C++ tutorial for C users

This book enunciates and illustrates features and basic principles of C++. It is aimed at experienced C users who wish to learn C++. It can also be interesting for beginner C++ users who leaved out some possibilities of the language. The original text is www.4p8.com/eric.brasseur/cppcen.html

I wish to thank 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 and Ralph Wu for their inspiration, advice, help, data, bug reports, references and bad practice busting.

Toni Ronkko:	hytti.uku.fi/~tronkko
Frédéric Cloth:	www.4p8.com/
Morten Brix Pedersen:	www.wtf.dk/hp
Ralph Wu:	blog.donews.com/ralix

Table of Contents

1. A new way to include libraries

There is a new way to **#include** libraries (the old method still works yet the compiler roars). The **.h** extension is no more written and the names of standard C libraries are written beginning with a **c**. In order for the program to use these libraries correctly **using namespace std;** has to be added:

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

int main ()
{
   double a;

   a = 1.2;
   a = sin (a);

   cout << a << endl;

   return 0;
}
```

Hints for beginners:

To compile this program, type it inside (or copy & paste it to) a text editor (gedit, kwrite, kate, kedit, vi, emacs, nano, pico, mcedit...), save it as a file named say **test01.cpp** (if you are a newbie, best put this file inside your home directory, that is say **/home/jones** on a Unix-like box).

To compile this source code file, type this command (on most open-source Unix-like boxes) in a console or terminal window:

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

To run the binary executable file **test01** that has been produced by the compilation (if there were no errors), type this:

```
./test01
```

Each time you modify the **test01.cpp** source code file, you need to compile it again if you want the modifications to echo in the **test01** executable file (type the upward arrow key on your keyboard to recall commands).

2. // for one-line remarks

You can use // to type a remark:

```
using namespace std;          // Using the standard library namespace.
#include <iostream>           // The iostream library is often used.

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

   a = 456.47;
   a = a + a * 21.5 / 100;    // A calculation.

   cout << a << endl;         // Display the content of a.

   return 0;                  // Program end.
}
```

The possibility to use // to type remarks has been added to C in C99 and ANSI C 2000.

3. Console input and output streams

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

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

void main()
{
   int a;                    // a is an integer variable
   char s [100];             // s points to a string of max 99 characters

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

   cout << endl;             // Just a 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;

   return 0;
}
```

4. Variable declarations can be put inside the code without using hooks

Variables can be declared everywhere inside the code without using hooks:

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

int 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;

   return 0;
}
```

Maybe try to use this feature to make your source codes more readable and not to mess them up :-).

Like in C, variables can be encapsulated between **{ }** hooks. Then they are local to the zone encapsulated between the **{** and **}**. Whatever happens with such variables inside the encapsulated zone will have no effect outside the zone:

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

int 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;

   return 0;
}
```

5. Variables can be initialized by a calculation involving other variables

A variable can be initialized by a calculation involving other variables:

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

int 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;

   return 0;
}
```

6. Variables can be declared inside a for loop declaration

C++ allows an iterator to be local to a **for** loop:

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

int main ()
{
   int i;                        // Simple declaration of i
   i = 487;

   for (int i = 0; i < 4; i++)  // Local declaration of i
   {
      cout << i << endl;         // This outputs 0, 1, 2 and 3
   }

   cout << i << endl;            // This outputs 487

   return 0;
}
```

In case the variable is not declared somewhere above the loop, you may be tempted to use it below the loop. Some early C++ compilers accept this. Then the variable has the value it had when the loop ended. You shouldn't do this. It's a bad practice:

```
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;
}
```

7. Global variables can be accessed even if a local variables has the same name

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

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

double a = 128;

int main ()
{
   double a = 256;

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

   return 0;
}
```

8. It is possible to declare a REFERENCE towards another variable

It is possible to make one variable be another:

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

int main ()
{
   double a = 3.1415927;

   double &b = a;                            // b is a

   b = 89;

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

   return 0;
}
```

If you are used at pointers and 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 after 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:

```
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;        // Displays 100, 4.

   return 0;
}
```

If you are used at pointers in C and wonder how exactly the program above works, here is how the C++ compiler would translate it to C:

```
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;        // Displays 100, 4.

   return 0;
}
```

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

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

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

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

   cout << "k: " << k << endl;       // Displays  3
   cout << "m: " << m << endl;       // Displays  7
   cout << endl;

   biggest (k, m) = 10;

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

   biggest (k, m) ++;

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

   return 0;
}
```

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:

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

double *biggest (double *r, double *r)
{
   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;

   (*(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;

   return 0;
}
```

To end with, for people who have to deal with pointers yet do not like it, references are useful to un-pointer variables. Beware this is considered a bad practice. You can go into trouble. See for example www.embedded.com/story/OEG20010311S0024

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

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

int 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 and r: " << b << endl;

   return 0;
}
```

9. Namespaces can be declared

Namespaces can be declared. The variables declared within a **namespace** can be used thanks to the **::** operator:

```
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;
}
```

10. A function can be declared inline

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 drawback is it will be bigger, because the full code of the function is inserted everywhere it is used:

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

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

int 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;

   return 0;
}
```

The possibility to use **inline** functions has been added to C in C99 and ANSI C 2000.

11. The exception structure has been added

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

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

int 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;

   return 0;
}
```

12. A function can have default parameters

It is possible to define default parameters for functions:

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

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

int main ()
{
   cout << test (14, 5) << endl;          // Displays 14 - 5
   cout << test (14)    << endl;          // Displays 14 - 7

   return 0;
}
```

13. PARAMETERS OVERLOAD: several functions can be declared with the same name provided there is a difference in their parameters list

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

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

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

int test (int a, int b)
{
   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;
}
```

14. The symbolic operators (+ - * / ...) can be defined for new data types

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

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

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;
}

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

   k.x =  2;                 // To be able to write
   k.y = -1;                 // k = vector (2, -1)
                             // see chapter 19

   m = 3.1415927 * k;        // Magic!

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

   return 0;
}
```

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

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

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

struct vector
{
   double x;
   double y;
};

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

int main ()
{
   vector a;

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

   cout << a << endl;            // Displays (35, 23)

   return 0;
}
```

15. Different functions for different data types will automatically be generated provided you define a template function

Tired of defining five times the same function? One definition for **int** type parameters, one definition for **double** type parameters, one definition for **float** type parameters... Didn't you forget one type? What if a new data type is used? No problem: the C++ compiler can generate automatically every version of the function that is necessary! Just tell him how the function looks like by declaring a **template** function:

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

template <class ttype>
ttype minimum (ttype a, ttype b)
{
   ttype r;

   r = a;
   if (b < a) r = b;

   return r;
}

int main ()
{
   int i1, i2, i3;
   i1 = 34;
   i2 = 6;
   i3 = minimum (i1, i2);
   cout << "Most little: " << i3 << endl;

   double d1, d2, d3;
   d1 = 7.9;
   d2 = 32.1;
   d3 = minimum (d1, d2);
   cout << "Most little: " << d3 << endl;

   cout << "Most little: " << minimum (d3, 3.5) << endl;

   return 0;
}
```

The function **minimum** is used three times in above program yet the C++ compiler generates only two versions of it: **int minimum (int a, int b)** and **double minimum (double a, double b)**. That does the job for the whole program.

Would you have tried something like calculating **minimum (i1, d1)** the compiler would have reported this as an error. Indeed the template tells both parameters are of the same type.

You can use a random number of different template data types in a template definition. And not all parameter types must be templates, some of them can be of standard types or user defined (**char**, **int**, **double**...). Here is an example where the **minimum** function takes parameters of any, possibly different, types and outputs a value that has the type of the first parameter:

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

template <class type1, class type2>
type1 minimum (type1 a, type2 b)
{
   type1 r, b_converted;
   r = a;
   b_converted = (type1) b;
   if (b_converted < a) r = b_converted;
   return r;
}

int main ()
{
   int i;
   double d;

   i = 45;
   d = 7.41;

   cout << "Most little: " << minimum (i, d)   << endl;
   cout << "Most little: " << minimum (d, i)   << endl;
   cout << "Most little: " << minimum ('A', i) << endl;

   return 0;
}
```

16. The keywords new and delete are much better to allocate and deallocate memory

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

new [] and **delete []** are used for arrays.

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

int 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. Note each 15
                                     // double will be constructed.
                                     // This is pointless here but it
                                     // is vital when using a data type
                                     // that needs its constructor be
                                     // used for each instance.
   d[0] = 4456;
   d[1] = d[0] + 567;

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

   delete [] d;                       // delete [] will deallocate the
                                     // memory zone. Note each 15
                                     // double will be destructed.
                                     // This is pointless here but it
                                     // is vital when using a data type
                                     // that needs its destructor be
                                     // used for each instance (the ~
                                     // method). Using delete without
                                     // the [] would deallocate the
                                     // memory zone without destructing
                                     // each of the 15 instances. That
                                     // would cause memory leakage.

   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;

   return 0;
}
```

17. To a class or struct you can add METHODS

In standard C a **struct** just contains data. In C++ a **struct** definition can also include functions. Those functions are own to the **struct** and are meant to operate on the data of the **struct**. Those functions are called METHODS. Example below defines the method **surface()** on the **struct vector**:

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

struct vector
{
   double x;
   double y;

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

int main ()
{
   vector a;

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

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

   return 0;
}
```

In the example above, **a** is an INSTANCE of **struct** "**vector**". (Note that the keyword "**struct**" was not necessary when declaring **vector a**.)

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.

What is a **class**? It's a **struct** yet that tends to keep its data hidden. Only the methods of the **class** can access the data. You can't access the data directly, unless authorized by the **public:** directive. Here is an example of a **class** definition. It behaves exactly the same way as the **struct** example above because the class data **x** and **y** are kept public:

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

class vector
{
public:

   double x;
   double y;

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

int main ()
{
   vector a;

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

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

   return 0;
}
```

In the example above, the **main()** function changes the data of instance **a** directly, using **a.x = 3** and **a.y = 4**. This is made possible by the **public:** directive in the class definition. This is a bad practice. See chapter 30.

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

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

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;
   }
};

int main ()
{
   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;

   return 0;
}
```

18. The CONSTRUCTOR and the DESTRUCTOR can be used to initialize and destroy an instance of a class

Very special and essential methods are the CONSTRUCTOR and DESTRUCTOR. They are automatically called whenever an instance of a class is created or destroyed (variable declaration, end of program, **new**, **delete**...).

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:

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

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;
   }

};

int 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;

   return 0;
}
```

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:

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

class vector
{
public:

   double x;
   double y;

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

int 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;

   return 0;
}
```

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:

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

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!

                                       // delete [] name would be more
                                       // academic but it is not vital
                                       // here since the array contains
                                       // no C++ sub-objects that need
                                       // to be deleted.

      cout << "Instance going to be deleted, 100 bytes freed" << endl;
   }
};

int 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;

   return 0;
}
```

Here is a short example of an array class definition. A method that is an overload of the **[]** operator and that outputs a reference (**&**) is used in order to generate an error if it is tried to access outside the limits of an 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)
   {
      if (i < 0 || i >= size)
      {
         cerr << endl << "Out of bounds" << endl;
         exit (EXIT_FAILURE);
      }
      else return data [i];
   }
};

int main ()
{
   array t (5);

   t[0] = 45;                      // OK
   t[4] = t[0] + 6;                // OK
   cout << t[4] << endl;           // OK

   t[10] = 7;                      // error!

   return 0;
}
```

19. Complex classes need the COPY CONSTRUCTOR and an overload of the = operator

If you cast an object like a vector, everything will happen all right. For example if vector **k** contains **(4, 7)**, after the cast **m = k** the vector **m** will contain **(4, 7)** too. The values of **k.x** and **k.y** have simply been copied to **m.x** and **m.y**. Now suppose you're playing with objects like the **person** class above. Those objects contain a pointer to a character string. If you cast such person object by writing **p = r** it is necesary that some function does the work to make **p** be a correct copy of **r**. Indeed otherwise **p.name** will point to the physical same character string as **r.name**. What's more the former character string pointed towards by **p.name** is lost and becomes a memory zombie. The result will be catastrophic: a mess of pointers and lost data. The methods that will do the job are the COPY CONSTRUCTOR and an overload of the = operator:

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

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 (const person &s)      // The COPY CONSTRUCTOR
   {
      name = new char[100];
      strcpy (name, s.name);
      age = s.age;
   }

   person& operator= (const person &s)  // overload of =
   {
      strcpy (name, s.name);
      age = s.age;
      return *this;
   }

   ~person ()
   {
      delete [] name;
   }
};

int 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;

   return 0;
}
```

The copy constructor allows your program to make copies of instances when doing calculations. It is a key method. During calculations, instances are created to hold intermediate results. They are modified, casted and destroyed without you being aware. This is why those methods can be useful even for simple objects (see chapter 14.).

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

20. The method bodies can be defined below the class definition (and Makefile usage example)

If a method cannot be **inline**, if you do not want it to be **inline**, if you want the class definition contain the minimum of information (or simply if you want the usual separated **.h** header file and **.cpp** source code file), then you must just put the prototype of the method inside the class and define the method below the class (or in a separated **.cpp** source file):

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

class vector
{
public:

   double x;
   double y;

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

double vector::surface()     // This is the method
{
   double s = 0;

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

   return s;
}

int main ()
{
   vector k;

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

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

   return 0;
}
```

For beginners:

If you intent to develop a serious C++ software, you need to separate the source code in **.h** header files and **.cpp** source files (just like for C). This is a short example of how it is done. The program above is split in three files:

A header file **vector.h**:

```
class vector
{
public:

   double x;
   double y;

   double surface();
};
```

A source code file **vector.cpp**:

```
using namespace std;
#include "vector.h"


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

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

   return s;
}
```

And a source code file **main.cpp**:

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

int main ()
{
   vector k;

   k.x = 4;
   k.y = 5;
   cout << "Surface: " << k.surface() << endl;

   return 0;
}
```

Assuming **vector.cpp** is perfect, you compile it once and for all into a **.o** "object file". The command below produces that object code file, that will bear the name **vector.o**:

```
g++ -c vector.cpp
```

Each time you modify the **main.cpp** source code file you compile it into say a **test20** executable file. You tell the compiler explicitely it has to link the **vector.o** object file into the final **test20** executable:

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

Run the executable this way:

```
./test20
```

This has several advantages:

- The source code of **vector.cpp** need to be compiled only once. This spares a lot of time on big softwares. (Linking the **vector.o** file into the **test20** executable is very fast.)

- You can give somebody the **.h** file and the **.o** file(s). That way he can use your software but not change it because he doesn't have the **.cpp** file(s) (don't rely too much on this, wait till you master these questions).

Note you can compile **main.cpp** too into an object file and then link it with **vector.o**:

```
g++ -c main.cpp

g++ main.o vector.o test20
```

If you want to look like a real C or C++ programmer you need to condense all this in a **Makefile** and compile using the **make** command. The file content beneath is an oversimplified version of such a **Makefile**. Copy it in a file named **Makefile**. You must not use spaces before the **g++** commands. Instead use the **Tab** character.

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

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

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

In order to compile, making use of that **Makefile**, type this command:

```
 make test20
```

The **make** command will parse through the file **Makefile** and infer what it has to do. To start with it will understand that **test20** depends on **main.o** and **vector.o**. So it will automatically launch "**make main.o**" and "**make vector.o**". Then it will check if **test20** allready exists and check for the date stamps of **test20**, **main.o** and **vector.o**. If **test20** allready exists and **main.o** and **vector.o** have a date stamp earlier than **test20**, the **make** command understands current version of **test20** is up to date so it has nothing to do. It will just report it did nothing. Otherwise, if **test20** does not exist, or **main.o** or **vector.o** are more recent than **test20**, the command that creates an up to date version of **test20** is executed, that is **g++ main.o vector.o -o test20**.

This next version of **Makefile** is closer to a standard **Makefile**:

```
all: test20

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

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

vector.o: vector.cpp vector.h
    g++ -c vector.cpp

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

You trigger the compilation by just typing the **make** command. The first line in the **Makefile** implies that if you just type **make** you intent "**make test20**":

```
make
```

This command erases all the files produced during compilation and all text editors backup files:

```
make clean
```

21. The keyword this is a pointer towards the instance a method is acting upon

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. The keyword **this** is intended therefore:

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

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;
   }
};

int 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;
   return 0;
}
```

22. Arrays of instances can be declared

Of course it is possible to declare arrays of objects:

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

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);
   }
};

int 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;

   return 0;
}
```

23. An example of complete class declaration

Here is an example of a full class declaration:

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

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 (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;
}

int 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;

   return 0;
}
```

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 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:

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

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;
}

int 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;

   return 0;
}
```

24. static variables inside a class definition

A class' variable can be declared **static**. Then only one instance of that variable exists, shared by all instances of the **class**. It must be initialized outside the **class** declaration:

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

class vector
{
public:

   double x;
   double y;
   static int count;

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

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

int vector::count = 0;

int main ()
{
   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;

   return 0;
}
```

25. const variables inside a class definition

A class variable can also be **const**ant. That's just like **static**, except it is alocated a value inside the class declaration and that value cannot be modified:

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

class vector
{
public:

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

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

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

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

   vector k (3, 4);

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

   return 0;
}
```

26. A class can be DERIVED from another class

A class can 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:

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

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
   }
};

int 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;

   return 0;
}
```

27. If a method is declared virtual the program will always first check the type of an instance that is pointed to and will use the appropriate method

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.

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

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;
}

int 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;

   return 0;
}
```

28. A class can be derived from more than one base classes

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

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

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);
   }
};

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

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

   return 0;
}
```

29. Class derivation allows to write generic methods

Class derivation allows to construct "more complicated" classes build above base 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. 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:

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

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;
   }
};

int 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;

   return 0;
}
```

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 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;
   }
}
```

30. ENCAPSULATION: public, protected and private

The **public:** directive means the variables or the methods below can be accessed and used everywhere in the program.

If you want the variables and methods 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 nothing external to the class can 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**.

The good practice is to encapsulate all the variables of a class. This can sound strange if you're common to structs in C. Indeed a struct only makes sense if you can access its data... In C++ you have to create methods to access the data inside a class. Example below uses the basic example of chapter 17 yet declares the class data is protected:

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

class vector
{
protected:

   double x;
   double y;

public:

   void set_x (int n)
   {
      x = n;
   }

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

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

int main ()
{
   vector a;

   a.set_x (3);
   a.set_y (4);

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

   return 0;
}
```

The example above is a bit odd since the class data **x** and **y** can be set yet they cannot be read back. Any attempt in function **main ()** to read **a.x** or **a.y** will result in a compilation error. In the next example **x** and **y** can be read back:

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

class vector
{
protected:

   double x;
   double y;

public:

   void set_x (int n)
   {
      x = n;
   }

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

   double get_x ()
   {
      return x;
   }

   double get_y ()
   {
      return y;
   }

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

int main ()
{
   vector a;

   a.set_x (3);
   a.set_y (4);

   cout << "The surface of a: " << a.surface() << endl;
   cout << "The width of a:   " << a.get_x() << endl;
   cout << "The height of a:  " << a.get_y() << endl;

   return 0;
}
```

In C++ one is not supposed to access the data of a class directly. Methods have to be declared. Why this? Many reasons exist. One is this allows to change the way the data is memorized inside the class. Another reason is this allows data inside the class to be "cross-dependent". Suppose **x** and **y** must always be of the same sign, otherwize ugly things can happen... If one is allowed to access the class data directly, it would be easy to impose say a positive **x** and a negative **y**. In the example below this is severely controlled:

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

int sign (double n)
{
   if (n >= 0) return 1;
   return -1;
}

class vector
{
protected:

   double x;
   double y;

public:

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

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

   double get_x ()
   {
      return x;
   }

   double get_y ()
   {
      return y;
   }

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

```
int main ()
{
   vector a;

   a.set_x (-3);
   a.set_y (4);

   cout << "The surface of a: " << a.surface() << endl;
   cout << "The width of a:   " << a.get_x() << endl;
   cout << "The height of a:  " << a.get_y() << endl;

   return 0;
}
```

31. Brief examples of file I/O

Let's talk about input/output. In C++ that's a broad subject.

This is a program that writes to a file:

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

int main ()
{
   fstream f;

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

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

   double a = 345;

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

   f.close();

   return 0;
}
```

This is a program that reads from a file:

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

int main ()
{
   fstream f;
   char c;

   cout << "What's inside the test.txt file:" << endl;
   cout << endl;

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

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

   f.close();

   return 0;
}
```

32. Character arrays can be used like files

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

This is a program that writes inside a character array:

```
using namespace std;
#include <iostream>
#include <strstream>
#include <cstring>
#include <cmath>

int 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;

   return 0;
}
```

A program that reads from a character string:

```
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);                       // 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;

   return 0;
}
```

33. An example of formated output

This program performs formated output two different ways. 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.

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

int 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;
   }

   return 0;
}
```

You now have a basic knowledge of C++. Inside good books you will learn many more things. The file management system is very powerful, it has much more 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 need 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... The best reference is of course the books written by Bjarn Stroustrup himself (I don't remind which one of them I read). Following book contains almost every detail about C and C++ and is constructed a way similar to this text:

Jamsa's C/C++ Programmer's Bible
© 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

Other references:
www.accu.org/bookreviews/public/reviews/0hr/index.htm
www.codersource.net/

If you are set back by the source code of important softwares, note lots of them are developed using user interface builder software like Glade or Qt Designer. Then use such a builder.