Tutorial di C++ per utenti C

Eric Brasseur
traduzione di Paolo Avogadro

Contents

1. Un nuovo modo di includere librerie

Esiste un nuovo modo per includere (**#include**) delle librerie (il vecchio metodo funziona comunque, ma il compilatore si lamenta). L'estensione **.h** non viene piu' usata, mentre i nomi delle librerie standard del C devono ora cominciare con la lettera **c**. Per fare si' che il porgramma usi queste librerie correttamente bisogna aggiungere la seguente linea di comandi: **using namespace std;**

```
using namespace std;
#include <iostream>     // questa e' una libreria chiave del C++
#include <cmath>        // questa e' la libreria standard del C chiamata math.h
int main ()
{
  double a;
  a = 1.2;
  a = sin (a);
  cout << a << endl;
  return 0;
}
```

```
output:  0932039
```

Qualche consiglio per chi ha poca esperienza:
Per compilare questo programma, prima va scritto (o copiato) in un editor di testo (gedit, kwrite, kate, kedit, vi, emacs, nano, pico, mcedit, Notepad...), il file va salvato e e chiamato, per esempio *test01.cpp* (se si e' proprio dei principianti sarebbe utile mettere il file all'interno della propria home directory, ovvero per esempio /home/jones in un sistema di tipo Unix).

Per compilare questo file sorgente, bisogna scrivere questo comando (sulla maggior parte dei sistemi di tipo Unix) in una console o nella terminal window:

```
g++ test01.cpp -o test01
```

(questo produrra' un binario eseguibile chiamato *test01*) per far girare questo eseguibile prodotto dalla compilazione (supponendo che non ci siano stati degli errori, nella fase di scrittura o copia), si deve scrivere:

```
./test01
```

Ogni volta che si modifica il file sorgente test01.cpp, e' necessario compilarlo ancora se si vuole che le modifiche si propaghino all'eseguibile (per esempio con la freccia cursore in su si puo' scorrere la lista dei vecchi comandi lanciati).

2· Nuovi modi per commentare linee di codice

E' possibile usrare // per indicare che una linea non e' codice ma un commento:

```
using namespace std;     // uso del namespace della libreria standard
#include <iostream>      // la libreria iostream e' molto utile
int main ()              // la funzione principale del codice
{
  double a;              // dichiarazione della variabile a
  a = 456.47;
  a = a + a * 21.5 / 100; // un calcolo
  cout << a << endl;       // mostra il contenuto di a
  return 0;                // fine della funzione
}
```

554.611

(La possibilita' di usare // per i commenti e' stata aggiunta al C nel C99 e nell' ANSI C 2000)

3· Nuovi modi per fare leggere comandi da tastiera e scrivere a video

Per interfacciare i comandi da tastiera e a schermo con il codice e' possibile usare dei nuovi comandi:

- cout <<
- cin >>

```
using namespace std;
#include <iostream>
int main()
{
  int a;          // a e' una variabile di tipo integer
  char s [100];   // s punta a una stringa
  cout << "Questo e' un programma di esempio." << endl;
  cout << endl;          // endl e' identico a \n (end of line)
  cout << "Inserisci la tua eta': ";
   cin >> a;
  cout << "Inserisci il tuo nome: ";
   cin >> s;
  cout << endl;
  cout << "Ciao " << s << ", hai " << a << " anni" << endl;
  cout << endl << endl << "arrivederci" << endl;
  return 0;
}
```

```
Questo e' un programma di esempio.
Inserisci la tua era':  12
Inserisci il tuo nome:  Paolo

Ciao Paolo, hai 12 anni


arrivederci
```

4. Dichiarazione delle variabili

Le variabili possono ora essere dichiarate in qualunque parte del codice:

```
using namespace std;
#include <iostream>
int main ()
{
double a;
cout << "Ciao, questo e' un programma di prova." << endl;
cout << "Inserisci il parametro a: ";
cin >> a;
a = (a + 1) / 2;
double c;         // <=========== variabile appena dichiarata
c = a * 5 + 1;
cout << "c contiene : " << c << endl;
int i, j;         // <=========== variabili appena dichiarate
i = 0;
j = i + 1;
cout << "j contiene : " << j << endl;
return 0;
}
```

```
Ciao, questo e' un programma di prova.
Inserisci il parametro a: 7
c contiene              : 21
j contiene              : 1
```

E' consigliabile usare questa caratteristica per rendere il proprio codice piu' leggibile, e non piu' disordinato. Come nel C, le variabili possono essere incapsulate tramite le parentesi graffe {}. In questo caso sono locali nello **scope** definito proprio dalle parentesi graffe. Quello che succede a tali variabili all'interno della zona **incapsulata** non modifica eventuali variabili con lo stesso nome ma che si trovano all'esterno.

```
using namespace std;
#include <iostream>
int main ()
{
  double a;
  cout << "Inserisci un numero: ";
  cin >> a;
```

```
    {
      int a = 1;
      a = a * 10 + 4;
      cout << "Numero locale: " << a << endl;
    }
    cout << "Tu hai inserito: " << a << endl;
    return 0;
}
```

```
Inserisci un numero: 9
Numero locale:       14
Tu hai inserito:  9
```

5. Inizializzazione di una variabile con dei calcoli

Una variabile puo' essere inizializzata con un calcolo di altre variabili, per esempio:

```
using namespace std;
#include <iostream>
int main ()
{
  double a = 12 * 3.25;
  double b = a + 1.112;
  cout << "a contiene: " << a << endl;
  cout << "b contiene: " << b << endl;

  a = a * 2 + b;
  double c = a + b * a;
  cout << "c contiene: " << c << endl;

  return 0;
}
```

```
a contiene: 39
b contiene: 40.112
c contiene: 4855.82
```

6· Variabile dichiarate nella dichiarazione di un loop

Il C++ consente di definire delle variabili *locali* per un loop:

```
using namespace std;
#include <iostream>
int main ()
{
  int i;
  i = 487;                      // dichiarazione di i
  for (int i = 0; i < 4; i++)   // dichiarazione locale di i
  {
     cout << i << endl;  // questo manda in output 0, 1, 2 e 3
  }
  cout << i << endl;      // questo invece 487
}
return 0;
```

```
0
1
2
3
487
```

Nel caso in cui la variabile non sia dichiarata da qualche parte prima del loop, una persona potrebbe essere tentata di usarla anche dopo il loop stesso. Alcuni vecchi compilatori accettano questo comportamento. In quel caso la variabile mantiene l'ultimo valore che aveva alla fine del loop stesso. E' molto SCONSIGLIATO usare questo modo di programmare (viene considerata una *bad practice*) che puo' indurre ad errori difficilmente trovabili).

```
using namespace std;
#include <iostream>
int main ()
  {
   for (int i = 0; i < 4; i++)
   {
     cout << i << endl;
   }
   cout << i << endl;    // bad practice
   i += 5;               // bad practice
   cout << i << endl;    // bad practice
```

```
    return 0;
}
```

```
t.cpp: In function âint main()â:
t.cpp:12: error: name lookup of âiâ changed for new ISO âforâ scoping
t.cpp:7: error: using obsolete binding at âiâ
```

7· Accesso alle variabili globali, anche se delle variabili locali hanno lo stesso nome

Si puo' accedere ad una variabile globale anche se all'interno di una funzione si e' dichiarata un'altra variabile con lo stesso nome.

```
using namespace std;
#include <iostream>
double a = 128;
int main ()
  {
  double a = 256;
  cout << "Local a: " << a << endl;
  cout << "Global a: " << ::a << endl;  // nota l'operatore ::
  return 0;
}
```

```
Local a: 256
Global a: 128
```

8. E' possibile dichiarare una REFERENZA ad un'altra variabile

E' possibile dichiarare una REFERENZA ad un'altra variabile. In pratica questo fa si' che una variabile diventi un'altra variabile, e quindi siano collegate tra loro (si usa per questo il simbolo di refereziazione &). Detto in altri termini, si costruisce un altro nome della stessa variabile.

```
using namespace std;
#include <iostream>
int main ()
{
  double a = 3.1415927;
  double &b = a;                          // b e' un altro nome per a!
  b = 89;
  cout << "a contiene: " << a << endl;   // mostra 89
  return 0;
}
```

```
a contiene: 89
```

Se sei abituato ai puntatori e vuoi assolutamente sapere cosa succede, semplicemente pensalo in questo modo:
`double &b = a` => `double *b= &a` e tutti i successivi b sono rimpiazzati da *b
Se si e' creata una referenza da una variabile ad un altra, questo non puo' poi essere modificato nel seguito del codice per collegare la variabile ad una nuova variabile. Per esempio non e' possibile scrivere, poche linee dopo, &b=c ed aspettarci che ora b sia c. Non funziona. La dichiarazione iniziale definisce una volta per tutte b. La b e a sono *sposate* per sempre e nulla le separera'.
Le **referenze** possono essere usate per consentire ad una funzione di modificare una variabile chiamante:

```
using namespace std;
#include <iostream>
void change (double &r, double s)
  {
  r = 100;
  s = 200;
  }
int main ()
  {
  double k, m;
```

```
  k = 3;
  m = 4;
  change (k, m);
  cout << k << ", " << m << endl;  // mostra 100, 4
  return 0;
}
```

```
100, 4
```

Chiaramente lo stesso risultato poteva essere ottenuto tramite i puntatori in C, in particolare il compilatore C++, se si dovesse tradurre questo codice in C, scriverebbe:

```
using namespace std;
#include <iostream>
void change (double *r, double s)
  {
  *r = 100;
   s = 200;
  }
int main ()
  {
  double k, m;
  k = 3;
  m = 4;
  change (&k, m);
  cout << k << ", " << m << endl; // mostra 100, 4
  return 0;
}
```

```
100, 4
```

Una referenza puo' essere usata per consentire ad una funzione di restituire una variabile:

```
using namespace std;
#include <iostream>
double &maggiore (double &r, double &s) // nota l'operatore & prima di maggiore
{
   if (r > s) return r;
   else
   return s;
}
int main ()
{
   double k = 3;
   double m = 7;

   cout << "k: " << k << endl;    // mostra 3
   cout << "m: " << m << endl;    // mostra 7
```

```
    cout << endl;

    maggiore (k, m) = 10;     // ho assegnato un valore alla funzione

    cout << "k: " << k << endl;    // mostra  3
    cout << "m: " << m << endl;    // mostra 10
    cout << endl;

    maggiore (k, m) ++;      // aggiungo uno al valore della funzione
                             //   (dopo averla chiamata)

    cout << "k: " << k << endl;  // mostra  3
    cout << "m: " << m << endl;  // mostra 11
    cout << endl;

  return 0;
}
```

```
k: 3
m: 7

k: 3
m: 10

k: 3
m: 11
```

Ancora una volta, se si e' abituati ai puntatori del C e si domanda come questa notazione funzioni, basta immaginare che il compilatore traduca quanto scritto sopra nel seguente codice C standard:

```
using namespace std;
#include <iostream>
double *maggiore (double *r, double *s)
{
  if (*r > *s) return r;
  else
 return s;
}
int main ()
{
  double k = 3;
  double m = 7;

  cout << "k: " << k << endl;
  cout << "m: " << m << endl;
  cout << endl;

  (*(maggiore (&k, &m))) = 10;

  cout << "k: " << k << endl;
  cout << "m: " << m << endl;
  cout << endl;

  (*(maggiore (&k, &m))) ++;

  cout << "k: " << k << endl;
  cout << "m: " << m << endl;
  cout << endl;
  return 0;
}
```

```
k: 3
m: 7

k: 3
m: 10

k: 3
m: 11
```

Per finire, per le persone che non amano i puntatori ma devono interagire con essi, le **referenze** possono essere utili per in pratica fare un "un-pointer" delle variabili. Attenzione che questo tipo di azione puo' essere considerata una "bad practice" e puo' creare problemi. Vedasi l'esempio:
`https://www.embedded.com/electronics-blogs/programming-pointers/4023307/References-vs-Pointers`

```
using namespace std;
#include <iostream>
double *silly_function ()  // questa funzione restituisce un puntatore ad un
    double
{
  static double r = 342;
  return &r;
}
int main ()
{
  double *a;
  a = silly_function();
  double &b = *a;    // ora b e' il double verso cui punta!

  b += 1;            // ottimo!
  b = b * b;         // non c'e' bisogno di scrivere *a ovunque!
  b += 4;

  cout << "Contenuto di *a, b and r: " << b << endl;
  return 0;
}
```

```
contenuto di *a, b e r: 117635
```

9. E' possibile dichiarare dei namespace

Si possono dichiarare dei namespace. Le variabili dichiarate entro un **namespace** possono essere usate tramite l'operatore ::

```
using namespace std;
#include <iostream>
#include <cmath>
namespace first
{
  int a;
  int b;
}

namespace second
{
  double a;
  double b;
}

int main ()
{
   first::a = 2;
   first::b = 5;
   second::a = 6.453;
   second::b = 4.1e4;
   cout << first::a + second::a << endl;
   cout << first::b + second::b << endl;

   return 0;
}
```

```
8.453
41005
```

10. Una funzione puo' essere dichiarata inline

Se una funzione contiene semplici linee di codice e non contiene **for** loop o simili, allora puo' essere dichiarata **inline**. Questo implica che il codice della funzione stessa, al momento della compliazione, verra' inserito in tutti i luoghi dove essa viene usata. In pratica diventa simile ad una macro. Il vantaggio principale e' che il codice diventa piu' veloce. Come piccolo difetto c'e' il fatto che l'eseguibile diventera' un po' piu' grande perche' tutte le linee della funzione verranno ripetute ovunque venga chiamata.

```
using namespace std;
#include <iostream>
#include <cmath>
inline double ipotenusa (double a, double b)
{
   return sqrt (a * a + b * b);
}
int main ()
{
   double k = 6, m = 9;
// le seguenti due linee producono esattamente lo stesso eseguibile:

   cout << ipotenusa (k, m) << endl;
   cout << sqrt (k * k + m * m) << endl;

   return 0;
}
```

```
10.8167
10.8167
```

11. E' stata aggiunta la struttura exception

Oltre alle classiche strutture di controllo del C: **for, if, do, while, switch...** nel C++ viene inserita una nuova struttura chiamata **exception**:

```
using namespace std;
#include <iostream>
#include <cmath>
int main ()
{
    int a, b;
    cout << "inserisci un numero: ";
    cin >> a;
    cout << endl;
    try
    {
        if (a > 100) throw 100;
        if (a < 10) throw 10;
        throw a / 3;
    }
    catch (int risultato)
    {
        cout << "il risultato e' : " << risultato << endl;
        b = risultato + 1;
    }
    cout << "b contiene: " << b << endl;
    cout << endl;

    // un altro esempio dell'uso di exception:

    char zero [] = "zero";
    char pair [] = "pari";
    char notprime [] = "non primo";
    char prime [] = "primo";

    try
    {
        if (a == 0) throw zero;
        if ((a / 2) * 2 == a) throw pair;
        for (int i = 3; i <= sqrt (a); i++)
        {
```

```
            if ((a / i) * i == a) throw notprime;
        }
        throw prime;
    }
    catch (char *conclusion)
    {
        cout << "il numero che hai inserito e' "<< conclusion << endl;
    }

    cout << endl;
    return 0;
}
```

```
Inserisci un numero: 5

il risultato e': 10
b  contiene: 11

il numero che hai inserito e' primo
```

12. E' possibile definire valori di default per gli argomenti delle funzioni

```
using namespace std;
#include <iostream>
double test (double a, double b = 7)  // se non sepcificato, b=7
{
  return a - b;
}
int main ()
{
   cout << test (14, 5) << endl;  // mostra a video 14 - 5
   cout << test (14) << endl;     // mostra a video 14 - 7
}
return 0;
```

```
9
7
```

13· Overload delle funzioni

Un notevole vantaggio del C e' la possibilita' di fare l'**overload** delle funzioni. Questo significa che funzioni differenti possono avere lo **stesso nome**, basta che ci sia qualcosa che consenta al compilatore di distinguerle in modo univoco, per esempio: il **numero** di parametri, il **tipo** dei parametri,...

```
using namespace std;
#include <iostream>
double test (double a, double b) // questa funzione prende 2 double e li somma
{
  return a + b;
}
int test (int a, int b)      // questa, invece, prende 2 interi e li sottrae
{                            // ma ha lo stesso nome, "test", di quella che somma
  return a - b;
}
int main ()
{
   double   m = 7,   n = 4;
   int      k = 5,   p = 3;
   cout << test(m, n) << " , " << test(k, p) << endl;
   return 0;
}
```

```
11, 2
```

14. Overload di operatori (+-*/...) per il loro utilizzo con nuovi tipi di dato

L'overload di operatori puo' essere usato per ridefinire dei simboli di base per lavorare con nuovi tipi di parametri:

```
using namespace std;
#include <iostream>
struct vettore   // creo l'oggetto vettore
{
   double x;
   double y;
};

vettore operator * (double a, vettore b)  //
{               // fa prodotto PER uno scalare e un vettore 2D
   vettore r;
   r.x = a * b.x;
   r.y = a * b.y;
   return r;
}

int main ()
{
   vettore k, m;        // Non c'e' bisogno di scrivere "struct vettore"
   k.x = 2;             // per essere in grado di scrivere
   k.y = -1;            // k = vettore (2, -1)
                        // vedi il Cap 19.
   m = 3.1415927 * k;   // Magia!

   cout << "(" << m.x << ", " << m.y << ")" << endl;
   return 0;
}
```

```
(6.28319, -3.14159)
```

Oltre all'operatore di moltiplicazione (*), in C++ ci sono altri 43 operatori di base di cui si puo' fare overload, tra cui ci sono +=, ++, l'array [], e cosi' via...
Tramite un overload, l'operatore << normalmente impiegato per lo shifting binario di interi, puo' portare all'output

di uno stream (per esempio **cout «**. E' possibile fare ulteriori overload dell'operatore <<, per l'output di nuovi tipi, come i vettori:

```
using namespace std;
#include <iostream>
struct vettore
{
  double x;
  double y;
};
ostream& operator << (ostream& o, vettore a)  // tipo in uscita e' un ostream
{
   o << "(" << a.x << ", " << a.y << ")";
   return o;
}
int main ()
{
  vettore a;
  a.x = 35;
  a.y = 23;
  cout << a << endl;  // mostra a video (35,23)
  return 0;
}
```

```
(35,23)
```

15. Template: funzioni indipendenti dal tipo

Stanchi di definire la stessa funzione 5 volte? Una definizione per parametri di tipo **int**, una nuova definizione per parametri di tipo **double**, una per i **float**... Non ti sarai dimenticato un tipo? E se dovessi usare la stessa funzione con un nuovo tipo? Nessun problema il compilatore C++ puo' generare automaticamente tutte le versioni delle funzioni che siano necessarie! Basta specificare come e' fatta la funzione dichiarando una funzione **template**:

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

template <class ttype>
ttype minimo (ttype a, ttype b)
{
   ttype r;
   r = a;
   if (b < a) r = b;  // gli operatori "<" e "=" devono essere
                      // definiti per tutti tipi per cui sono usati

   return r;
}
int main ()
{
   int i1, i2, i3;
   i1 = 34;
   i2 = 6;
   i3 = minimo (i1, i2);
   cout << "Piu' piccolo: " << i3 << endl;

   double d1, d2, d3;
   d1 = 7.9;
   d2 = 32.1;
   d3 = minimo (d1, d2);
   cout << "Piu' piccolo: " << d3 << endl;

   cout << "Piu' piccolo: " << minimo (d3, 3.5) << endl;

   return 0;
}
```

```
Piu' piccolo: 6
```

```
Piu' piccolo: 7.9
Piu' piccolo: 3.5
```

La funzione **minimo** viene usata tre volte nel codice qui sopra, nondimeno il compilatore C++ genera solo 2 versioni di essa:

- `int minimo(int a , int b)`
- `double minimo(double a, double b)`

Questo e' sufficiente per tutto il programma. Cosa sarebbe successo se avessi provato a calcolare qualcosa del tipo **minimo(i1,d1)**? (ovvero mettendo un int come primo ingresso e un double come secondo). Il compilatore avrebbe restituito un errore. Questo perche', nella forma scelta nell'esempio, nel **template** entrambi i parametri hanno lo stesso tipo. In genereale, fortunatamente, si puo' usare un numero arbitrario di tipi quando si definisce un template che possono essere "tipi" standard (**char, int, double**,...) o definiti dall'utente. Qui sotto c'e' un esempio dove la funzione minimo accetta parametri di qualunque tipo (differenti o identici tra loro) e restituisce un valore di somma che ha lo stesso tipo del primo parametro:

```
using namespace std;
#include <iostream>
template <class type1, class type2>  // tipe1 primo argomento, tipe2 secondo
type1 minimo (type1 a, type2 b)
{
   type1 r, b_convertito;       // dichiara un paio di oggetti di type1
   r = a;                       // di base prende a come minimo il primo
   b_convertito = (type1) b;    // fa un cast del secondo parametro nel tipo del
      primo
   if (b_convertito < a) r = b_convertito; // controlla se il secondo e' minore
   return r;
}
int main ()
{
   int i;
   double d;
   i = 45;
   d = 7.41;

   cout << "Piu' piccolo: " << minimo (i, d) << endl;      // int    vs double
   cout << "Piu' piccolo: " << minimo (d, i) << endl;      // double vs int
   cout << "Piu' piccolo: " << minimo ('A', i) << endl;    // char   vs int

   return 0;
}
```

```
Piu' piccolo: 7
Piu' piccolo: 7.41
Piu' piccolo: -
```

(Si noti che il codice ASCII per il carattere '-' e' 45, mentre il codice di 'A' e' 65, quindi nel caso dell'operatore < tra **char**, converte automaticamente nel corrispondente codice ASCII.

16. E' meglio usare NEW e DELETE per alloccare e dealloccare la memoria

I comandi **new** e **delete** possono essere usati per alloccare e dealloccare la memoria. Sono in qualche modo piu' puliti rispetto alle funzioni **malloc** e **free** del C standard. Si noti che per gli array si usano invece:

- **new []**
- **delete []**

```
using namespace std;
#include <iostream>
#include <cstring>
int main ()
{
double *d;    // d e' una variabile il cui scopo
              // e' di contenere l'indirizzo di
              // memoria dove e' posto un double

d = new double;  // new allocca una zona di memoria
                 // grande abbastanza da contenere un double
                 // e restituisce il suo indirizzo.
                 // L'indirizzo e' qunidi messo in d.

*d= 45.3         // Il numero 45.3 e' messo
                 // nella zona di memoria, il cui
                 // indirizzo e' dato da d.


cout << "Inserisci un numero: ";
cin >> *d;
*d = *d + 5;
cout << "Risultato: " << *d << endl;


delete d;  //  delete deallocca la zona
           //  di memoria il cui indirizzo
           //  e' dato dal puntatore d
           //  quell'indirizzo non puo' piu' essere usato
```

```
  d = new double[15] // allocca una zona per un array di 15 double
                     // Nota che ognuno dei 15 double viene costruito.
                     // In questo caso non serve, ma e' fondamentale
                     // quando si usano dei tipi di dato
                     // che necessitino che il loro constructor sia
                     // usato per ogni istanza

  d[0]=4456;
  d[1]=d[0]+567;

  cout << "Contenuto di d[1]: " << d[1] << endl;

  delete [] d;       // delete [] distrugge la zona di memoria
                     // Nota che ognuno dei 15 ingressi di tipo
                     // double verra' distrutto. Anche in questo
                     // caso, ora non e' importante, ma al momento in
                     // cui verranno usati i destructor per le istanze
                     // delle classi (il metodo ~). Se venisse usato
                     // delete senza mettere le parentesi quadre
                     // questo provocherebbe la dealloccazione della
                     // zona di memoria, senza distruggere ognuna delle 15 istanze
                     // Questo comportamento puo' causare un memory leakage.

  int n = 30;
  d = new double[n]; // new puo' essere usato per alloccare un array
                     // dinamicamente, ad una grandezza n
  for (int i = 0; i < n; i++)
  {
    d[i] = i;
  }

  delete [] d;

  char *s;
  s = new char[100];

  strcpy (s, "Ciao!");

  cout << s << endl;

  delete [] s;

  return 0;
  }
```

```
Inserisci un numero: 6
Risultato: 11
```

```
Contenuto di d[1]: 5023
Ciao!
```

17· Si possono mettere delle funzioni (metodi) in uno Struct o una Classe

Nel C standard, uno **struct** contiene solo dati. In C++, uno **struct** puo' contenere anche delle funzioni. Queste funzioni sono possedute dallo **struct** e sono pensate per operare sui dati dello **struct** stesso. Queste funzioni sono chiamate **METODI**. Nel seguito viene mostrato il metodo **superficie** associato allo struct **vettore**:

```
using namespace std;
#include <iostream>
struct vettore
{
   double x;
   double y;
   double superficie () // intesa come rettangolo di cui il vettore e' diagonale
   {
      double s;
      s = x * y;
      if (s < 0) s = -s;
      return s;
   }
};
int main ()
{
   vettore a;
   a.x = 3;
   a.y = 4;

   cout << "La superficie di a: " << a.superficie() << endl;
   return 0;
}
```

```
La superficie di a: 12
```

Nell'esempio qui sopra **a** e' un' **istanza** dello struct "vettore". (Si noti che il comando "**struct**" non era necessario quando si e' dichiarato il vettore **a**).

Proprio come per le funzioni, un metodo puo' essere un overload di qualsiasi operatore del C++, puo' avere un numero arbitrario di parametri (ma uno di questi parametri e' implicito: l'istanza su cui agisce), restituire qualunque tipo di parametri o nessuno (un metodo e' una funzione...).

17.1 Classi

- Cos'e' una **class** (classe)?
- Una **classe** e' una **struct** che mantiene i propri dati **nascosti**.
- solo i metodi della classe possono accedere ai dati. Non e' possibile accedere ai dati di una classe direttamente, a meno che questi siano stati definiti tramite la direttiva: **public**.

Qui sotto c'e' un esempio della **definizione** di una **classe**, che si comporta esattamente come lo struct sopra perche' i dati **x** e **y** sono definiti come public:

```
using namespace std;
#include <iostream>
class vettore
{
   public:
   double y;
   double x;

   double superficie ()
   {
      double s;
      s = x * y;
      if (s < 0) s = -s;
      return s;
   }
};


int main ()
{
   vettore a;
   a.x = 3;
   a.y = 4;
   cout << "La superficie di a: " << a.superficie() << endl;
   return 0;
}
```

```
La superficie di a: 12
```

Nell'esempio qui sopra, direttamente dal **main** e' possibile modificare i dati dell'istanza del vettore, usando:

- **a.x=3**
- **a.y=4**

Questo e' stato possibile per la direttiva **public:** usata nella definizione della classe. **L'uso della public e' considerata bad practice! si veda il Capitolo 30.**

Ad un metodo e' consentito cambiare le variabili dell'istanza su cui agisce:

```
using namespace std;
#include <iostream>
class vettore
{
  public:
  double x;     // parametri della classe (accessibili dall'esterno per il public)
  double y;     // parametri della classe (              "                        )

  vettore costruisci_opposto() // metodo che RESTITUISCE l'opposto dell'istanza
  {                            // nota che questo metodo ha un "tipo"
     vettore r;                // e' proprio la stessa classe!
     r.x = -x;
     r.y = -y;
     return r;
  }

  void trasforma_in_opposto()     // metodo che TRASFORMA l'istanza nel suo
     opposto
  {                       // e' un metodo VOID, non restituisce nulla!
      x = -x;             // trasforma soltanto l'istanza su cui agisce
      y = -y;
  }
  void da_calcolare (double a, double b, double c, double d)
                         // metodo che MODIFICA l'istanza
  {                      // (e' void)
     x = a - c;          // ATTENZIONE non c'e' un return...
     y = b - d;          //
  }

  vettore operator * (double a) // overload del prodotto
  {                       // occhio che non e' come un metodo normale
     vettore r;           // in cui gli argomenti vanno messi tra tonde ()
     r.x = x * a;         // l'istanza della classe va messa prima del *
     r.y = y * a;         // il double va messo dopo. NON si usano le parentesi
     return r;            // crea una NUOVA istanza
  }
};

int main ()
{
   vettore a, b;
   a.x = 3;
   a.y = 5;
   b = a.costruisci_opposto();

   cout << "Vector a: " << a.x << ", " << a.y << endl;
   cout << "Vector b: " << b.x << ", " << b.y << endl;
```

```
    b.trasforma_in_opposto();
    cout << "Vector b: " << b.x << ", " << b.y << endl;

    a.da_calcolare (7, 8, 3, 2);
    cout << "Vector a: " << a.x << ", " << a.y << endl;
    a = b * 2;      // questo e' istruttivo, avendo fatto
                    //l'OVERLOAD del *, il primo ingresso
                    //sara' un vettore, il secondo un double
                    // che era tra parentesi nella definizione del
                    // metodo. Nessuno degli "ingressi" dell'
                    // operatore, va messo tra parentesi (), si
                    // continua ad usare il * come al solito

    cout << "Vector a: " << a.x << ", " << a.y << endl;
    a = b.costruisci_opposto() * 2;

    cout << "Vector a: " << a.x << ", " << a.y << endl;
    cout << "x dell'opposto di a: " << a.trasforma_in_opposto().x << endl;

    return 0;
}
```

```
Vector a: 3, 5
Vector b: -3, -5
Vector b: 3, 5
Vector a: 4, 6
Vector a: 6, 10
Vector a: -6, -10
x dell'opposto di a: 6
```

18. Constructor e destructor: inizializzare e distruggere istanze di una classe

Esistono dei metodi speciali ed essenziali che sono i:

- **constructor** (costruttore)
- **destructor** (distruttore)

Questi metodi vengono chiamati automaticamente nei seguenti casi:

- al momento di creazione di un'istanza della classe (p.es. con un **new**)
- al momento di distruzione di un'istanza della classe (con un **delete**)
- alla fine del programma
- ...

Il costruttore (**constructor**) (e' un metodo con lo **stesso nome** della classe):

- inizializzera' le variabili dell'istanza
- fara' dei calcoli (definiti nella classe)
- allocchera' della memoria
- mandera' a video delle scritte
- ...

In generale il costruttore viene scritto per fare tutto quello che e' necessario per la classe. Nel seguito vediamo un esempio della definizione di una classe con due costruttori di cui viene fatto l'overload (sono funzioni... quindi si puo' fare l'overload dei costruttori):

```
using namespace std;
#include <iostream>
class vettore
{
   public:
   double x;
   double y;
```

```
    vettore ()  // questo e' un costruttore, lo si riconosce
    {           // perche' ha lo stesso NOME della classe
       x = 0;   // se viene fatta una istanza senza indicare
       y = 0;   // le componenti, questo le mette a 0 di default
    }

    vettore (double a, double b) // anche questo e' un costruttore
    {                            // ed e' un overload, perche' questo
        x = a;                   // viene chiamato con 2 argomenti
        y = b;                   // in particolare 2 double
    }                            // che diventeranno le componenti
};

int main ()
{
    vettore k;             // il costruttore vettore () viene chiamato
    cout << "vettore k: " << k.x << ", " << k.y << endl << endl;

    vettore m (45, 2);     // qui viene chiamato vettore (double, double)
    cout << "vettore m: " << m.x << ", " << m.y << endl << endl;

    k = vettore (23, 2); // viene creato un vettore, copiato  in k,
                         // e poi cancellato
                         // perche' viene cancellato? perche' finisce l'esecuzione
    cout << "vettore k: " << k.x << ", " << k.y << endl << endl;

    return 0;
}
```

```
vettore k: 0, 0

vettore m: 45, 2

vettore k: 23, 2
```

La buona pratica di scrittura di un codice suggerisce di non fare l'overload dei costruttori. Sarebbe meglio dichiarare solo un costruttore e dargli dei valori di default quando possibile.

```
using namespace std;
#include <iostream>
class vettore
{
    public:
    double x;
    double y;

    vettore (double a = 0, double b = 0)
```

```
    {
        x = a;
        y = b;
    }
};

int main ()
{
    vettore k;
    cout << "vettore k: " << k.x << ", " << k.y << endl << endl;

    vettore m (45, 2);
    cout << "vettore m: " << m.x << ", " << m.y << endl << endl;

    vettore p (3);
    cout << "vettore p: " << p.x << ", " << p.y << endl << endl;
    return 0;
}
```

```
vettore k: 0, 0

vettore m: 45, 2

vettore p: 3, 0
```

Il distruttore e' spesso non necessario. Puo' essere usato per fare dei calcoli quando l'istanza viene distrutta o per mandare a video dei testi per il debug. Se pero' le variabili dell'istanza puntano a qualche area di memoria alloccata, allora il ruolo del distruttore e' essenziale: **deve liberare la memoria**! Qui vediamio un esempio di un tale utilizzo:

```
using namespace std;
#include <iostream>
#include <cstring>
class persona
{
    public:
    char *name;
    int anni;
    persona (char *n = "nessun nome", int a = 0)  // costruttore
    {
        nome = new char [100];  // new e' meglio di malloc!
        strcpy (name, n);
        anni = a;
        cout << "Instanza inizializzata, 100 bytes allocati" << endl;
    }
    ~persona ()  // DISTRUTTORE, c'e' la ~ davanti al nome!
    {
        delete [] nome; // invece che un semplice free, uso delete
                        // si noti che potrebbe funzionare con un
```

```
                        // semplice delete senza  le [], questo
                        // perche' l'array non contiene C++ sub-oggetti
                        // che debbano essere cancellati. Pero', visto che
                        // il comportamento senza le [] non e' definito, e'
                        // meglio andare sul sicuro e mettere le quadre.

        cout << "L'istanza viene rimossa, 100 byte vengono liberati" << endl;
    }
};          // fine della definizione della classe


int main ()
{
    cout << "Ciao!" << endl << endl;
    persona a;

    cout << a.nome << ", anni " << a.anni << endl << endl;
    persona b ("John");

    cout << b.nome << ", anni " << b.anni << endl << endl;

    b.anni = 21;
    cout << b.nome << ", anni " << b.anni << endl << endl;

    persona c ("Miki", 45);

    cout << c.nome << ", anni " << c.anni << endl << endl;
    cout << "Ciao ciao!" << endl << endl;

    return 0;
}
```

```
Ciao!

L'istanza viene inizializzata, 100 byte allocati
nessun nome, anni 0

L'istanza viene inizializzata, 100 byte allocati
John, anni 0

John, anni 21

L'istanza viene inizializzata, 100 byte alloccati
Miki, anni 45

Ciao ciao!
```

```
L'istanza viene rimossa, 100 byte vengono liberati
L'istanza viene rimossa, 100 byte vengono liberati
L'istanza viene rimossa, 100 byte vengono liberati
```

Qui invece verra' mostrato un breve esempio di una definizione di una classe chiamata "array". C'e' poi un metodo che e' un overload dell'operatore **[]** e restituisce come valore una referenza (**&**) ed e' usato per generare un errore se si tenta di accedere a dati al di fuori dei limiti dell'array.

```
using namespace std;
#include <iostream>
#include <cstdlib>
class array
{
   public:
   int size;
   double *data;
   array (int s)
   {
      size = s;
      data = new double [s];
   }
   ~array ()
   {
      delete [] data;
   }

   double &operator [] (int i) // overload dell'operatore []
   {                           // restituisce un double
      if (i < 0 || i >= size)  // ha come parametro di ingresso un intero
      {
          cerr << endl << "Fuori dai limiti" << endl;
          exit (EXIT_FAILURE);
      }
      else return data [i];
   }
};


int main ()
{
   array t (5);          // OK
   t[0] = 45;            // Ok
   t[4] = t[0] + 6;      // Ok

   cout << t[4] << endl;

   t[10] = 7;            // ERRORE!
```

```
    return 0;
}
```

```
51

Fuori dai limiti
```

19. COPY constructor e l'overload dell'operatore "=" per copiare istanze

Se si copia un oggetto come un vettore, non c'e' nessun problema. Per esempio, se si ha il vettore **k**, di coordinate **(4,7)**, dopo averlo copiato **m=k**, il vettore **m** conterra' anche lui coordinate **(4,7)**. I valori di **k.x** e **k.y** sono stati semplicemente copiati in **m.x** e **m.y**.

Supponiamo ora che si stia usando un oggetto come la classe **persona** definita precedentemente. Questo tipo di oggetti contiene un **puntatore** ad una stringa di caratteri. Se si fa la copia dell'oggetto **persona** scrivendo **p=r** diventa necessario che qualche funzione si incarichi di produrre una copia corretta da **p** a **r**. Altrimenti, **p.nome** puntera' alla stessa stringa di **r.nome**.
Inoltre la precedente stringa puntata da **p.nome** e' persa e diventa un o zombi di memoria. Il risultato sarebbe catastrofico, un disastro di puntatori e dati persi. I metodi che risolvono questo tipo di problemi sono:

- COPY constructor
- un overload dell'operatore =

```
using namespace std;
#include <iostream>
#include <cstring>
class persona
{
   public:
   char *name;
   int anni;
   persona (char *n = "nessun nome", int a = 0)  // costruttore
   {
      name = new char[100];
      strcpy (name, n);
      anni = a;
   }

   persona (const persona &s)  // l'argomento di constructor e' un oggetto
   {                           // di tipo "persona"
      name = new char[100];    // allocca della nuova memoria
      strcpy (name, s.nome); // copia il nome dell'oggetto passato come nuovo
          nome
      anni = s.anni;     // mette come eta' quella della "persona" in argomento
   }
```

```
    persona& operator= (const persona &s) // overload dell' =, restituisce
    {                                     // un oggetto di tipo persona e prende
       strcpy (name, s.nome);             // come argomento un oggetto persona
       anni = s.anni;                     // associa l'eta' dell'argomento
       return *this;                 // restituisce l'istanza appena costruita
     }

     ~persona ()  // DISTRUTTORE
     {
        delete [] name;             // l'unica quantita' alloccata era per il nome
     }
};


void modifica_persona (persona& h)  // nota che questa e' una funzione
                                    // NON e' un metodo!
{
   h.anni += 7;                // semplicemente modifica la "persona" passata
}

persona calcola_persona (persona h)
{
   h.anni += 7;
   return h;
}


int main ()
{
   persona p;
   cout << p.nome << ", anni " << p.anni << endl << endl;
   // output: nessun nome, anni 0

   persona k ("John", 56);
   cout << k.nome << ", anni " << k.anni << endl << endl;
   // output: John, anni 56

   p = k;
   cout << p.nome << ", anni " << p.anni << endl << endl;
   // output: John, anni 56

   p = persona ("Bob", 10);
   cout << p.nome << ", anni " << p.anni << endl << endl;
   // output: Bob, anni 10

   // Ne il copy constructor ne l'overload
   // dell' = sono necessari per l'operazione che modifica
   // p dato (che faremo qui sotto) che solo la referenza verso p
```

```
    // e' passata alla funzione modifica_persona
    modifica_persona (p);

    cout << p.nome << ", anni " << p.anni << endl << endl;
    // output: Bob, anni 17

    // Il copy constructor e' chiamato per passare una
    // copia completa di p alla funzione calcola_persona.
    // La funzione usa quella copia per fare i suoi calcoli
    // poi una copia di quella copia modificata viene fatta per
    // restituire i risultati. In fine l'overload dell'=
    // viene chiamato per incollare la seconda copia dentro k

    k = calcola_persona (p);
    cout << p.nome << ", anni " << p.anni << endl << endl;
    // output: Bob, anni 17

    cout << k.nome << ", anni " << k.anni << endl << endl;
    // output: Bob, anni 24

    return 0;
}
```

```
nessun nome, anni 0

John, anni 56

John, anni 56

Bob, anni 10

Bob, anni 17

Bob, anni 17

Bob, anni 24
```

Il copy constructor consente al programma di fare copie delle istanze quando ci sono dei calcoli. E' un metodo chiave. Durante i calcoli, le istanze sono create per mantenere i riultati intermedi. Sono modificate, copiate e distrutte senza che il programmatore lo noti. Questo e' il motivo per cui questi metodi sono utili anche per oggetti semplici (vedi Capitolo 14).

In tutti gli esempi sopra, i metodi sono definiti all'interno delle definizioni delle classi. Questo implica che sono automaticamente dei metodi **inline**.

20. Prototipi: definire i metodi sotto la definizione di una classe

Consideriamo i seguenti casi:

- se un metodo non puo' essere inline (per esempio perche' ha un loop al suo interno)
- se non si vuole che il metodo diventi inline (per non ingrandire troppo l'eseguibile)
- se si vuole che la definizione di una classe contenga solo un "riassunto" dei metodi in modo che sia leggibile
- se, semplicemente, si vuole separare l'header file .h dal sorgente .cpp

allora:

1. nella classe si inserisce solo il **prototipo** (prototype) del metodo
2. il metodo stesso va definito **sotto** la definizione della classe stessa o in un file sorgente .cpp separato

```
using namespace std;
#include <iostream>
class vettore
{
   public:
   double x;
   double y;

   double superficie(); // il ; e il fatto che non ci siano {} mostra che e' un
      prototipo
};                      // <=== fine della definizione della classe


double vettore::superficie()  // qui sotto definisco esplicitamente il metodo
{
   double s = 0;
   for (double i = 0; i < x; i++)
   {
      s = s + y;  // e' una specie di modo per fare il prodotto
   }
   return s;
```

```
}


int main ()
{
    vettore k;
    k.x = 4;
    k.y = 5;

    cout << "Superficie: " << k.superficie() << endl;
    return 0;
}
```

```
Superficie: 20
```

20.1 Per chi e' alle prime armi

Se volete sviluppare un codice C o C++ serio, avete bisogno di separare il file sorgente in:

- uno (o piu') file header **.h**
- uno (o piu') file sorgente **.cpp**

Qui c'e' un breve esempio di come questo viene fatto. Il programma qui sopra viene diviso in 3 file:

1. **vettore.h** contenente la classe e i prototipi dei metodi
2. **vettore.cpp** dove vengono definiti i metodi della classe vettore
3. **main.cpp** dov c'e' il main

il file header **vettore.h** e':

```
class vettore
{
    public:
    double x;
    double y;

    double superficie();
};
```

Il file sorgente **vettore.cpp**:

```
using namespace std;
#include "vettore.h"
double vettore::superficie()
{
    double s = 0;
    for (double i = 0; i < x; i++)
```

```
    {
        s = s + y;
    }
    return s;
}
```

ed infine un nuovo file sorgente chiamato **main.cpp**

```
using namespace std;
#include <iostream>
#include "vettore.h"
int main ()
{
    vettore k;
    k.x = 4;
    k.y = 5;
    cout << "Superficie: " << k.superficie() << endl;
    return 0;
}
```

Se si assume che il file **vettore.cpp** sia perfetto, allora abbiamo bisogno di compliarlo una volta sola in un file **.o** (un "object file"), tramite il comando (per esempio):

```
g++ -c vettore.cpp
```

(che produce un file **vettore.o**).

Ogni volta che il file **main.cpp** viene modifcato, va compilato in un file eseguibile, per esempio chiamato **test20**, per fare questo bisogna dire esplicitamente al compilatore che deve "linkare" l'oggetto **vettore.o** all'interno dell'esegubile **test20**:

```
g++ main.cpp vettore.o -o test20
```

L'eseguibile (su macchine tipo Unix) viene fatto "girare" con"

```
./test20
```

Questo procedimento ha un certo numero di vantaggi:

- Il codice sorgente `vettore.cpp` deve essere compilato solo una volta. Questo fa risparmiare molto tempo nei progetti grandi (fare il "linking" del file `vettore.o` nell'eseguibile `test20` e' molto veloce).
- E' possibile passare i file `.h` e i `.o`. In questo modo possono usare il tuo programma ma non cambiarlo perche' non hanno i file `.cpp` (non fidarti molto di questo, aspetta fino a quando avrai una comprensione piena di questi problemi).

Si noti che e' possibile compilare anche il file `main.cpp` in un file di tipo "object" e poi linkarlo con `vettore.o`:

```
g++ -c main.cpp
```

```
g++ main.o vettore.o test20
```

20.2 MAKEFILE

Qui viene fatta una digressione rispetto all'argomento "differenze tra C e C++". Se si vuole apparire come *veri* programmatori, si dovrebbe condensare i comandi qui sopra in un `Makefile` e compilare usando il comando `make`. Il contenuto del file sotto e' una versione super semplificata di un tale Makefile. Prova a copiarlo in un file chiamato proprio `Makefile`. Attenzione nota che, ed e' molto importante, lo spazio tra i comandi del g++ deve **obbligatoriamente** essere un `Tab`. Non usare lo spazio, usa invece il carattere di tabulazione (tutto a sinistra di una tastiera internazionale sopra il caps lock).

```
test20: main.o vettore.o
        g++ main.o vettore.o -o test20

main.o: main.cpp vettore.h
        g++ -c main.cpp

vettore.o: vettore.cpp vettore.h
          g++ -c vettore.cpp
```

Per utilizzare questo Makefile per compilare, e' sufficiente usare questo comando:

```
make test20
```

Il comando `make` fara' un "parse" (legge i comandi parola per parola) attraverso il `Makefile` e capira' quello che deve essere fatto. All'inizio, viene detto che:

- `test20` dipende da `main.o` e `vettore.o`
- a questo punto lancera' automaticamente `make main.o` e `make vettore.o`
- poi controllera' se `test20` esiste gia' e controllera' **se** la data di creazione dei file `main.o` e `vettore.o` e' precedente a `test20`. Nel caso in cui il comando `make` determini che la versione corrente di `test20` e' aggiornata non fara' nulla e dira' che non ha fatto nulla.
- altrimenti se `test20` non esiste, oppure `main.o` o `vettore.o` sono piu' recenti di `test20` il comando creera' una versione aggiornata di `test20`, eseguendo: `g++ main.o vettore.o -o test20`

La versione di seguito del `Makefile` e' piu' vicina allo "standard" Makefile:

```
all: test20

test20: main.o vettore.o
        g++ main.o vettore.o -o test20

main.o: main.cpp vettore.h
        g++ -c main.cpp

vettore.o: vettore.cpp vettore.h
```

```
        g++ -c vettore.cpp

clean:
        rm -f *.o test20 *~ #*
```

Si fa partire la compilazione con il comando `make`.

- la prima linea nel Makefile implica che se scrivi solo `make`, in realta' intendi: `make test20`

Se si usa il comando seguente, tutti i file prodotti durante la compilazione e tutti i file di testo di backup verranno cancellati:

```
make clean
```

21. This: per puntare all'istanza su cui sta agendo un metodo

Quando un metodo e' applicato ad un'istanza, quel metodo puo' usare le variabili dell'istanza e modificarle... ma alle volte e' necessario conoscere l'indirizzo di un'istanza. Nessun problema, il comando **this** serve proprio a questo scopo:

```
using namespace std;
#include <iostream>
#include <cmath>
class vettore
{
   public:
   double x;
   double y;

   vettore (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }

   double modulo()
   {
      return sqrt (x * x + y * y);
   }

   void definisci_lunghezza (double a = 1)
   {
      double lunghezza;

      lunghezza = this->modulo(); // fa il il modulo di QUESTA istanza

      x = x / lunghezza * a;
      y = y / lunghezza * a;
   }
};

int main ()
{
```

```
    vettore c (3, 5);
    cout << "Il modulo del vettore c: " << c.modulo() << endl;

    c.definisci_lunghezza(2);       // Transforma c in un vettore di lunghezza 2.
    cout << "Il modulo del vettore c:" << c.modulo() << endl;

    c.definisci_lunghezza();        // Transforma b in un vettore unitario.
    cout << "Il modulo del vettore c: " << c.modulo() << endl;

    return 0;
}
```

```
Il modulo del vettore c: 5.83095
Il modulo del vettore c: 2
Il modulo del vettore c: 1
```

22. Array di istanze di classi

Ovviamente e' possibile dichiarare degli oggetti che sono array di classi:

```
using namespace std;
#include <iostream>
#include <cmath>
class vettore
{
   public:
   double x;
   double y;

   vettore (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }

   double modulo ()
   {
      return sqrt (x * x + y * y);
   }
};

int main ()
{
   vettore s [1000];
   vettore t[3] = {vettore(4, 5), vettore(5, 5), vettore(2, 4)};

   s[23] = t[2];
   cout << t[0].modulo() << endl;
   return 0;
}
```

6.40312

23· Esempio di dichiarazione di una classe

```
using namespace std;
#include <iostream>
#include <cmath>
class vettore
{
  public:
     double x;
     double y;

     vettore (double = 0, double = 0);  // questo e' il costruttore
     vettore operator + (vettore);      // overload del +
     vettore operator - (vettore);      // overload del -
     vettore operator - ();         // overload del - davanti alla classe stessa
     vettore operator * (double a);// overload del * prodotto per uno scalare
     double modulo();               // metodo per calcolare il modulo della classe
     void definisci_lunghezza (double = 1); // modifica il modulo della classe
};


vettore::vettore (double a, double b)    // specifico cosa fa il costruttore
{
  x = a;
  y = b;
}

vettore vettore::operator + (vettore a)   // specifico l'overload del +
{
   return vettore (x + a.x, y + a.y);
}

vettore vettore::operator - (vettore a)   // specifico l'overload del -
{
   return vettore (x - a.x, y - a.y);
}

vettore vettore::operator - ()    // se tra parentesi non c'e' nulla, allora
{                                 // l'operatore va messo davanti alla istanza
   return vettore (-x, -y);       // su cui deve agire
```

```
}

vettore vettore::operator * (double a)   // overload del * per fare prodotto per
    scalare
{
   return vettore (x * a, y * a);
}

double vettore::modulo()          // nota che questo metodo ha un tipo e un
{                                 // HA un valore di return
   return sqrt (x * x + y * y);
}

void vettore::definisci_lunghezza (double a) // questo metodo invece e' void, e
{                                            // NON ha un valore di return
   double lunghezza = this->modulo();
   x = x / lunghezza * a;
   y = y / lunghezza * a;
}

ostream& operator << (ostream& o, vettore a) // questo NON e' un metodo
{                                            // infatti non era tra i
   o << "(" << a.x << ", " << a.y << ")";    // prototipi
   return o;
}

int main ()
{
   vettore a;
   vettore b;
   vettore c (3, 5);

   a = c * 3;
   a = b + c;
   c = b - c + a + (b - a) * 7;

   c = -c;

   cout << "Il modulo del vettore c: " << c.modulo() << endl;
   cout << "Il contenuto del vettore a: " << a << endl;
   cout << "L'opposto del vettore a: " << -a << endl;

   c.definisci_lunghezza(2);   // Transforma c in un vettore di modulo 2

   a = vettore (56, -3);
   b = vettore (7, c.y);
   b.definisci_lunghezza();    // Transforma b in un vettore unitario

   cout << "Il contenuto del vettore b: " << b << endl;
```

```
    double k;

    k = vettore(1, 1).modulo(); // k conterra' 1.4142 (=radice di 2)
    cout << "k contiene: " << k << endl;
    return 0;
}
```

```
Il modulo del vettore c: 40.8167
Il contenuto del vettore a: (3, 5)
L' opposto del vettore a: (-3, -5)
Il contenuto del vettore b: (0.971275, 0.23796)
k contiene: 1.41421
```

E' altresi' possibile definire una funzione che produca la somma di due oggetti vettore senza che questo venga menzionato all'interno della definizione della classe. In questo caso non sara' un metodo della classe, ma piuttosto una funzione che usa vettori:

```
vettore operator + (vettore a, vettore b)
{
return vettore (a.x + b.x, a.y + b.y);
}
```

Nell'esempio delle definizione della classe vettore qui sopra e' definita la moltiplicazione di un vettore per un double. Supponiamo di volere la moltiplicazione di un double per un vettore. In questo caso dobbiamo scrivere una funzione isolata fuori dalla classe:

```
vettore operator * (double a, vettore b)
{
return vettore (a * b.x, a * b.y);
}
```

Chiaramente le keyword **new** e **delete** funzionano anche per le istanze delle classi. In piu', **new** automaticamente chiama il constructor per inizializzare gli oggetti, e **delete** automaticamente chiama il destructor prima di dealloccare la memoria delle variabili dell'istanza come:

```
using namespace std;
#include <iostream>
#include <cmath>
class vettore
{
    public:
    double x;
    double y;

    vettore (double = 0, double = 0);
    vettore operator + (vettore);
    vettore operator - (vettore);
    vettore operator - ();
```

```
    vettore operator * (double);

    double modulo();
    void definisci_lunghezza (double = 1);
};

vettore::vettore (double a, double b)
{
    x = a;
    y = b;
}

vettore vettore::operator + (vettore a)
{
    return vettore (x + a.x, y + a.y);
}

vettore vettore::operator - (vettore a)
{
    return vettore (x - a.x, y - a.y);
}

vettore vettore::operator - ()
{
    return vettore (-x, -y);
}

vettore vettore::operator * (double a)
{
    return vettore (a * x, a * y);
}

double vettore::modulo()
{
    return sqrt (x * x + y * y);
}

void vettore::definisci_lunghezza (double a)
{
    vettore &il_vettore = *this;
    double lunghezza = il_vettore.modulo();
    x = x / lunghezza * a;
    y = y / lunghezza * a;
}

ostream& operator << (ostream& o, vettore a)
{
    o << "(" << a.x << ", " << a.y << ")";
    return o;
```

```
}

int main ()
{
   vettore c (3, 5);
   vettore *r;      // r e' un puntatore a vettore.
   r = new vettore;//  new alloccca la memoria necessaria
                   // per contenere le variabili di vettore,
                   // chiama il costruttore che le
                   // inizializza a 0, 0. Infine
                   // new restituisce l'indirizzo dell'istanza di vettore

   cout << *r << endl;

   r->x = 94;
   r->y = 345;
   cout << *r << endl;

   *r = vettore (94, 343);
   cout << *r << endl;

   *r = *r - c;
   r->definisci_lunghezza(3);
   cout << *r << endl;

   *r = (-c * 3 + -*r * 4) * 5;
   cout << *r << endl;
   delete r;  // Chiama il distruttore di vettore
              // poi libera la memoria.

   r = &c;    // r punta verso il vettore c
   cout << *r << endl;
   r = new vettore (78, 345);  // Crea un nuovo vettore.
   cout << *r << endl;         // Il costruttore  inizializzera'
                               // la x e la y di vettore a 78 and 345

   cout << "componente x di r: " << r->x << endl;
   cout << "componente y di r: " << (*r).x << endl;

   delete r;
   r = new vettore[4];  // crea un array di 4 vettore
   r[3] = vettore (4, 5);
   cout << r[3].modulo() << endl;
   delete [] r;         // cancella l'array

   int n = 5;
   r = new vettore[n];  // bello!
   r[1] = vettore (432, 3);
   cout << r[1] << endl;
```

```
    delete [] r;

    return 0;
}
```

```
(0, 0)
(94, 345)
(94, 343)
(0.77992, 2.89685)
(-60.5984, -132.937)
(3, 5)
(78, 345)
componente y di r: 78
componente x di r: 78
6.40312
(432, 3)
```

24. Variabili “static” in una classe

Una o piu' variabili in una classe possono essere dichiarate **static**. In questo caso:

- esiste **una sola istanza** di una variabile static
- questa variabile e' **condivisa** da tutte le istanze della classe
- questa variabile deve essere **inizializzata fuori** dalla dichiarazione della classe (e puo' essere modificata)

```
using namespace std;
#include <iostream>
class vettore
{
   public:
      double x;
      double y;
      static int count;   // <= count e' una variabile STATIC

      vettore (double a = 0, double b = 0)  // COSTRUTTORE
      {
         x = a;
         y = b;
         count++;  // ogni volta che viene chiamato, si aggiunge 1 a count
      }

      ~vettore()    // DISTRUTTORE
      {
         count--;  // ogni volta che viene chiamato si toglie 1 a count
      }
};

int vettore::count = 0; // Inizializzazione FUORI
                        // dalla definizione della classe

int main ()
{
   cout << "Quanti vettore ci sono:" << endl;

   vettore a;    // istanza di vettore, il costruttore aggiunge 1 a count!
   cout << vettore::count << endl;
```

```
    vettore b;    // altra istanza, count diventa 2!
    cout << vettore::count << endl;

    vettore *r, *u;  // due puntatori, il costruttore NON e' ancora chiamato
    r = new vettore; // new chiama il costruttore, count diventa 3
    cout << vettore::count << endl;

    u = new vettore; // new chiama il costruttore, count diventa 4
    cout << a.count << endl;
    delete r;        // delete chiama il DISTRUTTORE, count diventa 3
    cout << vettore::count << endl;

    delete u;        // altro delete, count diventa 2
    cout << b.count << endl;

    return 0;
}
```

```
Quanti vettore ci sono:
1
2
3
4
3
2
```

25· Variabili “const” in una classe

Un tipo diverso di variabile e' la const. Una variabile **const**:

- viene **definita all'interno** della classe
- **non** puo' piu' essere modificata

```
using namespace std;
#include <iostream>
class vettore
{
   public:
      double x;
      double y;
      const static double pi = 3.1415927; // <= pi greco non puo' essere
         modificato

      vettore (double a = 0, double b = 0)
      {
         x = a;
         y = b;
      }

      double cylinder_volume ()
      {
         return x * x / 4 * pi * y;
      }
};

int main()
{
   cout << "Il valore di pi greco: " << vettore::pi << endl << endl;

   vettore k (3, 4);
   cout << "Risultato: " << k.cylinder_volume() << endl;
   return 0;
}
```

```
Il valore di pi greco: 3.14159
```

```
Risultato: 28.2743
```

26· E' possibile DERIVARE un'altra classe da un'altra

Una classe puo' essere derivata da un'altra classe. La nuova classe, **eredita** (inherits) le variabili e i metodi dalla **base class**. Si possono aggiungere nuovi metodi e variabili alla classe appena creata:

```
using namespace std;
#include <iostream>
#include <cmath>
class vettore
{
   public:
      double x;
      double y;

      vettore (double a = 0, double b = 0)
      {
         x = a;
         y = b;
      }

     double modulo()
     {
        return sqrt (x*x + y*y);
     }

     double superficie()
     {
        return x * y;
    }
};

class trivettore: public vettore  // trivettore e' derivato da vettore
{
   public:
   double z;  // variabile in piu' rispetto a x e y di vettore

   trivettore (double m=0, double n=0, double p=0): vettore (m, n)
   {
      z = p; // il constructor di vettore verra'
             // chiamato PRIMA del costruttore di trivettore
             // con parametri m e n
```

```
    }

    trivettore (vettore a)// questo constructor indica cosa fare nel caso in cui
    {                     // venga fatto un cast da un vettore a un trivettore
       x = a.x;
       y = a.y;
       z = 0;
    }

    double modulo ()  // ri-definisce il modulo() per un trivettore
    {
       return sqrt (x*x + y*y + z*z);
    }

    double volume ()
    {
        return this->superficie() * z; // usa il metodo "superficie" di questa
           istanza
    }
};

int main ()
{
    vettore a (4, 5);
    trivettore b (1, 2, 3);
    cout << "a (4, 5)    b (1, 2, 3)  *r = b" << endl << endl;
    cout << "Superficie di a: " << a.superficie() << endl;
    cout << "Volume di b: " << b.volume() << endl;
    cout << "Superficie della base di b: " << b.superficie() << endl;
    cout << "Modulo di a: " << a.modulo() << endl;
    cout << "Modulo di b: " << b.modulo() << endl;
    cout << "Modulo di base di b: " << b.vettore::modulo() << endl;

    trivettore k;
    k = a;      // grazie alla definizione di trivettore(vettore)
                // copia di x e y,       k.z = 0
    vettore j;
    j = b;      // copia di x e y.       b.z non fa nulla

    vettore *r; // puntatore di un vettore
    r = &b;     // che pero' punta ad un TRIVECTOR
    cout << "Superficie di r: " << r->superficie() << endl;
    cout << "Modulo di r: " << r->modulo() << endl;

    return 0;
}
```

```
a (4, 5)  b (1, 2, 3)  *r = b
```

```
Superficie di a: 20
Volume di b: 6
Superficie della base di b: 2
Modulo di a: 6.40312
Modulo di b: 3.74166
Modulo della base di b: 2.23607
Superficie di r: 2
Modulo di r: 2.23607
```

27· Metodi virtuali

Se un metodo viene dichiarato virtuale, il programma controllera' sempre il tipo dell'istanza a cui punta e usera' il metodo appropriato.

Nel programma qui sopra, **r->modulo()** calcola il modulo del vettore, usando **x** e **y**, perche' **r** e' stato dichiarato come un puntatore ad un vettore. Il fatto che **r** in pratica punti ad un trivettore non viene preso in considerazione. Se vuoi che il programma controlli il tipo dell'oggetto puntato e scelga il metodo appropriato, allora bisogna dichiarare il metodo come **virtuale** all'interno della classe di base.

Se perlomeno uno dei metodi della classe di base e' virtuale, allora un "header" di 4 byte viene aggiunto ad ogni istanza della classe. Questo consente al programma di determinare a cosa punti un vettore (4 byte e' probabilmente specifico del compilatore usato, su una macchina a 64 bit forse sono 8 byte).

```
using namespace std;
#include <iostream>
#include <cmath>
class vettore
{
   public:
   double x;
   double y;

   vettore (double a = 0, double b = 0)
   {
      x = a;
      y = b;
   }
   virtual double modulo() //   metodo dichiarato come VIRTUALE
   {
      return sqrt (x*x + y*y);
   }
};

class trivettore: public vettore
{
   public:
      double z;
      trivettore (double m = 0, double n = 0, double p = 0)
      {
         x = m;  // Solo per fare un esempio pertinente
         y = n;  // qui NON viene chiamato il costruttore
```

```
            z = p;  // di vettore e si lascia che il costruttore
                    // di trivettore faccia tutto il lavoro.
                    // Il risultato non cambia
        }
        double modulo ()
        {
            return sqrt (x*x + y*y + z*z);
        }
};

void test (vettore &k)
{
    cout << "Risultato della funzione test:  " << k.modulo() << endl;
}

int main ()
{
    vettore a (4, 5);
    trivettore b (1, 2, 3);
    cout << "a (4, 5)  b (1, 2, 3)" << endl << endl;
    vettore *r;
    r = &a;
    cout << "modulo di vettore a: " << r->modulo() << endl;
    r = &b;
    cout << "modulo di trivettore b: " << r->modulo() << endl;
    test (a);
    test (b);

    vettore &s = b;
    cout << "modulo di trivettore b: " << s.modulo() << endl;
    return 0;
}
```

```
a (4, 5) b (1, 2, 3)

modulo di vettore a: 6.40312
modulo di trivettore b: 3.74166
Risultato della funzione test:
 6.40312
Risultato della funzione test:
 3.74166
modulo di trivettore b: 3.74166
```

28. Derivare una classe da piu' di una classe di base

Se vi state domandando se si possa derivare una classe da piu' di una classe di base, la risposta e' si':

```
using namespace std;
#include <iostream>
#include <cmath>
class vettore
{
   public:
      double x;
      double y;

      vettore (double a = 0, double b = 0)
      {
         x = a;
         y = b;
      }

      double superficie()
      {
         return fabs (x * y);
      }
};

class numero
{
   public:
      double z;
      numero (double a)
      {
         z = a;
      }

      int is_negative ()
      {
         if (z < 0) return 1;
         else
         return 0;
      }
};
```

```
class trivettore: public vettore, public numero
{
   public:
      trivettore(double a=0, double b=0, double c=0): vettore(a,b), numero(c)
      {
      }  // Il costruttore di trivettore chiama il
         // costruttore di vettore, poi il costruttore
         // di numero e in questo esempio non fa nulla
         // di piu'
      double volume()
      {
         return fabs (x * y * z);
      }
};

int main ()
{
   trivettore a(2, 3, -4);
   cout << a.volume() << endl;
   cout << a.superficie() << endl;
   cout << a.is_negative() << endl;

return 0;
}
```

```
24
6
1
```

29. Derivazione delle classi e metodi generici

La derivazione da classi gia' esistenti consente di costruire classi piu' complesse costruite a partire da una classe di base. Esiste un'altra applicazione della derivazione delle classi: consentire al programmatore di scrivere delle **generic functions**.

Supponiamo di definire una classe di base senza alcuna variabile. In pratica, non ha senso usare istanze di quella classe all'interno del proprio programma. Supponiamo pero' di avere costruito una funzione, il cui scopo sia quello di ordinare istanze proprio di quella classe. Questa funzione sara' in grado di ordinare qualunque tutti gli oggetti, che siano istanze di classi derivate dalla classe di base iniziale. La sola condizione e' che all'interno dei ognuna delle definizioni di classe derivata, tutti i metodi di cui ha bisogno la funzione di ordinamento siano definiti correttamente:

```
using namespace std;
#include <iostream>
#include <cmath>
class octopus                        // costruiamo una classe senza argomenti
{
   public:

      virtual double modulo() = 0;// =0 implica che la funzione non e' definita
                                  // Questo fa si' che istanze di questa
                                  // classe non possano essere dichiarate
};

double modulo_maggiore (octopus &a, octopus &b, octopus &c)
{
   double r = a.modulo();     // questa funzione restituisce come valore il modulo
   if (b.modulo() > r) r = b.modulo();  // della istanza com modulo
                                        // maggiore tra le 3
   if (c.modulo() > r) r = c.modulo();
   return r;
}

class vettore: public octopus    // costruisco una nuova classe
                                 // che eredita da octopus
{
   public:
   double x;
   double y;

   vettore (double a = 0, double b = 0)
```

```
    {
        x = a;
        y = b;
    }

    double modulo()            // ha un metodo modulo come la classe di base
    {
        return sqrt (x * x + y * y);
    }
};

class numero: public octopus  // costruisco un'altra classe derivata da octopus
{
    public:
        double n;
        numero (double a = 0)
        {
            n = a;
        }
        double modulo()        // anche questa ha un metodo chiamato modulo
        {
            if (n >= 0) return n;
            else        return -n;
        }
};

int main ()
{
    vettore k (1,2), m (6,7), n (100, 0);
    numero p (5),  q (-3), r (-150);
    cout << modulo_maggiore (k, m, n) << endl; // accetta tutte le classi
    cout << modulo_maggiore (p, q, r) << endl; // che sono derivate da octopus
    cout << modulo_maggiore (p, q, n) << endl; // quindi anche vettore e numero

    return 0;
}
```

```
100
150
100
```

Una persona potrebbe pensare: "ok, l'idea di derivare classi dalla classe **octopus** e' buona perche', in quel modo posso applicare loro istanze dei metodi della mia classe e funzioni che erano state progettate in modo generico per la classe **octopus**. Cosa potrebbe succedere se ci fosse un'altra classe di base, chiamata **cuttlefish**, che ha dei metodi e delle funzioni molto interessanti? E' forse necessario scegliere tra **octopus** e **cuttlefish** quando si vuole derivare una classe¿' No chiaramente no!. Una classe derivata puo' essere derivata sia da **octopus** che da **cuttlefish**.

Questo e' **POLIMORFISMO**

La classe derivata, semplicemente, deve definire sia metodi necessari per **octopus** che per **cuttlefish**:

```
class octopus
{
   virtual double modulo() = 0;
};

class cuttlefish
{
   virtual int test() = 0;
};

class vettore: public octopus, public cuttlefish
{
   double x;
   double y;
   double modulo ()
   {
      return sqrt (x * x + y * y);
   }

   int test ()
   {
      if (x > y) return 1;
      else       return 0;
   }
}
```

30· Encapsulation: public, protected e private

- La direttiva **public**, significa che, ovunque nel programma, si puo' accedere ed usare le variabili e i metodi scritti di seguito alla direttiva stessa.
- La direttiva **protected**, significa che le variabili e i metodi scritti sotto ad essa sono accedibili ed usabili solo tramite metodi della classe stessa **E** metodi di classi derivate da essa.
- La direttiva **private** e' ancora piu' restrittiva, e' possibile accedere alle variabili ed ai metodi della classe stessa (non da quelle derivate o dall'esterno).

Il fatto che le variabili o i metodi siano dichiarati **private** or **protected** significa che nulla esterno alla classe puo' accedere a loro o usarli. Questo tipo di caratteristica si chiama **ENCAPSULATION** (se si vuole dare ad una funzione il diritto di accesso a quei metodi o a quelle variabili, allora e' necessario includere il **prototype** della funzione all'interno della definizione di classe, con davanti la keyword: **friend**.

E' considerata una buona pratica di programmazione il fatto di fare "encapsulation" per tutte le variabili di una classe. Questo comportamento puo' sembrare strano se si e' abituati agli **struct**, in C. In effetti uno **struct** ha senso solo se e' possibile accedere a tutti i dati al suo interno. In C++, invece, si deve creare un metodo apposito per accedere ai dati all'interno di una classe. L'esempio seguente usa l'esempio di base del capitolo 17, ma dichiara i dati della classe come **protected**:

```
using namespace std;
#include <iostream>
class vettore
{
   protected:
     double x;
     double y;

   public:
      void definisci_x (int n)
      {
         x = n;
      }

      void definisci_y (int n)
      {
         y = n;
      }
```

```
        double superficie ()
        {
            double s;
            s = x * y;
            if (s < 0) s = -s;
            return s;
        }
};

int main ()
{
    vettore a;
    a.definisci_x (3);
    a.definisci_y (4);
    cout << "La superficie di a: " << a.superficie() << endl;

    return 0;
}
```

```
La superficie di a: 12
```

L'esempio di cui sopra, e' un po' strano dato che i parametri x e y della classe possono essere inizializzati, ma non letti! Qualunque tentativo, all'interno del main() di leggere **a.x** o **a.y** portera' ad un errore di compilazione. Nell'esempio seguente, invece si potra' anche leggere sia x che y:

```
using namespace std;
#include <iostream>
class vettore
{
    protected:
        double x;
        double y;

    public:
        void definisci_x (int n)
        {
            x = n;
        }

        void definisci_y (int n)
        {
            y = n;
        }

        double ottieni_x ()
        {
            return x;
        }
```

```
        double ottieni_y ()
        {
            return y;
        }

        double superficie ()
        {
            double s;
            s = x * y;
            if (s < 0) s = -s;
            return s;
        }
};

int main ()
{
    vettore a;
    a.definisci_x(3);
    a.definisci_y(4);

    cout << "La superficie di a: " << a.superficie() << endl;
    cout << "La larghezza di a:  " << a.ottieni_x() << endl;
    cout << "L'altezza di a: " << a.ottieni_y() << endl;

    return 0;
}
```

```
La superficie di a: 12
La larghezza di a: 3
L'altezza di a: 4
```

In C++ una persona non dovrebbe poter accedere ai dati di una classe direttamente. Si dovrebbero dichiarare dei metodi appositi. Perche' viene richiesto questo comportamento? Ci sono molte ragioni. Una di queste e' che in questo modo si consente di cambiare il modo in cui i dati sono rappresentati all'interno della classe. Un'altra ragione e' che quasto consente ai dati all'interno della classe di essere "cross-dependent". Supponiamo che x e y debbano sempre avere lo stesso segno, altrimenti questo causi dei problemi... Se fosse consentito accedere ai dati della classe direttamente, sarebbe possibile e semplice imporre per esempio che x sia positivo e negativo. Nell'esempio qui sotto questo e' controllato strettamente.

```
using namespace std;
#include <iostream>
int sign (double n)
{
    if (n >= 0) return 1;
    return -1;
}
class vettore
{
```

```
    protected:
        double x;
        double y;

    public:
        void definisci_x (int n)
        {
            x = n;
            if (sign (x) != sign(y)) y = -y;
        }

        void definisci_y (int n)
        {
            y = n;
            if (sign (y) != sign(x)) x = -x;
        }

        double ottieni_x ()
        {
            return x;
        }

        double ottieni_y ()
        {
            return y;
        }

        double superficie ()
        {
            double s;
            s = x * y;
            if (s < 0) s = -s;
            return s;
        }
};

int main ()
{
    vettore a;
    a.definisci_x(3);
    a.definisci_y(4);

    cout << "La superficie di a: " << a.superficie() << endl;
    cout << "La larghezza di a: " << a.ottieni_x() << endl;
    cout << "L'altezza di a: " << a.ottieni_y() << endl;

    return 0;
}
```

```
La superficie di a: 12
La larghezza di a: 3
L'altezza di a: 4
```

31. Brevi cenni all' Input e output di file

Parliamo ora di input/output. Questo e' un argomento molto vasto per quanto riguarda il C++. Nel seguito c'e' un programma che scrive un file:

```
using namespace std;
#include <iostream>
#include <fstream>
int main ()
{
   fstream f;
   f.open("test.txt", ios::out);
   f << "Questo testo e' inviato in output ad un file." << endl;
   double a = 345;
   f << "Il numero a: " << a << endl;
   f.close();

   return 0;
}
```

```
Il contenuto del file test.txt:
Questo testo e' inviato in output ad un file.
Il numero a: 345
```

Qui invece c'e' il programma che legge da file:

```
using namespace std;
#include <iostream>
#include <fstream>
int main ()
{
   fstream f;
   char c;

   cout << "Cosa c'e' nel file test.txt" << endl;
   cout << endl;

   f.open("test.txt", ios::in);
   while (! f.eof() )
   {
```

```
        f.get(c);
        cout << c;
    }

    f.close();
    return 0;
}
```

```
Questo testo e' inviato in output ad un file.
Il numero a: 345
```

32· Array di character possono essere usati come file

In generale e' possibile fare le stesse operazioni sugli array di character che sui file. Questo risulta molto utile per convertire dati o per gestire array di memoria.

Qui c'e' un programma che scrive su un array di character:

```
using namespace std;
#include <iostream>
#include <strstream>
#include <cstring>
#include <cmath>
int main ()
{
   char a[1024];
   ostrstream b(a, 1024);
   b.seekp(0);                          // Inizia dal primo char.
   b << "2 + 2 = " << 2 + 2 << ends; // OCCHIO usa il comando "ends", non "endl"
                                        // "ends" e' semplicemente il
                                        // carattere null o '\0'

   cout << a << endl;
   double v = 2;
   strcpy (a, "A sinus: ");
   b.seekp(strlen (a));
   b << "sin (" << v << ") = " << sin(v) << ends;
   cout << a << endl;
   return 0;
}
```

```
2 + 2 = 4
A sinus: sin (2) = 0.909297
```

Un programma che legge da una stringa di caratteri:

```
using namespace std;
#include <iostream>
#include <strstream>
#include <cstring>
int main ()
{
```

```
    char a[1024];
    istrstream b(a, 1024);
    strcpy (a, "45.656");
    double k, p;
    b.seekg(0);  // Inizia dal primo carattere.
    b >> k;
    k = k + 1;
    cout << k << endl;
    strcpy (a, "444.23 56.89");
    b.seekg(0);
    b >> k >> p;
    cout << k << ", " << p + 1 << endl;
    return 0;
}
```

```
46.656
444.23  57.89
```

33· Un esempio di output formattato

Questo programma produce un output formattato in due modi. Si noti che i MODIFICATORI **width()** e **setw()** hanno effetto SOLO sul prossimio output dello stream. Elementi successivi non verranno influenzati dai MODIFICATORI.

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

int main ()
{
   int i;
   cout << "Una lista di numeri:" << endl;
   for (i = 1; i <= 1024; i *= 2)
   {
      cout.width (7);
      cout << i << endl;
   }

   cout << "Una tabella di numeri:" << endl;
   for (i = 0; i <= 4; i++)
   {
      cout << setw(3) << i << setw(5) << i * i * i << endl;
   }

   return 0;
}
```

```
Una lista di numeri:
      1
      2
      4
      8
     16
     32
     64
    128
    256
    512
```

```
1024
Una tabella di numeri:
0 0
1 1
2 8
3 27
4 64
```

A questo punto avete una conoscenza di base del C++. Da un buon libro potrete imparare molte altre cose. Il *file management system* e' molto potente, e ha a disposizione molte altre possibilita' rispetto a quelle illustrate in questa guida. C'e' molto altro da dire riguardo alle classi, **classi template**, **clssi virtual**, ... Per poter lavorare efficacemente con il C++ avrete bisogno di un buon libro di riferimento, proprio come succede con il C. Avrete inoltre bisogno di sapere come il C++ viene usato nel vostro particolare dominio di lavoro. Gli standard, l'approccio globale, i trucchi, i tipici problemi che si incontrano e le loro soluzioni... La migliore referenza sono ovviamente i libri scritti da Bjarne Stroutrup (non ricordo quale io abbia letto). Il seguente libro contiene praticamente ogni dettaglio del C e C++, ed e' costruito in modo molto simile a questo testo e contiene un CD:

```
Jamsa's C/C++ Programmer's Bible
&copyright; 1998 Jamsa Press
Las Vegas, United States

French edition:
C/C++ La Bible du programmeur
Kris Jamsa, Ph.D - Lars Klander
France : Editions Eyrolles
www.eyrolles.com
Canada : Les Editions Reynald Goulet inc.
www.goulet.ca
ISBN 2-212-09058-7
```

Questo e' ormai obsoleto ed e' ora:

```
It has been obsoleted and is now:
Jamsa's C/C++/C# Programmer's Bible
Onword Press
```

Altre referenze:

accu:`www.accu.org/bookreviews/public/reviews/0hr/index.htm`
CoderSource.net: `www.codersource.net/`
C++ Guide: `google-styleguide.googlecode.com/svn/trunk/cppguide.xml`
C++ Reference: `fresh2refresh.com/c/c-language-history`
A similar tutorial for Ada is available at `www.adahome.com/Ammo/cpp2ada.html`
A Haskell tutorial by a C programmer: `learnyouahaskell.com`

Desidero ringraziare Didier Bizzarri, Toni Ronkko, Frédéric Cloth, Jack Lam, Morten Brix Pedersen, Elmer Fittery, Ana Yuseepi, William L. Dye, Bahjat F. Qaqish, Muthukumar Veluswamy, Marco Cimarosti, Jarrod Miller,

Nikolaos Pothitos, Ralph Wu, Dave Abercrombie, Alex Pennington, Scott Marsden, Robert Krten, Dave Panter, Cihat Imamoglu, Bohdan Zograf, David L. Markowitz, Marko Pozner, Filip Zaludek, Kevin Wheeler e Patrick Einheber per la loro ispirazione, suggerimenti, aiuti, dati, controllo sui bug, referenze, miglioramento della versione inglese e traduzione.

Eric Brasseur - 23 Febbraio 1998 fino al 25 Dicembre 2016