Classes in C++

A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword class, with the following format:

```
class class_name {
        access_specifier_1:
                member1;
        access_specifier_2:
                member2;
        ...
} object_names;
```

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class.

The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers

Access specifiers

An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the the members following them acquire:

- private members of a class are accessible only from within other members of the same class or from their friends.
- protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members.

Therefore, any member that is declared before one other class specifier automatically has private access. For example:

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

declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: In the previous example, CRectangle was the class name (i.e., the type), whereas rect was an object of type CRectangle. It is the same relationship ``int'' and ``a'' have in the following declaration:

```
int a
```

where int is the type name (the class) and a is the variable name (the object).

After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. For example:

```
rect.set_values (3,4);
myarea = rect.area();
```

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.

```cpp
// Example1
#include <iostream>
using namespace std;
class CRectangle {
    int x, y;
public:
    void set_values (int,int);
    int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b) {
    x = a;
    y = b;
}
int main () {
    CRectangle rect;
    rect.set_values (3,4);
    cout << "area: " << rect.area() << endl;
    return 0;
}

// result - area: 12
```

Example 1

- The most important new thing in this code is the operator of scope (::, two colons) included in the definition of set_values(). It is used to define a member of a class from outside the class definition itself.

You may notice that the definition of the member function area() has been included directly within the definition of the CRectangle class given its extreme simplicity, whereas set_values() has only its prototype declared within the class, but its definition is outside it. In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function.

The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

- Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the class.

This makes sense, since we have already defined a member function to set values for those members within the object: the member function set_values(). Therefore, the rest of the program does not need to have direct access to them.

- One of the greater advantages of a class is that, as any other type, we can declare several objects of it.

- For example, following with the previous example of class CRectangle, we could have declared the object rectb in addition to the object rect – see Example 2.

```cpp
// Example 2: one class, two objects

#include <iostream>

using namespace std;
class CRectangle {
    int x, y;
public:
    void set_values (int,int);
    int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b) {
    x = a;
    y = b;
}
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;
    return 0;
}

// result - rect area: 12
//          rectb area: 30
```

In Example 2 the class is CRectangle with two objects: rect and rectb. Each one of them has its own member variables and member functions.

Notice that the call to rect.area() does not give the same result as the call to rectb.area().

This is because each object of class CRectangle has its own variables x and y, as they, in some way, have also their own function members set_value() and area() that each uses its object's own variables to operate.

Object-oriented programming

- That is the basic concept of object-oriented programming: Data and functions are both members of the object.
- We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions embedded as members.
- Notice that we have not had to give any parameters in any of the calls to rect.area or rectb.area. Those member functions directly used the data members of their respective objects rect and rectb.

Constructors and destructors

- Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution.

  For example, what would happen if in the previous example we called the member function area() before having called function set_values()?

  Probably we would have gotten an undetermined result since the members x and y would have never been assigned a value.

Constructors

- In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even void.
- We are going to implement CRectangle including a constructor: see Example 3

```cpp
// Example 3: class constructor

#include <iostream>

using namespace std;
class CRectangle {
    int width, height;
public:
    CRectangle (int,int);
    int area () {return (width*height);}
};
CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}
int main () {
    CRectangle rect (3,4);
    CRectangle rectb (5,6);
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}

//output -   rect area: 12
//           rectb area: 30
```

We have removed the member function set_values() and have included instead a constructor that performs a similar action: initializes the values of x and y with the parameters that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
CRectangle rect (3,4);

CRectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created. You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value; not even void.

Destructors

The destructor fulfills the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated. (See Example 4)

```
// Example 4: on constructors and destructors

#include <iostream>

using namespace std;
class CRectangle {
    int *width, *height;
public:
    CRectangle (int,int);
    ~CRectangle ();
    int area () {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;
}
// output -     rect area: 12
//              rectb area: 30
```

Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call.

In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration: see Example 5

```
//Example 5: overloading class constructors

#include <iostream>
using namespace std;

class CRectangle {
    int width, height;
public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return (width*height);}
};
CRectangle::CRectangle () {
    width = 5;
    height = 5;
}
CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}
int main () {
    CRectangle rect (3,4);
    CRectangle rectb;
    cout << "rect area: " << rect.area() << endl;
    cout << "rectb area: " << rectb.area() << endl;
    return 0;
}

//output -  rect area: 12
//       -  rectb area: 25
```

In this case, rectb was declared without any arguments, so it has been initialized with the constructor that has no parameters,

which initializes both width and height with a value of 5.

- Important: Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses ():

```
CRectangle rectb; // right
CRectangle rectb(); // wrong!
```

Default constructor

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

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

the compiler assumes that CExample has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```
CExample ex;
```

As soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

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

Here we have declared a constructor that takes two parameters of type int. Therefore the following object declaration would be correct:

```
CExample ex (2,3);
```

But

```
CExample ex;
```

would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor

Overloading operators

Thanks to the C++ feature to overload operators, we can design classes able to

perform operations using standard operators.

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

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

Class of 2-dim vectors

To overload an operator in order to use it with classes we declare operator functions, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload.

The format is:

type operator sign (parameters) { /*...*/ }

We are going to create a class to store two-dimensional vectors and then we are going to add two of them: a(3,1) and b(1,2) – see Example 6

```
//Example 6: vectors - overloading operators example
#include <iostream>
using namespace std;
class CVector { public:
    int x,y; CVector () {x=0; y=0;};
    CVector (int,int);
    CVector operator+ (CVector);
};
CVector::CVector (int a, int b)
{
    x = a;
    y = b;
}
CVector CVector::operator+ (CVector param)
{
    cout << x << endl;
    CVector temp; temp.x = x + param.x; temp.y = y + param.y;
    cout << temp.x << " = " << x << " + " << param.x << endl;
    return (temp);
}
int main ()
{    CVector k (3,1);
    CVector l (1,2);
    CVector m;
    // m = k.operator+ (l);
    m = k + l;
    cout << m.x << "," << m.y << endl;
    system ("pause");
    return 0;
}

// output -   4,3
```

It may be a little confusing to see so many times the CVector identifier.

But, consider that some of them refer to the class name (type) CVector and some others are functions with that name (constructors must have the same name as the class). Do not confuse them:

```
CVector (int, int); // function name CVector (constructor)
CVector operator+ (CVector); // function returns a Cvector
```

The function operator+ of class CVector is the one that is in charge of overloading the addition operator (+). This function can be called either implicitly using the operator, or explicitly using the function name:

```
c = a + b;
c = a.operator+ (b);
```

Both expressions are equivalent.

Notice also that we have included the empty constructor (without parameters) and we have defined it with a block:

```
CVector () { x=0; y=0; };
```

This is necessary, since we have explicitly declared another constructor:

```
CVector (int, int);
```

and when we explicitly declare any constructor, with any number of parameters, the default constructor with no parameters that the compiler can declare automatically is not declared, so we need to declare it ourselves in order to be able to construct objects of this type without parameters. Otherwise, the declaration:

```
CVector c;
```

included in main() would not have been valid.

As well as a class includes a default constructor and a copy constructor even if they are not declared, it also

includes a default definition for the assignment operator (=) with the class itself as parameter. The behavior which

is defined by default is to copy the whole content of the data members of the object passed as argument (the one at the right side of the sign) to the one at the left side:

```
CVector d (2,3);
CVector e;
e = d; // copy assignment operator
```

The copy assignment operator function is the only operator member function implemented by default. Of course, you can redefine it to any other functionality that you want, like for example, copy only certain class members or perform additional initialization procedures.

```
//Example 7: vectors - overloading operators example
#include <iostream>
using namespace std;
class CVector { public:
    int x,y; CVector () {x=0; y=0;};
    CVector (int,int);
    CVector operator* (double);
    };
CVector::CVector (int a, int b)
{
    x = a;
    y = b;
}
CVector CVector::operator* (double param)
{
    CVector temp; temp.x = x*param; temp.y = y* param;
    cout << temp.x << " = " << x << " * " << param << endl;
    return (temp);
}
int main ()
{   double n = 2;
    CVector k (3,1);
    CVector v;
    //v = k.operator*(n);
     v = k * n;
    cout << v.x << "," << v.y << endl;
    system ("pause");
    return 0;
}

// output -   6,2
```

Table with a summary on how the different operator functions have to be declared (replace @ by the operator in each case):

Expression	Operator	Member function	Global f-n
@ a	+ - * & ! ~ ++ --	A::operator @ ()	operator@(A)
a @	++ --	A::operator @ (int)	operator@(A,int)
a @ b	+ - * / % ^ & | < > == != <= >= << >> && ||	A::operator @ (B)	operator@(A,B)
a @ b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator @ (B)	
a(b,c,…)	()	A::operator () (B, C…)	
a -> x	->	A::operator -> ()	

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

You can see in this panel that there are two ways to overload some class operators: as a member function and as a global function.

Friends

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect friends.

- Friends are functions or classes declared as such.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend - see Example 8

```cpp
//Example 8:  friend functions
#include <iostream>
using namespace std;
class CRectangle {
    int width, height;
public:
    void set_values (int, int);
    int area () {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() << endl;
    return 0;
}
// output -   24
```

Friend functions

- The duplicate function is a friend of CRectangle. From within that function we have been able to access the members width and height of different objects of type CRectangle, which are private members. Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It isn't! It simply has access to its private and protected members without being a member.

- The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle.