C++ (partially from http://www.cplusplus.com/)

C++ (partially from http://www.cplusplus.com/)....1
Structure of a C++ program....2
  Structure: 2
  comments: 2
Variables. Data types. Constants....3
  Data types:....3
  Constants: Literals:....4
  Defined constants (#define)....4
Operators....5
  Priority of operators....7
Communication through console....8
Control Structures....9
  if and else....9
  Repetitive structures or loops....9
  The do-while loop....10
  The for loop....10
  Bifurcation of control and jumps....11
  switch....11
Functions....13
  Functions with no types. The use of void....13
  Arguments passed by value and by reference....14
  Default values in arguments....14
  Overloaded functions....14
Arrays....16
  Arrays as parameters....16
Strings of Characters....17
  Converting strings to other types....17
  Functions to manipulate strings....18
Pointers....19
  Address (dereference) operator (&)....19
  Reference operator (*)....19
  Pointers and arrays....20
Dynamic memory....21
  Operators new and new[ ]....21
  Operator delete....21
Classes....22
  Constructors and destructors....23
  Overloading Constructors....24
  Pointers to classes....24
Overloading operators....26
  The keyword this....27
  Static members....27
Relationships between classes....29
  Inheritance between classes....30

Structure of a C++ program

Structure:

```
// my first program in C++

#include <iostream.h>

int main ()
{
  cout << "Hello World!";
  return 0;
}
```

comments:

```
// line comment
/* block comment */
```

Variables. Data types. Constants.

Data types:

Name	Bytes*	Description	Range*
char	1	character or integer 8 bits length.	signed: -128 to 127 unsigned: 0 to 255
short	2	integer 16 bits length.	signed: -32768 to 32767 unsigned: 0 to 65535
long	4	integer 32 bits length.	signed:-2147483648 to 2147483647 unsigned: 0 to 4294967295
int	*	Integer. Its length traditionally depends on the length of the system's Word type, thus in MSDOS it is 16 bits long, whereas in 32 bit systems (like Windows 9x/2000/NT and systems that work under protected mode in x86 systems) it is 32 bits long (4 bytes).	See short, long
float	4	floating point number.	3.4e + / - 38 (7 digits)
double	8	double precision floating point number.	1.7e + / - 308 (15 digits)
long double	10	long double precision floating point number.	1.2e + / - 4932 (19 digits)
bool	1	Boolean value. It can take one of two values: true or false NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it. Consult section `bool type` for compatibility information.	true or false
wchar_t	2	Wide character. It is designed as a type to store international characters of a two-byte character set. NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it.	wide characters

declaration:

type identifier [= initial_value] ;

Constants: Literals:

Character constants and string constants have certain peculiarities, like the escape codes. These are special characters that cannot be expressed otherwise in the sourcecode of a program, like newline (\n) or tab (\t). All of them are preceded by an inverted slash (\). Here you have a list of such escape codes:

\n	newline
\r	carriage return
\t	tabulation
\v	vertical tabulation
\b	backspace
\f	page feed
\a	alert (beep)
\'	single quotes (')
\"	double quotes (")
\?	question (?)
\\	inverted slash (\)

Defined constants (#define)

format:

#define identifier value

For example:

#define PI 3.14159265
#define NEWLINE '\n'
#define WIDTH 100

declared constants (const)

With the const prefix you can declare constants with a specific type exactly as you would do with a variable:
const int width = 100;
const char tab = '\t';
const zip = 12440;

Operators.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by the language are:
+ addition
- subtraction
* multiplication
/ division
% module

Compound assignation operators (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

value += increase; is equivalent to value = value + increase;
a -= 5; is equivalent to a = a - 5;
a /= b; is equivalent to a = a / b;
price *= units + 1; is equivalent to price = price * (units + 1);

Increase and decrease.

a++;
a+=1;
a=a+1;

Example 1	Example 2
B=3; A=++B; // A is 4, B is 4	B=3; A=B++; // A is 3, B is 4

Relational operators (==, !=, >, <, >=, <=)

==	Equal
!=	Different
>	Greater than
<	Less than
>=	Greater or equal than
<=	Less or equal than

Logic operators (!, &&, ||).

<table>
<tr><th>First Operand a</th><th>Second Operand b</th><th>result a && b</th><th>result a || b</th></tr>
<tr><td>true</td><td>true</td><td>true</td><td>true</td></tr>
<tr><td>true</td><td>false</td><td>false</td><td>true</td></tr>
<tr><td>false</td><td>true</td><td>false</td><td>true</td></tr>
<tr><td>false</td><td>false</td><td>false</td><td>false</td></tr>
</table>

Bitwise Operators (&, |, ^, ~, <<, >>).

<table>
<tr><th>op</th><th>asm</th><th>Description</th></tr>
<tr><td>&</td><td>AND</td><td>Logical AND</td></tr>
<tr><td>|</td><td>OR</td><td>Logical OR</td></tr>
<tr><td>^</td><td>XOR</td><td>Logical exclusive OR</td></tr>
<tr><td>~</td><td>NOT</td><td>Complement to one (bit inversion)</td></tr>
<tr><td><<</td><td>SHL</td><td>Shift Left</td></tr>
<tr><td>>></td><td>SHR</td><td>Shift Right</td></tr>
</table>

Explicit type casting operators

```
int i;
float f = 3.14;
i = (int) f; //i=3


i = int ( f );
```

Priority of operators

<table>
<tr><th>Priority</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>1</td><td>::</td><td>scope</td><td>Left</td></tr>
<tr><td>2</td><td>() [ ] -> . sizeof</td><td></td><td>Left</td></tr>
<tr><td rowspan="6">3</td><td>++ --</td><td>increment/decrement</td><td rowspan="6">Right</td></tr>
<tr><td>~</td><td>Complement to one (bitwise)</td></tr>
<tr><td>!</td><td>unary NOT</td></tr>
<tr><td>& *</td><td>Reference and Dereference (pointers)</td></tr>
<tr><td>(type)</td><td>Type casting</td></tr>
<tr><td>+ -</td><td>Unary less sign</td></tr>
<tr><td>4</td><td>* / %</td><td>arithmetical operations</td><td>Left</td></tr>
<tr><td>5</td><td>+ -</td><td>arithmetical operations</td><td>Left</td></tr>
<tr><td>6</td><td><< >></td><td>bit shifting (bitwise)</td><td>Left</td></tr>
<tr><td>7</td><td>< <= > >=</td><td>Relational operators</td><td>Left</td></tr>
<tr><td>8</td><td>== !=</td><td>Relational operators</td><td>Left</td></tr>
<tr><td>9</td><td>& ^ |</td><td>Bitwise operators</td><td>Left</td></tr>
<tr><td>10</td><td>&& ||</td><td>Logic operators</td><td>Left</td></tr>
<tr><td>11</td><td>?:</td><td>Conditional</td><td>Right</td></tr>
<tr><td>12</td><td>= += -= *= /= %=
>>= <<= &= ^= |=</td><td>Assignation</td><td>Right</td></tr>
<tr><td>13</td><td>,</td><td>Comma, Separator</td><td>Left</td></tr>
</table>

Communication through console.

cout

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120;              // prints number 120 on screen
cout << x;               // prints the content of variable x on screen

cout << "Hello, " << "I am " << "a C++ sentence";

cout << "First sentence.\n ";
```

cin

```
cin >> a >> b;
```

is equivalent to:

```
cin >> a;
cin >> b;
```

Control Structures.

if and else

if (condition) statement

For example:

```
if (x == 100)
  cout << "x is 100";
```

if (condition) statement1 else statement2

For example:

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

Repetitive structures or loops

while (expression) statement

```
// custom countdown using while
#include <iostream.h>
int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;
  while (n>0) {
    cout << n << ", ";
    --n;
  }
  cout << "FIRE!";
  return 0;
}
```

```
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The do-while loop.

Format:

do statement while (condition);

```
// number echoer
#include <iostream.h>
int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end):
";
    cin >> n;
    cout << "You entered: " << n << "\
n";
  } while (n != 0);
  return 0;
}
```

```
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277
Enter number (0 to end): 0
You entered: 0
```

The for loop.

Its format is:

for (initialization; condition; increase) statement;

It works the following way:

- initialization is executed. Generally it is an initial value setting for a counter varible. This is executed only once.
- condition is checked, if it is true the loop continues, otherwise the loop finishes and statement is skipped.
- statement is executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets { }.
- finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

```
// countdown using a for loop
#include <iostream.h>
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!";
  return 0;
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

Bifurcation of control and jumps.

The break instruction.

<pre>// break loop example
#include <iostream.h>
int main ()
{
  int n;
  for (n=10; n>0; n--) {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
  return 0;
}</pre>
	10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

The continue instruction.

<pre>// break loop example
#include <iostream.h>
int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "FIRE!";
  return 0;
}</pre>
	10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE

The exit function.

exit is a function defined in cstdlib (stdlib.h) library.
The purpose of exit is to terminate the running program with an specific exit code. Its prototype is:

void exit (int exit code);
The exit code is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means an error happened

switch.

the following:

switch (expression) {

```
case constant1:
  block of instructions 1
  break;
case constant2:
  block of instructions 2
  break;
.
.
.
default:
  default block of instructions
}
```

switch example	if-else equivalent

<pre>switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }</pre>
	<pre>if (x == 1) {
	  cout << "x is 1";
	  }
	else if (x == 2) {
	  cout << "x is 2";
	  }
	else {
	  cout << "value of x unknown";
	  }</pre>

Functions

type name (argument1, argument2, ...) statement

where:

- type is the type of data returned by the function.
- name is the name by which it will be possible to call the function.
- arguments (as many as wanted can be specified). Each argument consists of a type of data followed by its identifier, like in a variable declaration (for example, int x) and which acts within the function like any other variable. They allow passing parameters to the function when it is called. The different parameters are separated by commas.
- statement is the function's body. It can be a single instruction or a block of instructions. In the latter case it must be delimited by curly brackets {}.

Here you have the first function example:

Code	Output

// function example
#include <iostream.h>
 	
int addition (int a, int b)
{
 int r;
 r=a+b;
 return (r);
}
 	
int main ()
{
 int z;
 z = addition (5,3);
 cout << "The result is " << z;
 return 0;
}
	The result is 8

Functions with no types. The use of void.

Code	Output

// void function example
#include <iostream.h>
 	
void dummyfunction (void)
{
 cout << "I'm a function!";
}
 	
int main ()
{
 dummyfunction ();
 return 0;}
	I'm a function!

Arguments passed by value and by reference.

<pre>// passing parameters by reference
#include <iostream.h>
 	
void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}
 	
int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ",
z=" << z;
  return 0;
}</pre>
	x=2, y=6, z=14

Default values in arguments.

<pre>// default values in functions
#include <iostream.h>
 	
int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}
 	
int main ()
{
  cout << divide (12);
  cout << endl;
  cout << divide (20,4);
  return 0;
}</pre>
	6
	5

Overloaded functions.

<pre>// overloaded function
#include <iostream.h>
 	
int divide (int a, int b)
{
  return (a/b);
}</pre>
	2
	2.5

```
float divide (float a, float b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << divide (x,y);
  cout << "\n";
  cout << divide (n,m);
  cout << "\n";
  return 0;
}
```

Arrays

A typical declaration for an array in C++ is:

type name [elements];

where type is a valid object type (int, float...), name is a valid variable identifier and the elements field, that is enclosed within brackets [], specifies how many of these elements the array contains.

int billy [5] = { 16, 2, 77, 40, 12071 };

this declaration would have created an array like the following one:

	0	1	2	3	4
billy	16	2	77	40	12071

int jimmy [3][5];

Arrays as parameters

```
arrays as parameters
#include <iostream.h>

void printarray (int arg[], int length) {
  for (int n=0; n<length; n++)
    cout << arg[n] << " ";
  cout << "\n";
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
  return 0;
}
```

```
5 10 15
2 4 6 8 10
```

Strings of Characters.

initialize the string mystring with values by either of these two ways:

```
char mystring [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char mystring [] = "Hello";

strcpy (mystring, "Hello");
```

For example:

Code	Output

// setting value to string
#include <iostream.h>
#include <string.h>
 	
int main ()
{
char szMyName [20];
strcpy (szMyName,"J. Soulie");
cout << szMyName;
return 0;
}
	J. Soulie

Converting strings to other types

functions for this purpose:
atoi: converts string to int type.
atol: converts string to long type.
atof: converts string to float type.

```
// cin and ato* functions
#include <iostream.h>
#include <stdlib.h>

int main ()
{
  char mybuffer [100];
  float price;
  int quantity;
  cout << "Enter price: ";
  cin.getline (mybuffer,100);
  price = atof (mybuffer);
  cout << "Enter quantity: ";
  cin.getline (mybuffer,100);
  quantity = atoi (mybuffer);
  cout << "Total price: " << price*quantity;
  return 0;
}
```

```
Enter price: 2.75
Enter quantity: 21
Total price: 57.75
```

Functions to manipulate strings

The cstring library (string.h) defines many functions to perform manipulation operations with C-like strings (like already explained strcpy). Here you have a brief look at the most usual:

strcat: char* strcat (char* dest, const char* src);
Appends src string at the end of dest string. Returns dest.

strcmp: int strcmp (const char* string1, const char* string2);
Compares strings string1 and string2. Returns 0 is both strings are equal.

strcpy: char* strcpy (char* dest, const char* src);
Copies the content of src to dest. Returns dest.

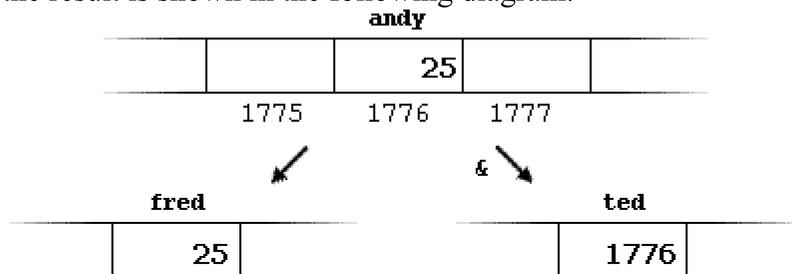
strlen: size_t strlen (const char* string);
Returns the length of string.

NOTE: char* is the same as char[]

Pointers

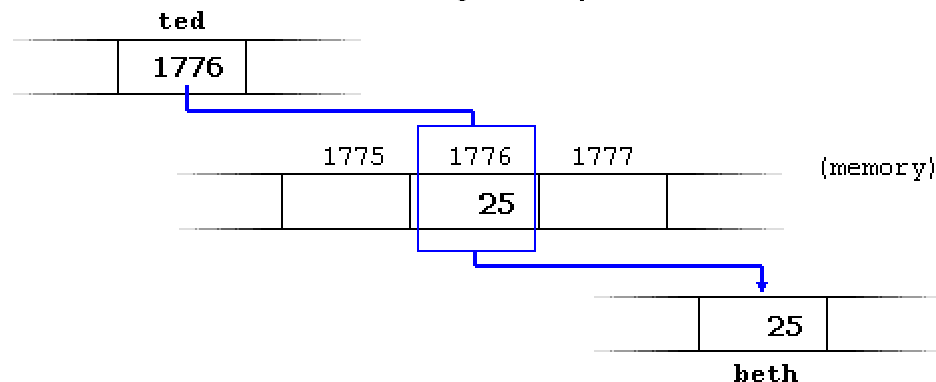
Address (dereference) operator (&).

andy = 25;
fred = andy;
ted = &andy;
the result is shown in the following diagram:



Reference operator (*)

beth = *ted;
(that we could read as: "beth equal to value pointed by ted") beth would take the value 25, since ted is 1776, and the value pointed by 1776 is 25.



beth = ted;   // beth equal to ted (1776)
beth = *ted; // beth equal to value pointed by ted (25)

```
// my first pointer
#include <iostream.h>

int main ()
{
  int value1 = 5, value2 = 15;
  int * mypointer;

  mypointer = &value1;
  *mypointer = 10;
  mypointer = &value2;
  *mypointer = 20;
  cout << "value1==" << value1 << "/
value2==" << value2;
  return 0;}
```

```
value1==10 / value2==20
```

```
// more pointers
#include <iostream.h>

int main ()
{
  int value1 = 5, value2 = 15;
  int *p1, *p2;

  p1 = &value1;     // p1 = address of value1
  p2 = &value2;     // p2 = address of value2
  *p1 = 10;         // value pointed by p1 = 10
  *p2 = *p1;        // value pointed by p2 = value pointed
by p1
  p1 = p2;          // p1 = p2 (value of pointer copied)
  *p1 = 20;         // value pointed by p1 = 20

  cout << "value1==" << value1 << "/ value2==" <<
value2;
  return 0;
}
```

```
value1==10 / value2==20
```

Pointers and arrays

int numbers [20];
int * p;
the following allocation would be valid:
p = numbers;

```
// more pointers
#include <iostream.h>

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

Dynamic memory.

Operators new and new[ ]

In order to request dynamic memory, the operator new exists. new is followed by a data type and optionally the number of elements required within brackets []. It returns a pointer to the beginning of the new block of assigned memory. Its form is:

pointer = new type
or
pointer = new type [elements]
The first expression is used to assign memory to contain one single element of type. The second one is used to assign a block (an array) of elements of type.

For example:
int * bobby;
bobby = new int [5];

Operator delete.

delete pointer;
or
delete [] pointer;

```
// rememb-o-matic
#include <iostream.h>
#include <stdlib.h>

int main ()
{
  char input [100];
  int i,n;
  long * l;
  cout << "How many numbers do you
want to type in? ";
  cin.getline (input,100); i=atoi (input);
  l= new long[i];
  if (l == NULL) exit (1);
  for (n=0; n<i; n++)
  {
    cout << "Enter number: ";
    cin.getline (input,100); l[n]=atol (input);
  }
  cout << "You have entered: ";
  for (n=0; n<i; n++)
    cout << l[n] << ", ";
  delete[] l;
  return 0;
}
```

```
How many numbers do you want to type
in? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Classes

Its form is:

```
class class_name {
  permission_label_1:
    member1;
  permission_label_2:
    member2;
  ...
  } object_name;
```

where class_name is a name for the class (user defined type) and the optional field object_name is one, or several, valid object identifiers. The body of the declaration can contain members, that can be either data or function declarations, and optionally permission labels, that can be any of these three keywords: private:, public: or protected:. They make reference to the permission which the following members acquire:

- private members of a class are accessible only from other members of their same class or from their "friend" classes.
- protected members are accessible from members of their same class and friend classes, and also from members of their derived classes.
- Finally, public members are accessible from anywhere the class is visible.

If we declare members of a class before including any permission label, the members are considered private, since it is the default permission that the members of a class declared with the class keyword acquire.

For example:

```
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void);
  } rect;
```

```
// class example
#include <iostream.h>

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}
```

```
rect area: 12
rectb area: 30
```

```
int main () {
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() <<
endl;
  cout << "rectb area: " << rectb.area() <<
endl;
}
```

Constructors and destructors

```
// example on constructors and destructors
#include <iostream.h>

class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area (void) {return (*width *
*height);}
};

CRectangle::CRectangle (int a, int b) {
  width = new int;
  height = new int;
  *width = a;
  *height = b;
}

CRectangle::~CRectangle () {
  delete width;
  delete height;
}

int main () {
  CRectangle rect (3,4), rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() <<
endl;
  return 0;
}
```

```
rect area: 12
rectb area: 30
```

Overloading Constructors

In fact, in the cases where we declare a class and we do not specify any constructor the compiler automatically assumes two overloaded constructors ("default constructor" and "copy constructor"). For example, for the class:

```
class CExample {
  public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
  };
```

with no constructors, the compiler automatically assumes that it has the following constructor member functions:

- Empty constructor

It is a constructor with no parameters defined as nop (empty block of instructions). It does nothing.

```
CExample::CExample () { };
```

- Copy constructor

It is a constructor with only one parameter of its same type that assigns to every nonstatic class member variable of the object a copy of the passed object.

```
CExample::CExample (const CExample& rv) {
  a=rv.a;  b=rv.b;  c=rv.c;
  }
```

Pointers to classes

`// pointer to classes example`
`#include <iostream.h>`
 	
`class CRectangle {`
`    int width, height;`
`  public:`
`    void set_values (int, int);`
`    int area (void) {return (width * height);}`
`};`
 	
`void CRectangle::set_values (int a, int b) {`
`  width = a;`
`  height = b;`
`}`
 	
`int main () {`
`  CRectangle a, *b, *c;`
`  CRectangle * d = new CRectangle[2];`
	a area: 2
	*b area: 12
	*c area: 2
	d[0] area: 30
	d[1] area: 56

```
  b= new CRectangle;
  c= &a;
  a.set_values (1,2);
  b->set_values (3,4);
  d->set_values (5,6);
  d[1].set_values (7,8);
  cout << "a area: " << a.area() << endl;
  cout << "*b area: " << b->area() << endl;
  cout << "*c area: " << c->area() << endl;
  cout << "d[0] area: " << d[0].area() <<
endl;
  cout << "d[1] area: " << d[1].area() <<
endl;
  return 0;
}
```

Next you have a summary on how can you read some pointer and class operators (*, &, ., ->, [ ]) that appear in the previous example:

*x can be read: pointed by x
&x can be read: address of x
x.y can be read: member y of object x
(*x).y can be read: member y of object pointed by x
x->y can be read: member y of object pointed by x (equivalent to the previous one)
x[0] can be read: first object pointed by x
x[1] can be read: second object pointed by x
x[n] can be read: (n+1)th object pointed by x

Overloading operators

Here is a list of all the operators that can be overloaded:

```
+    -    *    /    =    <    >    +=   -=   *=   /=   <<   >>
<<=  >>=  ==   !=   <=   >=   ++   --   %    &    ^    !    |
~    &=   ^=   |=   &&   ||   %=   []   ()   new  delete
```

<pre>// vectors: overloading operators example
#include <iostream.h>
 	
class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};
 	
CVector::CVector (int a, int b) {
  x = a;
  y = b;
}
 	
CVector CVector::operator+ (CVector param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}
 	
int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
}</pre>
	4,3

Although the prototype of a function operator+ can seem obvious since it takes the right side of the operator as the parameter for the function operator+ of the left side object, other operators are not so clear. Here you have a table with a summary on how the different operator functions must be declared (replace @ by the operator in each case):

<table>
<tr><th>Expression</th><th>Operator (@)</th><th>Function member</th><th>Global function</th></tr>
<tr><td>@a</td><td>+ - * & ! ~ ++ --</td><td>A::operator@()</td><td>operator@(A)</td></tr>
<tr><td>a@</td><td>++ --</td><td>A::operator@(int)</td><td>operator@(A, int)</td></tr>
<tr><td>a@b</td><td>+ - * / % ^ & | < > == != <= >= << >> && || ,</td><td>A::operator@(B)</td><td>operator@(A, B)</td></tr>
<tr><td>a@b</td><td>= += -= *= /= %= ^= &= |= <<= >>= [ ]</td><td>A::operator@(B)</td><td>-</td></tr>
<tr><td>a(b, c...)</td><td>()</td><td>A::operator()(B, C...)</td><td>-</td></tr>
<tr><td>a->b</td><td>-></td><td>A::operator->()</td><td>-</td></tr>
</table>

* where a is an object of class A, b is an object of class B and c is an object of class C.

The keyword this

The keyword **this** represents within a class the address in memory of the object of that class that is being executed. It is a pointer whose value is always the address of the object.
It is also frequenty used in **operator**= member functions that return objects by reference (avoiding the use of temporary objects). Following with the vector's examples seen before we could have written an operator= function like this:

```
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

In fact this is a probable default code generated for the class if we include no operator= member function.

Static members

A class can contain static members, either data or functions.
Static data members of a class are also known as "class variables", because their content does not depend on any object. There is only one unique value for all the objects of that same class.

```
// static members in classes
#include <iostream.h>

class CDummy {
  public:
```

```
7
6
```

```
    static int n;
    CDummy () { n++; };
    ~CDummy () { n--; };
};

int CDummy::n=0;

int main () {
  CDummy a;
  CDummy b[5];
  CDummy * c = new CDummy;
  cout << a.n << endl;
  delete c;
  cout << CDummy::n << endl;
  return 0;
}
```

Relationships between classes

In order to allow an external function to have access to the private and protected members of a class we have to declare the prototye of the external function that will gain access preceded by the keyword friend within the class declaration that shares its members. In the following example we declare the friend function duplicate:

```
// friend functions
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area (void) {return (width * height);}
    friend CRectangle duplicate
(CRectangle);
};

void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}

CRectangle duplicate (CRectangle
rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
}
```

```
24
```

Inheritance between classes

class derived_class_name: **public** base_class_name;

where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. public may be replaced by any of the other access specifiers protected or private, and describes the access for the inherited members, as we will see right after this example:

```
// derived classes
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

```
20
10
```

We can summarize the different access types according to whom can access them in the following way:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not-members	yes	no	no