**Classes**

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.

All is very similar to the declaration on data structures, except that we can now include also functions and

members, but also this new thing called *access specifier*. An access specifier is one of the following three

keywords: private, public or protected. These specifiers modify the access rights that 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.

* Finally, 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. All very similar to what we did with plain data

structures before.

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.

Here is the complete example of class CRectangle:

// classes example

#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();

return 0;

}

Output:

area: 12

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.

The only difference between defining a class member function completely within its class or to include only the

prototype and later its definition, is that in the first case the function will automatically be considered an inline

member function by the compiler, while in the second it will be a normal (not-inline) class member function, which

in fact supposes no difference in behavior.

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. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two

variables, but in greater projects it may be very important that values cannot be modified in an unexpected way

(unexpected from the point of view of the object).

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:

// example: 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;

}

Output-

rect area: 12

rectb area: 30

In this concrete case, the class (type of the objects) to which we are talking about is CRectangle, of which there

are two instances or 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.

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.