C++ tutorial for C users

This text is aimed at C users who wish to learn C++. It is also interesting for experienced C++ users who leaved out some features of the language. The possibilities of C++ are briefly enunciated and illustrated. When you will try to use them for your own programs you will encounter a lot of problems and compilation errors. Come back to this text and look carefully at the examples. Make use of the help files of your development environment. Do not hesitate to copy the examples from this text and paste them inside your development environment in order to test and modify them.

You can use / / **to type a remark :**

```
#include <iostream.h>        // This library is often used

void main ()                 // The program's main routine.
{
   double a;                 // Declaration of variable a.

   a = 456;
   a = a + a * 21.5 / 100;   // A calculation.

   cout << a;                // Display the content of a.
}
```

Input from keyboard and output to screen can be performed through cout << **and** cin >> **:**

```
#include <iostream.h>

void main()
{
```

```
   int a;                       // a is an integer variable
   char s [100];                // s points to a sring of 99 characters

   cout << "This is a sample program." << endl;

   cout << endl;                // Line feed (end of line)

   cout << "Type your age  : ";
   cin >> a;

   cout << "Type your name : ";
   cin >> s;

   cout << endl;

   cout << "Hello " << s << " you're " << a << " old." << endl;
   cout << endl << endl << "Bye !" << endl;
}
```

Variables can be declared everywhere :

```
#include <iostream.h>

void main ()
{
   double a;

   cout << "Hello, this is a test program." << endl;

   cout << "Type parameter a : ";
   cin >> a;
```

```
   a = (a + 1) / 2;

   double c;

   c = a * 5 + 1;

   cout << "c contains       : " << c << endl;

   int i, j;

   i = 0;
   j = i + 1;

   cout << "j contains       : " << j << endl;
}
```

A variable can be initialised by a calculation :

```
#include <iostream.h>

void main ()
{
   double a = 12 * 3.25;
   double b = a + 1.112;

   cout << "a contains : " << a << endl;
   cout << "b contains : " << b << endl;

   a = a * 2 + b;

   double c = a + b * a;
```

```
   cout << "c contains : " << c << endl;
}
```

Like in C, variables can be encapsulated between hooks :

```
#include <iostream.h>

void main()
{
   double a;

   cout << "Type a number   : ";
   cin >> a;

   {
      int a = 1;
      a = a * 10 + 4;
      cout << "Local number    : " << a << endl;
   }

   cout << "You typed       : " << a << endl;
}
```

C++ allows to declare a variable inside the for loop declaration. It's like if the variable had been declared just before the loop :

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

```
void main ()
{
   for (int i = 0; i < 4; i++)
   {
      cout << i << endl;
   }

   cout << "i contains : " << i << endl;

   for (i = 0; i < 4; i++)
   {
      for (int i = 0; i < 4; i++)        // we're between
      {                                  // previous for's hooks
         cout << i;
      }
      cout << endl;
   }
}
```

A global variable can be accessed even if another variable with the same name has been declared inside the function :

```
#include <iostream.h>

double a = 128;

void main ()
{
   double a = 256;

   cout << "Local a  : " << a << endl;
```

```
   cout << "Global a : " << ::a << endl;
}
```

It is possible to make one variable be another :

```
#include <iostream.h>

void main ()
{
   double a = 3.1415927;

   double &b = a;                              // b IS a

   b = 89;

   cout << "a contains : " << a << endl;     // Displays 89.
}
```

(If you are used at pointers and absolutely want to know what happens, simply think double &b = a **is translated to double** *b = &a **and all subsequent** b **are replaced by** *b**.)**

The value of REFERENCE b **cannot be changed afther its declaration. For example you cannot write, a few lines further,** &b = c **expecting now** b **IS** c**. It won't work.**

Everything is said on the declaration line of b**. Reference** b **and variable** a **are married on that line and nothing will separate them.**

References can be used to allow a function to modify a calling variable :

```
#include <iostream.h>

void change (double &r, double s)
```

```
{
   r = 100;
   s = 200;
}

void main ()
{
   double k, m;

   k = 3;
   m = 4;

   change (k, m);

   cout << k << ", " << m << endl;        // Displays 100, 4.
}
```

If you are used at pointers in C and wonder how exactly the program above works, here is how the C++ compiler translates it (those who are not used at pointers, please skip this ugly piece of code) :

```
#include <iostream.h>

void change (double *r, double s)
{
   *r = 100;
   s = 200;
}

void main ()
{
   double k, m;
```

```
   k = 3;
   m = 4;

   change (&k, m);

   cout << k << ", " << m << endl;        // Displays 100, 4.
}
```

A reference can be used to let a function return a variable :

```
#include <iostream.h>

double &biggest (double &r, double &s)
{
   if (r > s) return r;
   else       return s;
}

void main ()
{
   double k = 3;
   double m = 7;

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

   biggest (k, m) = 10;

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

```
   cout << endl;

   biggest (k, m) ++;

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

Again, provided you're used at pointer arithmetics and if you wonder how the program above works, just think the compiler translated it into the following standard C program :

```
#include <iostream.h>

double *biggest (double *r, double *r)
{
   if (*r > *s) return r;
   else         return s;
}

void main ()
{
   double k = 3;
   double m = 7;

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

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

   cout << "k : " << k << endl;
```

```
   cout << "m : " << m << endl;
   cout << endl;

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

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

To end with, for people who have to deal with pointers yet do not like it, references are very useful to un-pointer variables :

```
#include <iostream.h>

double *silly_function ()    // This function returns a pointer to a double
{
   static double r = 342;
   return &r;
}

void main()
{
   double *a;

   a = silly_function();

   double &b = *a;          // Now b IS the double towards which a points !

   b += 1;                  // Great !
   b = b * b;               // No need to write *a everywhere !
   b += 4;
```

```
   cout << "Content of *a, b, r : " << b << endl;
}
```

If they contain just simple lines of code, use no for loops or the like, C++ functions can be declared INLINE. This means their code will be inserted right everywhere the function is used. That's somehow like a macro. Main advantage is the program will be faster. A little drawback is it will be bigger, because the full code of the function was inserted everywhere it is used :

```
#include <iostream.h>
#include <math.h>

inline double hypothenuse (double a, double b)
{
   return sqrt (a * a + b * b);
}

void main ()
{
   double k = 6, m = 9;

   // Next two lines produce exactly the same code :

   cout << hypothenuse (k, m) << endl;
   cout << sqrt (k * k + m * m) << endl;
}
```

You know the classical structures of C : for, if, do, while, switch... C++ adds one more structure named EXCEPTION :

```
#include <iostream.h>
#include <math.h>

void main ()
{
   int a, b;

   cout << "Type a number : ";
   cin >> a;
   cout << endl;

   try
   {
      if (a > 100) throw 100;
      if (a < 10)  throw 10;
      throw a / 3;
   }
   catch (int result)
   {
      cout << "Result is     : " << result << endl;
      b = result + 1;
   }

   cout << "b contains    : " << b << endl;

   cout << endl;

   // another example of exception use :

   char zero[] = "zero";
   char pair[] = "pair";
   char notprime[] = "not prime";
   char prime[] = "prime";
```

```
   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 << "The number you typed is "<< conclusion << endl;
   }

   cout << endl;

}
```

It is possible to define default parameters for functions :

```
#include <iostream.h>

double test (double a, double b = 7)
{
   return a - b;
}

void main ()
{
   cout << test (14, 5) << endl;
   cout << test (14) << endl;
```

```
}
```

One important advantage of C++ is the "operators overload". Different functions can have the same name provided something allows to distinguish between them : number of parameters, type of parameters...

```
#include <iostream.h>

double test (double a, double b)
{
   return a + b;
}

int test (int a, int b)
{
   return a - b;
}

void main ()
{
   double   m = 5,  n = 3;
   int      k = 5,  p = 3;

   cout << test(m, n) << " , " << test(k, p) << endl;
}
```

The "operators overload" can be used to define the basic symbolic operators for new sorts of parameters :

```
#include <iostream.h>

struct vector
{
   double x;
   double y;
};

vector operator * (double a, vector b)
{
   vector r;

   r.x = a * b.x;
   r.y = a * b.y;

   return r;
}

void main ()
{
   vector k, m;              // No need to type "struct vector"

   k.x =  2;                 // Keep cool, soon you'll be able
   k.y = -1;                 // to write k = vector (45, -4).

   m = 3.1415927 * k;        // Magic !

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

Besides multiplication, 43 other basic C++ operators can be overloaded, including +=, ++, the matrix [], and so on...

The operation cout << **is an overload of the binary shift of integers. That way the** << **symbol is used a completely different way. It is thus possible to define the output of vectors :**

```
#include <iostream.h>

struct vector
{
   double x;
   double y;
};

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

void main ()
{
   vector a;

   a.x = 35;
   a.y = 23;

   cout << a << endl;
}
```

The keywords new **and** delete **can be used to allocate and deallocate memory. They are much more sweet than the functions** malloc **and** free **from standard C :**

```
#include <iostream.h>
#include <string.h>
```

```
void main ()
{
   double *d;                          // d is a variable whose purpose
                                       // is to contain the address of a
                                       // zone where a double is located


   d = new double;                     // new allocates a zone of memory
                                       // large enough to contain a double
                                       // and returns its address.
                                       // That address is stored in d.

   *d = 45.3;                          // The number 45.3 is stored
                                       // inside the memory zone
                                       // whose address is given by d.

   cout << "Type a number   : ";
   cin >> *d;

   *d = *d + 5;

   cout << "Result          : " << *d << endl;

   delete (d);                         // delete deallocates the
                                       // zone of memory whose address
                                       // is given by pointer d.
                                       // Now we can no more use that zone.


   d = new double[15];                 // allocates a zone for an array
                                       // of 15 doubles

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

```
   cout << "Content of d[1] : " << d[1] << endl;

   delete (d);



   int n = 30;

   d = new double[n];                // new can be used to allocate an
                                     // array of random size.
   for (int i = 0; i < n; i++)
   {
      d[i] = i;
   }

   delete (d);



   char *s;

   s = new char[100];

   strcpy (s, "Hello !");

   cout << s << endl;

   delete (s);

}
```

What is a CLASS ? Well, that's a struct yet with more possibilities. METHODS can be defined. They are C++ functions dedicated to the class. Here is an example of such a class definition :

```
#include <iostream.h>

class vector
{
public:

   double x;
   double y;

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

void main(void)
{
   vector a;

   a.x = 3;
   a.y = 4;

   cout << "The surface of a : " << a.surface() << endl;
}
```

In the example above, a **is an INSTANCE of the class "vector".**

Just like a function, a method can be an overload of any C++ operator, have any number of parameters (yet one parameter is always implicit : the instance it acts upon), return any type of

parameter, or return no parameter at all.

A method is allowed to change the variables of the instance it is acting upon :

```
#include <iostream.h>

class vector
{
public:

   double x;
   double y;

   vector its_oposite()
   {
      vector r;

      r.x = -x;
      r.y = -y;

      return r;
   }

   void be_oposited()
   {
      x = -x;
      y = -y;
   }

   void be_calculated (double a, double b, double c, double d)
   {
      x = a - c;
      y = b - d;
   }
```

```
   vector operator * (double a)
   {
      vector r;

      r.x = x * a;
      r.y = y * a;

      return r;
   }
};

void main (void)
{
   vector a, b;

   a.x = 3;
   b.y = 5;

   b = a.its_oposite();

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

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

   a.be_calculated (7, 8, 3, 2);
   cout << "Vector a : " << a.x << ", " << a.y << endl;

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

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

```
   cout << "x of oposite of a : " << a.its_oposite().x << endl;
}
```

Very special and essential methods are the CONSTRUCTORS and DESTRUCTORS. They are automatically called whenever an instance of a class is created or destroyed.

The constructor will initialize the variables of the instance, do some calculation, allocate some memory for the instance, output some text... whatever is needed.

Here is an example of a class definition with two overloaded constructors.

```
#include <iostream.h>

class vector
{
public:

   double x;
   double y;

   vector ()                    // same name as class
   {
      x = 0;
      y = 0;
   }

   vector (double a, double b)
   {
      x = a;
      y = b;
   }
```

```
};

void main ()
{
   vector k;                    // vector () is called

   cout << "vector k : " << k.x << ", " << k.y << endl << endl;

   vector m (45, 2);            // vector (double, double) is called

   cout << "vector m : " << m.x << ", " << m.y << endl << endl;

   k = vector (23, 2);          // vector created, copied to k, then
erased

   cout << "vector k : " << k.x << ", " << k.y << endl << endl;
}
```

It is a good practice to try not to overload the constructors. Best is to declare only one constructor and give it default parameters wherever possible :

```
#include <iostream.h>

class vector
{
public:

   double x;
   double y;

   vector (double a = 0, double b = 0)
   {
```

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

void main ()
{
   vector k;
   cout << "vector k : " << k.x << ", " << k.y << endl << endl;

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

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

The destructor is often not necessary. You can use it to do some calculation whenever an instance is destroyed or output some text for debugging. But if variables of the instance point towards some allocated memory then the role of the destructor is essential : it must free that memory ! Here is an example of such an application :

```
#include <iostream.h>
#include <string.h>

class person
{
public:

   char *name;
   int age;

   person (char *n = "no name", int a = 0)
```

```
   {
      name = new char[100];                // better than malloc !
      strcpy (name, n);
      age = a;
      cout << "Instance initialized, 100 bytes allocated" << endl;
   }

   ~person ()                              // The destructor
   {
      delete (name);                       // instead of free !
      cout << "Instance going to be deleted, 100 bytes freed" << endl;
   }
};

void main ()
{
    cout << "Hello !" << endl << endl;

    person a;
    cout << a.name << ", age " << a.age << endl << endl;

    person b ("John");
    cout << b.name << ", age " << b.age << endl << endl;

    b.age = 21;
    cout << b.name << ", age " << b.age << endl << endl;

    person c ("Miki", 45);
    cout << c.name << ", age " << c.age << endl << endl;

   cout << "Bye !" << endl << endl;
}
```

If you cast an object like a vector, everything will happen all right. For example if vector k **contains** (4, 7)**, afther the cast** m = k **the vector** m **will contain** (4, 7) **too. Now suppose you're playing with objects like the person class above. If you cast such person object** p**,** r **by writing** p = r **it is necesary that some function does the necessary work to make** p **be a correct copy of** r**. Otherwise the result will be catastrophic; a mess of pointers and lost data. The method that will do that job is the COPY CONSTRUCTOR :**

```
#include <iostream.h>
#include <string.h>

class person
{
public:

   char *name;
   int age;

   person (char *n = "no name", int a = 0)
   {
      name = new char[100];
      strcpy (name, n);
      age = a;
   }

   person (person &s)               // The COPY CONSTRUCTOR
   {
      strcpy (name, s.name);
      age = s.age;
   }

   ~person ()
   {
      delete (name);
   }
};
```

```
void main ()
{
   person p;
   cout << p.name << ", age " << p.age << endl << endl;

   person k ("John", 56);
   cout << k.name << ", age " << k.age << endl << endl;

   p = k;
   cout << p.name << ", age " << p.age << endl << endl;

   p = person ("Bob", 10);
   cout << p.name << ", age " << p.age << endl << endl;
}
```

In all the examples above the methods are defined inside the class definition. That makes them automatically be inline methods.

If a method cannot be inline, if you do not want it to be inline or if you want the class definition contain the minimum of information, then you must just put the prototype of the method inside the class and define the method below the class :

```
#include <iostream.h>

class vector
{
public:

   double x;
   double y;

   double surface();         // The ; and no {} shows it is a prototype
```

```
};

double vector::surface()
{
   double s = 0;

   for (double i = 0; i < x; i++)
   {
      s = s + y;
   }

   return s;
}

void main ()
{
   vector k;

   k.x = 4;
   k.y = 5;

   cout << "Surface : " << k.surface() << endl;
}
```

When a method is applied to an instance, that method may use the instance's variables, modify them... But sometimes it is necessary to know the address of the instance. No problem, the keyword "this" **is intended therefore :**

```
#include <iostream.h>
#include <math.h>

class vector
{
```

```
public:

   double x;
   double y;

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

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

   void set_length (double a = 1)
   {
      double length;

      length = this->module();

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

void main ()
{
   vector c (3, 5);

   cout << "The module of vector c : " << c.module() << endl;

   c.set_length(2);            // Transforms c in a vector of size 2.
```

```
   cout << "The module of vector c : " << c.module() << endl;

   c.set_length();             // Transforms b in an unitary vector.

   cout << "The module of vector c : " << c.module() << endl;
}
```

Of course it is possible to declare arrays of objects :

```
#include <iostream.h>
#include <math.h>

class vector
{
public:

   double x;
   double y;

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

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

```
void main ()
{
   vector s[1000];

   vector t[3] = {vector(4, 5), vector(5, 5), vector(2, 4)};

   s[23] = t[2];

   cout << t[0].module() << endl;
}
```

Here is an example of a full class declaration :

```
#include <iostream.h>
#include <math.h>

class vector
{
public:

   double x;
   double y;

   vector (double = 0, double = 0);

   vector operator + (vector);
   vector operator - (vector);
   vector operator - ();
   vector operator * (double a);
   double module();
   void set_length (double = 1);
```

```
};

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

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

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

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

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

double vector::module()
{
   return sqrt (x * x + y * y);
}

void vector::set_length (double a)
{
```

```
   double length = this->module();

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

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

void main ()
{
   vector a;
   vector b;
   vector c (3, 5);

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

   cout << "The module of vector c  : " << c.module() << endl;

   cout << "The content of vector a : " << a << endl;
   cout << "The oposite of vector a : " << -a << endl;

   c.set_length(2);            // Transforms c in a vector of size 2.

   a = vector (56, -3);
   b = vector (7, c.y);

   b.set_length();             // Transforms b in an unitary vector.
```

```
   cout << "The content of vector b : " << b << endl;


   double k;
   k = vector(1, 1).module();  // k will contain 1.4142.
   cout << "k contains             : " << k << endl;
}
```

It is also possible to define the sum of vectors without mentioning it inside the vector class definition. Then it will not be a method of the class vector. Just a function that uses vectors :

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

In the example above of a full class definition, the multiplication of a vector by a double is defined. Suppose we want the multiplication of a double by a vector be defined too. Then we must to write an isolated function outside the class :

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

Of course the keywords new **and** delete **work for class instances too. What's more,** new **automatically calls the constructor in order to initialize the objects, and** delete **automatically calls the destructor before deallocating the zone of memory the instance variables take :**

```
#include <iostream.h>
#include <math.h>

class vector
{
public:

   double x;
   double y;

   vector (double = 0, double = 0);

   vector operator + (vector);
   vector operator - (vector);
   vector operator - ();
   vector operator * (double);
   double module();
   void set_length (double = 1);
};

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

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

vector vector::operator - (vector a)
{
```

```
   return vector (x - a.x, y - a.y);
}

vector vector::operator - ()
{
    return vector (-x, -y);

}

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

double vector::module()
{
   return sqrt (x * x + y * y);
}

void vector::set_length (double a)
{
   vector &the_vector = *this;

   double length = the_vector.module();

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

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

```
void main ()
{
   vector c (3, 5);

   vector *r;                  // r is a pointer to a vector.

   r = new vector;             // new allocates the memory necessary
   cout << *r << endl;         // to hold a vectors' variable,
                               // calls the constructor who will
                               // initialize it to 0, 0. Then finally
                               // new returns the address of the vector.

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

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

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

   *r = (-c * 3  +  -*r * 4) * 5;
   cout << *r << endl;

   delete (r);                 // Calls the vector destructor then
                               // frees the memory.

   r = &c;                     // r points towards vector c
   cout << *r << endl;

   r = new vector (78, 345);   // Creates a new vector.
   cout << *r << endl;         // The constructor will initialise
```

```
                                // the vector's x and y at 78 and 345

   cout << "x component of r : " << r->x << endl;
   cout << "x component of r : " << (*r).x << endl;

   delete (r);

   r = new vector[4];           // creates an array of 4 vectors

   r[3] = vector (4, 5);
   cout << r[3].module() << endl;

   delete (r);                  // deletes the array

   int n = 5;
   r = new vector[n];           // Cute !

   r[1] = vector (432, 3);
   cout << r[1] << endl;

   delete (r);

}
```

A class' variable can be declared STATIC. Then only one instance of that variable exists, shared by all instances of the class :

```
#include <iostream.h>

class vector
{
public:
```

```
   double x;
   double y;
   static int count;

   vector (double a = 0, double b = 0)
   {
      x = a;
      y = b;
      count = count + 1;
   }

   ~vector()
   {
      count = count - 1;
   }
};

void main ()
{
   vector::count = 0;

   cout << "Number of vectors :" << endl;

   vector a;
   cout << vector::count << endl;

   vector b;
   cout << vector::count  << endl;

   vector *r, *u;

   r = new vector;
   cout << vector::count << endl;
```

```
   u = new vector;
   cout << a.count << endl;

   delete (r);
   cout << vector::count << endl;

   delete (u);
   cout << b.count << endl;
}
```

A class variable can also be CONSTANT. That's just like static, except it is alocated a value inside the class declaration and that value may not be modified :

```
#include <iostream.h>

class vector
{
public:

   double x;
   double y;
   const double pi = 3.1415927;

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

   double cilinder_volume ()
   {
      return x * x / 4 * pi * y;
```

```
   }
};

void main(void)
{
   cout << "The value of pi : " << vector::pi << endl << endl;

   vector k (3, 4);

   cout << "Result : " << k.cilinder_volume() << endl;
}
```

A class may be DERIVED from another class. The new class INHERITS the variables and methods of the BASE CLASS. Additional variables and/or methods can be added :

```
#include <iostream.h>
#include <math.h>

class vector
{
public:

   double x;
   double y;

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

   double module()
```

```
   {
      return sqrt (x*x + y*y);
   }

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

class trivector : public vector   // trivector is derived from vector
{
public:
   double z;                      // added to x and y from vector

   trivector (double m=0, double n=0, double p=0) : vector (m, n)
   {
      z = p;                      // Vector constructor will
   }                              // be called before trivector
                                  // constructor, with parameters
                                  // m and n.

   trivector (vector a)           // What to do if a vector is
   {                              // cast to a trivector
      x = a.x;
      y = a.y;
      z = 0;
   }

   double module ()               // define module() for trivector
   {
      return sqrt (x*x + y*y + z*z);
   }
```

```
    double volume ()
    {
        return this->surface() * z;         // or x * y * z
    }
};

void main()
{
    vector a (4, 5);
    trivector b (1, 2, 3);

    cout << "a (4, 5)    b (1, 2, 3)    *r = b" << endl << endl;

    cout << "Surface of a         : " << a.surface() << endl;
    cout << "Volume of b          : " << b.volume() << endl;
    cout << "Surface of base of b : " << b.surface() << endl;

    cout << "Module of a          : " << a.module() << endl;
    cout << "Module of b          : " << b.module() << endl;
    cout << "Module of base of b  : " << b.vector::module() << endl;

    trivector k;
    k = a;               // thanks to trivector(vector) definition
                         // copy of x and y,       k.z = 0
    vector j;
    j = b;               // copy of x and y.       b.z leaved out

    vector *r;
    r = &b;

    cout << "Surface of r         : " << r->surface() << endl;
    cout << "Module of r          : " << r->module() << endl;
}
```

In the program above, r->module() **calculates the vector module, using** x **and** y**, because** r **has been declared a vector pointer. The fact** r **actually points towards a trivector is not taken into account. If you want the program to check the type of the pointed object and choose the appropriate method, then you must declare that method VIRTUAL inside the base class.**

(If at least one of the methods of the base class is virtual then a "header" of 4 bytes is added to every instance of the classes. This allows the program to determine towards what a vector actually points.)

```
#include <iostream.h>
#include <math.h>

class vector
{
public:

   double x;
   double y;

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

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

class trivector : public vector
{
public:
   double z;
```

```
   trivector (double m = 0, double n = 0, double p = 0)
   {
      x = m;                  // Just for the game,
      y = n;                  // here I do not call the vector
      z = p;                  // constructor and I make the
   }                          // trivector constructor do the
                              // whole job. Same result.

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

void test (vector &k)
{
    cout << "Test result :          " << k.module() << endl;
}

void main()
{
   vector a (4, 5);
   trivector b (1, 2, 3);

   cout << "a (4, 5)    b (1, 2, 3)" << endl << endl;

   vector *r;

   r = &a;
   cout << "module of vector a    : " << r->module() << endl;

   r = &b;
   cout << "module of trivector b : " << r->module() << endl;
```

```
   test (a);

   test (b);

   vector &s = b;

   cout << "module of trivector b : " << s.module() << endl;
}
```

Maybe you wonder if a class can be derived from more than one base class. Answer is yes :

```
#include <iostream.h>
#include <math.h>

class vector
{
public:

   double x;
   double y;

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

   double surface()
   {
      return fabs (x * y);
   }
```

```
};

class number
{
public:

   double z;

   number (double a)
   {
      z = a;
   }

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

class trivector : public vector, public number
{
public:

   trivector(double a=0, double b=0, double c=0) : vector(a,b), number(c)
   {
   }           // The trivector constructor calls the vector
               // constructor, then the number constructor,
               // and in this example does nothing more.

   double volume()
   {
      return fabs (x * y * z);
   }
```

```
};

void main()
{
   trivector a(2, 3, -4);

   cout << a.volume() << endl;
   cout << a.surface() << endl;
   cout << a.is_negative() << endl;
}
```

Class derivation allows to construct "more complicated" classes build above other classes. There is another application of class derivation : allow the programmer to write generic functions.

Suppose you define a base class with no variables. It makes no sense to use instances of that class inside your program. But you write a function whose purpose is to sort instances of that class. Well, that function will be able to sort any types of objects provided they belong to a class derived from that base class ! The only condition is that inside every derived class definition all methods the sort function needs are correctly defined :

```
#include <iostream.h>
#include <math.h>

class octopus
{
public :

   virtual double module() = 0;  // = 0 implies function is not
                                 // defined. This makes instances
                                 // of this class cannot be declared.
};

double biggest_module (octopus &a, octopus &b, octopus &c)
{
```

```
    double r = a.module();
    if (b.module() > r) r = b.module();
    if (c.module() > r) r = c.module();
    return r;
}

class vector : public octopus
{
public:

   double x;
   double y;

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

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

class number : public octopus
{
public:

   double n;

   number (double a = 0)
   {
      n = a;
```

```
   }

   double module()
   {
      if (n >= 0) return n;
      else        return -n;
   }
};

void main ()
{
    vector k (1,2), m (6,7), n (100, 0);
    number p (5),   q (-3),  r (-150);

    cout << biggest_module (k, m, n) << endl;
    cout << biggest_module (p, q, r) << endl;

    cout << biggest_module (p, q, n) << endl;
}
```

Perhaps you think "okay, that's a good idea to derive classes from the class octopus **because that way I can apply to instances of my classes methods and function that were designed a generic way for the** octopus **class. But what if there exists another base class, named** cuttlefish**, which has very interesting methods and functions too ? Do I have to make my choice between** octopus **and** cuttlefish **when I want to derive a class ?" No, of course. A derived class can be at the same time derived from** octopus **and from** cuttlefish**. That's POLYMORPHISM. The derived class simply has to define the methods necessary for** octopus **together with the methods necessary for** cuttlefish **:**

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

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

class vector : public octopus, public cuttlefish
{
   double x;
   double y;

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

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

Probably you wonder what all those public: **keywords mean. They mean the variables or the methods below them may be accessed and used everywhere in the program.**

If you want them to be accessible only to methods of the class AND to methods of derived classes then you must put the keyword protected: **above them.**

If you want variables or methods be accessible ONLY to methods of the class then you must put the keyword private: **above them.**

The fact variables or methods are declared private or protected means no function external to the class may access or use them. That's ENCAPSULATION. If you want to give to a specific function the right to access those variables and methods then you must include that function's prototype inside the class definition, preceded by the keyword friend.

Now let's talk about input/output. In C++ that's a very broad subject.

Here is a program that writes to a file :

```
#include <iostream.h>
#include <fstream.h>

void main ()
{
   fstream f;

   f.open("c:\\test.txt", ios::out);

   f << "This is a text output to a file." << endl;

   double a = 345;

   f  << "A number : " << a << endl;

   f.close();
}
```

Here is a program that reads from a file :

```
#include <iostream.h>
#include <fstream.h>

void main ()
{
   fstream f;
   char c;

   cout << "What's inside the test.txt file from";
```

```
   cout << "the C: hard disk root " << endl;
   cout << endl;

   f.open("c:\\test.txt", ios::in);

   while (! f.eof() )
   {
      f.get(c);                          // Or c = f.get()
      cout << c;
   }

   f.close();
}
```

Roughly said, it is possible to do on character arrays the same operations as on files. This is very useful to convert data or manage memory arrays.

Here is a program that writes inside a character array :

```
#include <iostream.h>
#include <strstrea.h>
#include <string.h>
#include <math.h>

void main ()
{
   char a[1024];
   ostrstream b(a, 1024);

   b.seekp(0);                        // Start from first char.
   b << "2 + 2 = " << 2 + 2 << ends;  // ( ends, not endl )
                                      // ends is simply the
                                      // null character   '\0'
```

```
   cout << a << endl;

   double v = 2;

   strcpy (a, "A sinus : ");

   b.seekp(strlen (a));
   b << "sin (" << v << ") = " << sin(v) << ends;

   cout << a << endl;
}
```

A program that reads from a character string :

```
#include <iostream.h>
#include <strstrea.h>
#include <string.h>

void main ()
{
   char a[1024];
   istrstream b(a, 1024);

   strcpy (a, "45.656");

   double k, p;

   b.seekg(0);                        // Start from first character.
   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;
}
```

This program performs formated output two different ways. Please note the width() **and** setw() **MODIFIERS are only effective on the next item output to the stream. The second next item will not be influenced.**

```
#include <iostream.h>
#include <iomanip.h>

void main ()
{
   int i;

   cout << "A list of numbers :" << endl;
   for (i = 1; i <= 1024; i *= 2)
   {
      cout.width (7);
      cout << i << endl;
   }

   cout << "A table of numbers :" << endl;
   for (i = 0; i <= 4; i++)
   {
      cout << setw(3) << i << setw(5) << i * i * i << endl;
```

```
    }
}
```

You now have a basic knowledge about C++. Inside good books you will learn many more things. The file management system is very powerful. It has many other possibilities than those illustrated here. There is also a lot more to say about classes : template classes, virtual classes...

In order to work correctly with C++ you will need a good reference book, just like you needed one for C.

You will also need information on how C++ is used in your particular domain of activity. The standards, the global approach, the tricks, the typical problems encountered and their solutions...

Eric Brasseur - 23 february 1998 [Homepage]