhe.
- Grazie all'overloading degli operatori questa classe rende molto più agevole la manipolazione di stringhe di testo rispetto alle stringhe C-like (array di char)
- Ecco l'elenco degli operatori sovraccaricati:
- =, ==, < , >, <= >=, !=, +, +=, <<, >>



Classe string esempi:

```
#include <iostream>
#include <string>          //inclusione header string
using namespace std;

int main()
{
        string s1("alpha");
        string s2("beta");
        string s3("omega");
        string s4,s5;
        s4=s1;                              //assegnamento
        cout<<"S4 "<<s4<<endl;
        s5= s1+s2;        //concatenazione +assegnamento
        cout<<"S5 "<<s5<<endl;
        s4+=s3;           // concatenazione + assegnamento
        cout<<"S4 "<<s4<<endl;
        s4=s1+"-"+s2;
        cout<<"S4 "<<s4<<endl;
        s3="pippo"+s1;
        cout<<"S3 "<<s3<<endl;
        s3=s1+"pippo";
        cout<<"S3 "<<s3<<endl;
return 0
}
```



Classe string esempi

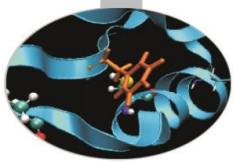
- Anche le operazioni di confronto logico sono immediate

```
if(s1 > s2 )
    cout <<"S2 precede S1 "<<endl;
if( s1 == s2 )
    cout<<"S1 ed S2 sono identiche "<<endl;
```

- Come le operazioni di I/O

```
cout<<endl<<"Inserire una stringa "<<endl;
cin>>s;
cout<<"Stringa inserita "<<s;
```

Classe string vs C-like string



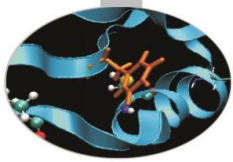
C++ string

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
   string str1 ("Hello");
   string str2 ("World");
   string str3;
   str3= str1;
   cout<< "str3 "<<str3;
   str1+=str2;
   cout<< "str1 "<str1;
   cout<< "str1 size"<<
str1.size()<<endl;
   return 0;
}
```

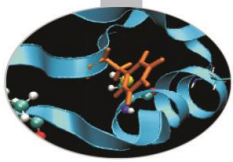
C-like string

```
#include <stdio.h>
#include <string.h>
int main ()
{
     char str1[12] = "Hello";
     char str2[12] = "World";
     char str3[12];
     int len ;
     /* copy str1 into str3 */
     strcpy(str3, str1);
     printf("strcpy( str3, str1) : %s\n", str3 );
     /* concatenates str1 and str2 */
     strcat( str1, str2);
     printf("strcat( str1, str2): %s\n", str1 );
     /* total lenghth of str1 after concatenation*/
     len = strlen(str1);
     printf("strlen(str1) : %d\n", len );
     return 0;
}
```

Classe string vs C-like string

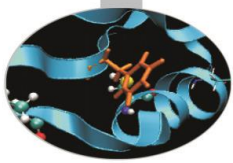


- La dimensione della stringa non deve essere specificata
- Gli oggetti di tipo string vengono dimensionati automaticamente con la stringa che devono contenere
- Nelle operazioni di assegnamento e concatenazione la dimensione della stringa si aggiusta automaticamente. La gestione dinamica della memoria evita tutti gli errori di overflow .
- Le operazioni tra stringhe sfruttano l'overloding degli operatori anziché funzioni di libreria.



Costruttore di copia e di assegnamento

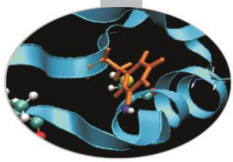
- Quando un oggetto è *passato per valore* ad una funzione, viene eseguita una copia bit a bit dell'oggetto. Se l'oggetto contiene un puntatore ad una locazione di memoria allocata dinamicamente, la copia dell'oggetto conterrà a sua volta un puntatore alla stessa area di memoria e, dunque, ogni modifica al contenuto di quest'area di memoria si trasmetterà anche all'oggetto originale. All'uscita dalla funzione, inoltre, la chiamata al distruttore della copia dell'oggetto potrà andare a danneggiare l'oggetto originale.
- La *restituzione* di un oggetto da parte di una funzione provoca, a sua volta, la creazione di un oggetto temporaneo contenente i valori restituiti dalla funzione, attraverso il meccanismo della copia bit a bit. Quando tale oggetto temporaneo è restituito alla sezione di codice chiamante, esso esce dal suo scope e ne viene invocato il distruttore, che potrebbe distruggere, per es., aree di memoria allocate dinamicamente e, quindi, importanti anche per l'oggetto nella sezione di codice chiamante



Costruttore di copia e di assegnamento

```
#include <iostream>
using namespace std;
class Test
{
        public:
                Test(int n, string name);
                ~Test();
                void stampa();
        private:
                int *pointer;
                int n;
                string name;
};
Test::Test(int n, string name)
{
        this->n=n;
        this->name=name;
        pointer=new int[n];
        for(int i=0;i<n;i++)
         pointer[i]=i;
}
Test::~Test()
{
        cout<<"Distruttore "<<name<<endl;
        delete [] pointer;
}
```

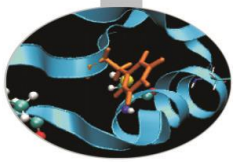
Costruttore di copia e di assegnamento



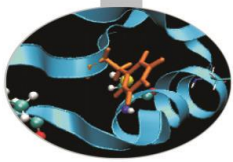
```
void Test::stampa()
{
        cout<<"Stampa "<<name<<endl;
         for(int i=0;i<n;i++)
                cout<<pointer[i]<<endl;
}
void function(Test t)
{
        cout<<"Function"<<endl;
        t.stampa();
}

int main()
{
        Test t(3, "t");
        function(t);
        cout<<"End function"<<endl;
         return 0;
}
```

Output
./example
Function
Stampa t
0
1
2
Distruttore t
End function
Distruttore t
*** glib detected ***./example
double free or corruption

Costruttore di copia e di assegnamento



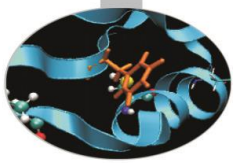
- La copia bit a bit sta, inoltre, alla base dell'operazione di *assegnamento* di un oggetto ad un altro oggetto della stessa classe. Anche in questo caso, se l'oggetto che sta a destra dell'operatore di assegnamento **=** (rvalue) contiene fra i suoi membri un puntatore ad una ben definita locazione di memoria, anche l'oggetto che funge da lvalue conterrà un puntatore alla medesima area di memoria con la conseguenza che modifiche sul puntatore di uno dei due oggetti si ripercuoteranno anche sul puntatore appartenente all'altro oggetto.
- In C++ la creazione di una copia bit a bit di un oggetto è, dunque da tenere in considerazione. Questa situazione si verifica, in generale, durante l'**assegnamento** o l'**inizializzazione**.



Costruttore di copia e di assegnamento

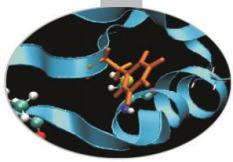
```
#include <iostream>
using namespace std;
class Test
{
        public:
                Test(int n, string name);
               ~Test();
                void stampa();
                void change(int n);
        private:
                int *pointer;
                int n;
                string name;
};
void Test::change(int n)
{
        cout<<"Change "<<name<<endl;
         for(int i=0;i<n;i++)
                 pointer[i]+=n;
}
```

Costruttore di copia e di assegnamento



```
int main()
{
        Test t1(4,"t1");
        Test t=t1;
        t1.change(3);
        t1.stampa();
        t.stampa();

        return 0;
}
```

```
./example2
Costruttore t1
Change t1
Stampa t1
3
4
5
3
Stampa t1
3
4
5
3
Distruttore t1
Distruttore t1
*** glib detected ***./example
double free or corruption
```



Costruttore di copia e di assegnamento

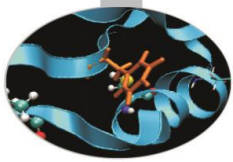
- Il **costruttore di copia** è un modo per ovviare ai potenziali errori/problemi descritti, ma si può applicare **solo** alle **inizializzazioni**. Nel caso di **assegnamenti** è, invece, necessario ricorrere all'**overloading** dell'**operatore** di assegnamento **=**.

- Un'**inizializzazione** può avvenire in tre circostanze differenti:

  - nella dichiarazione di un oggetto (*Time t1 = t2*);
  - nel passaggio di un oggetto ad una funzione (*time_fun(t1)*) ;
  - nella creazione di un oggetto temporaneo che rappresenti il valore restituito da una funzione (*t2=alarm_time()*).

- Se un **costruttore di copia** è stato definito all'interno di una classe, allora viene automaticamente chiamato dal compilatore al presentarsi di ciascuna delle tre situazioni precedenti. Il compilatore crea ancora una copia bit a bit dell'oggetto che, però, non fa più riferimento alle locazioni di memoria eventualmente puntate dall'oggetto originale.

Costruttore di copia e di assegnamento



- La dichiarazione del **costruttore di copia** segue il modello:

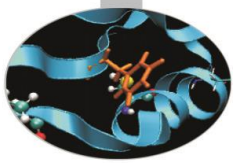
*nome_classe(nome_classe**&**);*

ovvero è *necessario* utilizzare la chiamata per *reference*. Se, infatti, si usasse una chiamata per valore, al costruttore di copia verrebbe passata una copia dell'oggetto, ma per ottenere questa copia il compilatore invocherebbe nuovamente il costruttore di copia stesso: si avrebbe, in questo modo, una *ricorsione* infinita.

- Nell'overloading dell'**operatore** di **assegnamento** è invece obbligatorio che il *tipo restituito* sia un *reference*: ciò consente, infatti, di eseguire assegnamenti multipli (es. a=b=c). Avremo quindi una definizione che segue la sintassi:

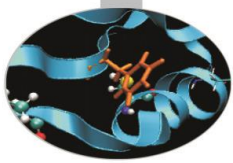
```
   nome_classe&
nome_classe::operator=(nome_classe& nome_oggetto);
{   istruzioni ;
    …
    return *this; }
```

Esempio



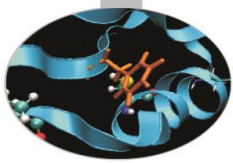
- **Esempio1**: uso del costruttore di copia sulla classe Ptr_class che contiene un dato membro di tipo puntatore a intero. Non è stato eseguito l'overloading dell'operatore di assegnamento.

```
// header file ptr_class.h
#ifndef PTR_CLASS_H
#define PTR_CLASS_H
#include<iostream.h>
class Ptr_class{
   private:
        int *ptr;
   public:
        Ptr_class ();   Ptr_class( int );  // costruttori
        Ptr_class( const Ptr_class& );  // costruttore di copia
        ~Ptr_class();
        int getValue() const;
        void printPtr() const; // stampa su video l'indirizzo
                               // contenuto in ptr
        void setValue( int );   };
#endif
```



Esempio

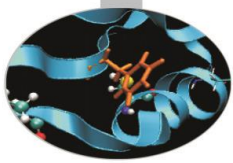
```
// file ptr_class_fun.cc

#include<iostream.h>
#include”ptr_class.h”
Ptr_class::Ptr_class(){
     ptr=new int;
    *ptr=0;
     cout << "Costruttore senza argomenti" << endl; }

Ptr_class::Ptr_class(int var){
     ptr = new int;
    *ptr = var;
     cout << “Costruttore “<< endl; }

Ptr_class::Ptr_class(const Ptr_class& obj){
     ptr = new int;
    *ptr = *obj.ptr;
     cout << “Costruttore di copia” << endl; }
```



Esempio

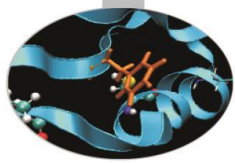
```
// segue

Ptr_class::~Ptr_class(){
     delete ptr;
}
int Ptr_class::getValue() const{
     return *ptr;
}
void Ptr_class::printPtr() const{
  cout << ptr << endl;
}
void Ptr_class::setValue(int n_val){
  *ptr=n_val;
}
```

Esempio



// file ptr_class_prg.cc

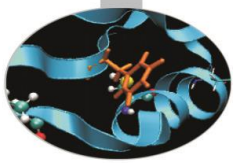
```
#include<iostream.h>
#include"ptr_class.h"
int main(){
   Ptr_class p1(20);
   cout << "p1 is: " << p1.getValue() <<" " << endl;
   cout << "The address pointed by p1.ptr is: ";
   p1.printPtr();
   Ptr_class p2 = p1;  // viene chiamato il costruttore di copia
   cout << "p2 is: " << p2.getValue() <<" " << endl;
   cout << "The address pointed by p2.ptr is: ";
   p2.printPtr();
   Ptr_class p3;
   cout << "p3 is: " << p3.getValue() <<" " << endl;
   cout << "The address pointed by p3.ptr is: ";
   p3.printPtr();

// continua
```



Esempio

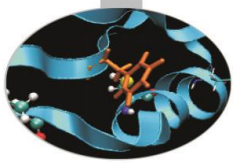
```
p3 = p1;  // è un assegnamento, non viene
          // chiamato il costruttore di copia
cout << "p3 is now: " << p3.getValue() << endl;
cout << "The address pointed by p3.ptr is: ";
p3.printPtr();
cout << "Let's assign to p3 the new value 32"
   << endl;
p3.setValue(32);
cout << "p3 is now: " << p3.getValue() << endl;
cout << "p1 is now: " << p1.getValue() << endl;
return 0;}
```

Esempio



Vediamo l'output del programma con alcuni commenti:

```
Costruttore          // costruttore chiamato per p1
p1 is: 20
The address pointed by p1.ptr is: 0x400053c8
Costruttore di copia    // costruttore di copia chiamato per p2
p2 is: 20
The address pointed by p2.ptr is: 0x400053d8
   // p2.ptr punta ad un indirizzo diverso da quello puntato da
   p1.ptr
Costruttore senza argomenti   // costruttore senza argomenti
   chiamato per p3
p3 is: 0
The address pointed by p3.ptr is: 0x400053e8
p3 is now: 20   // p1 è stato assegnato a p3 tramite copia bit a
   bit
The address pointed by p3.ptr is: 0x400053c8
   // dopo la copia bit a bit p3.ptr punta allo stesso indirizzo
   puntato da p1.ptr
Let's assign to p3 the new value 32
p3 is now: 32
p1 is now: 32
// una modifica al valore di *p3.ptr si ripercuote anche su
   *p1.ptr
```

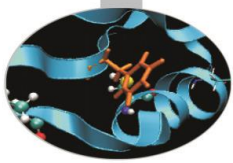


Esempio

esempio2: riscriviamo il programma precedente implementando l'overloading dell'operatore di assegnamento

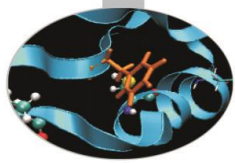
```
// header file o_ptr_class.h
#ifndef O_PTR_CLASS_H
#define O_PTR_CLASS_H
#include<iostream.h>
class Ptr_class{
   private:
        int *ptr;
   public:
        Ptr_class();  Ptr_class( int );
        Ptr_class( const Ptr_class& ); // costruttore di copia
         ~Ptr_class();
        int getValue() const; void printPtr() const;
        void setValue(int);
        Ptr_class& operator=(Ptr_class&); // overloading
                                          //dell'operatore =
};
#endif
```

Esempio



// file o_ptr_class_fun.cc

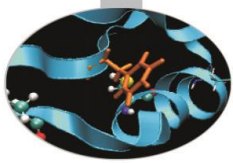
```
#include<iostream.h>
#include"ptr_class.h"
Ptr_class::Ptr_class(){
   ptr = new int;
  *ptr=0;
   cout << "Costruttore senza argomenti" << endl; }
Ptr_class::Ptr_class(int var){
   ptr = new int;
  *ptr = var;
   cout << "Costruttore" << " " << endl; }
Ptr_class::Ptr_class(const Ptr_class& obj){
   ptr = new int;
  *ptr = *obj.ptr;
    cout << "Costruttore di copia" << " " << endl; }
 Ptr_class::~Ptr_class(){
    delete ptr; }
 // continua
```



Esempio

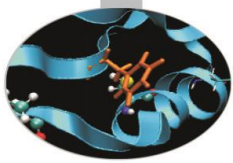
```
// segue

int Ptr_class::getValue() const{
     return *ptr;
}
void Ptr_class::printPtr() const{
     cout << ptr << endl;
}
void Ptr_class::setValue(int n_val){
     *ptr=n_val;
}

Ptr_class& Ptr_class::operator=(Ptr_class& p1){
  *ptr=*p1.ptr;
  return *this;
}
```

Esempio



```
// file o_ptr_class_prg.cc
#include<iostream.h>
#include"ptr_class.h"
int main(){
   Ptr_class p1(20);
   cout << "p1 is: " << p1.getValue() <<" " << endl;
   cout << "The address pointed by p1.ptr is: ";
   p1.printPtr();
   Ptr_class p2 = p1;
   cout << "p2 is: " << p2.getValue() <<" " << endl;
   cout << "The address pointed by p2.ptr is: ";
   p2.printPtr();
   Ptr_class p3;
   cout << "p3 is: " << p3.getValue() <<" " << endl;
   cout << "The address pointed by p3.ptr is: ";
   p3.printPtr();
   p3 = p1; // è un assegnamento, viene chiamato l'overloading
                di =
// continua
```

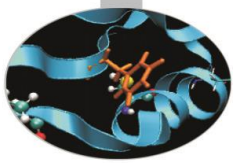
Esempio



// segue

```
   cout << "p3 is now: " << p3.getValue() << endl;
   cout << "The address pointed by p3.ptr is: ";
   p3.printPtr();
   cout << "Let's assign to p3 the new value 32" << endl;
   p3.setValue(32);
   cout << "p3 is now: " << p3.getValue() << endl;
   cout << "p1 is now: " << p1.getValue() << endl;
   Ptr_class p4(15);
   cout << "p4 is: " << p4.getValue() <<" " << endl;
   p3 = p2 = p4; // chiamata "a cascata" all'overloading
dell'operatore =
   cout << "After a multiple assignment "
        << "p2 is now: " << p2.getValue()
        << " and p3 is now: " << p3.getValue() << endl;
   return 0;}
```

Esempio

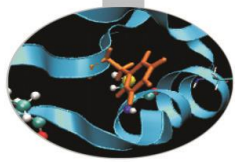


- L'output del programma è ora:

```
Costruttore
p1 is: 20
The address pointed by p1.ptr is: 0x400053c8
Costruttore di copia
p2 is: 20
The address pointed by p2.ptr is: 0x400053d8
Costruttore senza argomenti
p3 is: 0
The address pointed by p3.ptr is: 0x400053e8
p3 is now: 20
   // p1 è stato assegnato a p3 tramite l'overloading
   dell'operatore =
The address pointed by p3.ptr is: 0x400053e8
   // l'indirizzo puntato da p3.ptr non è stato modificato
```

… continua

Esempio

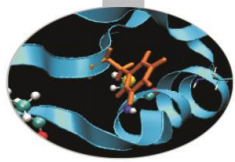


… segue

```
Let's assign to p3 the new value 32
p3 is now: 32
p1 is now: 20
  // modiche su *p3.ptr non si ripercuotono su *p1.ptr
Costruttore     // costruttore chiamato per p4
p4 is: 15
After a multiple assignment p2 is now: 15 and p3 is now:
   15
```

// la chiamata multipla dell'overloading dell'operatore = funziona correttamente

L'overloading degli operatori di cast

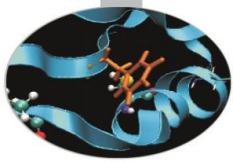


- E' possibile convertire un oggetto di una classe in un oggetto di un'altra classe o di un tipo predefinito facendo uso di un appropriato *operatore di cast*, che può essere realizzato come una funzione membro di una classe.
- *L'overloading di un operatore di cast* che consenta la conversione di un oggetto di tipo *nome_classe,* generalmente definito dall'utente, in un oggetto di tipo *nome_tipo,* è definito come:

```
nome_classe :: operator nome_tipo ( ){
    corpo dell' overloading
}
```

- Il tipo di dato restituito non è, naturalmente, specificato perché corrisponde al tipo di dato in cui viene convertito l'oggetto (nome_tipo).

L'overloading degli operatori di cast



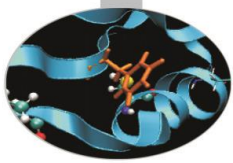
Esempio:

```
Time::operator float(); // converte un oggetto di tipo
Time in un reale
Time::operator newTime() const; // converte un oggetto
    //  di tipo Time in un oggetto di tipo newTime.
```

Esempio: scriviamo l'overloading dell'operatore di cast static_cast<int> in modo tale che agisca sulla classe Time

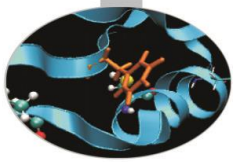
```
// header file cast.h
#ifndef CAST_H
#define CAST_H
class Time{
 private:
  int hour, minute, second;
 public:
  Time(int=0, int=0, int=0);
  ~Time();
  operator int();
  void printTime() const;
};#endif
```

L'overloading degli operatori di cast



```
// file cast_fun.cc
#include<iostream.h>
#include"cast.h"

Time::Time(int hr, int mn, int sc) {
  hour=(hr>=0 && hr<24)?hr :0;
  minute=(mn>=0 && mn<60)?mn :0;
  second=(sc>=0 && sc<60)?sc :0;   }

Time::~Time(){ }

void Time::printTime() const{
  cout << "Time is: " << hour << ":" << minute << ":" <<
   second << endl; }

Time::operator int(){
  return hour+minute+second; }  /* converte un oggetto
   della classe Time in un intero pari alla somma dei suoi
   dati membro private */
```

L'overloading degli operatori di cast



// file cast_prg.cc

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

int main(){
  Time tempo(20,32,12);
  tempo.printTime();
  cout << "The integer for tempo is: " <<
  static_cast<int>(tempo) << endl;
  return 0; }
```

Come output abbiamo:

```
Time is: 20:32:12
The integer for tempo is: 64
```