**Classes (I)**

*Classes* are an expanded concept of *data structures*: like data structures, they can contain data members, but they can also contain functions as members.  
  
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 defined using either keyword class or keyword struct, with the following syntax:

|  |
| --- |
| 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*, which can either be data or function declarations, and optionally *access specifiers*.  
  
Classes have the same format as plain *data structures*, except that they can also include functions and have these new things called *access specifiers*. An *access specifier* is one of the following three keywords: private, public or protected. These specifiers modify the access rights for the members that follow them:

* 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 other members of the same class (or 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 any other *access specifier* has private access automatically. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 | class Rectangle {  int width, height;  public:  void set\_values (int,int);  int area (void);  } rect; |  |

Declares a class (i.e., a type) called Rectangle and an object (i.e., a variable) of this class, called rect. This class contains four members: two data members of type int (member width and member height) with *private access* (because private is the default access level) and two member functions with *public access*: the functions set\_values and area, of which for now we have only included their declaration, but not their definition.  
  
Notice the difference between the *class name* and the *object name*: In the previous example, Rectangle was the *class name* (i.e., the type), whereas rect was an object of type Rectangle. 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 declarations of Rectangle and rect, any of the public members of object rect can be accessed as if they were normal functions or normal variables, by simply inserting a dot (.) between *object name* and *member name*. This follows the same syntax as accessing the members of plain data structures. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 | rect.set\_values (3,4);  myarea = rect.area(); |  |

The only members of rect that cannot be accessed from outside the class are width and height, since they have private access and they can only be referred to from within other members of that same class.  
  
Here is the complete example of class Rectangle:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // classes example  #include <iostream>  using namespace std;  class Rectangle {  int width, height;  public:  void set\_values (int,int);  int area() {return width\*height;}  };  void Rectangle::set\_values (int x, int y) {  width = x;  height = y;  }  int main () {  Rectangle rect;  rect.set\_values (3,4);  cout << "area: " << rect.area();  return 0;  } | area: 12 |  |

This example reintroduces the *scope operator* (::, two colons), seen in earlier chapters in relation to namespaces. Here it is used in the definition of function set\_values to define a member of a class outside the class itself.  
  
Notice that the definition of the member function area has been included directly within the definition of class Rectanglegiven its extreme simplicity. Conversely, set\_values it is merely declared with its prototype within the class, but its definition is outside it. In this outside definition, the operator of scope (::) is used to specify that the function being defined is a member of the class Rectangle and not a regular non-member function.  
  
The scope operator (::) specifies the class to which the member being defined belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, the function set\_values in the previous example has access to the variables width and height, which are private members of class Rectangle, and thus only accessible from other members of the class, such as this.  
  
The only difference between defining a member function completely within the class definition or to just include its declaration in the function and define it later outside the class, is that in the first case the function is automatically considered an *inline* member function by the compiler, while in the second it is a normal (not-inline) class member function. This causes no differences in behavior, but only on possible compiler optimizations.  
  
Members width and height have private access (remember that if nothing else is specified, all members of a class defined with keyword class have private access). By declaring them private, access from outside the class is not allowed. 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 how restricting access to these variables may be useful, 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).  
  
The most important property of a class is that it is a type, and as such, we can declare multiple objects of it. For example, following with the previous example of class Rectangle, we could have declared the object rectb in addition to object rect:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | // example: one class, two objects  #include <iostream>  using namespace std;  class Rectangle {  int width, height;  public:  void set\_values (int,int);  int area () {return width\*height;}  };  void Rectangle::set\_values (int x, int y) {  width = x;  height = y;  }  int main () {  Rectangle 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;  } | rect area: 12  rectb area: 30 |  |

In this particular case, the class (type of the objects) is Rectangle, of which there are two instances (i.e., objects): rectand 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 Rectangle has its own variables width and height, as they -in some way- have also their own function members set\_value and area that operate on the object's own member variables.  
  
Classes allow programming using object-oriented paradigms: Data and functions are both members of the object, reducing the need to pass and carry handlers or other state variables as arguments to functions, because they are part of the object whose member is called. Notice that no arguments were passed on the calls to rect.area or rectb.area. Those member functions directly used the data members of their respective objects rect and rectb.

**Constructors**

What would happen in the previous example if we called the member function area before having called set\_values? An undetermined result, since the members width and height had never been assigned a value.  
  
In order to avoid that, a class can include a special function called its *constructor*, which is automatically called whenever a new object of this class is created, allowing the class to initialize member variables or allocate storage.  
  
This constructor function is declared just like a regular member function, but with a name that matches the class name and without any return type; not even void.  
  
  
The Rectangle class above can easily be improved by implementing a constructor:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | // example: class constructor  #include <iostream>  using namespace std;  class Rectangle {  int width, height;  public:  Rectangle (int,int);  int area () {return (width\*height);}  };  Rectangle::Rectangle (int a, int b) {  width = a;  height = b;  }  int main () {  Rectangle rect (3,4);  Rectangle rectb (5,6);  cout << "rect area: " << rect.area() << endl;  cout << "rectb area: " << rectb.area() << endl;  return 0;  } | rect area: 12  rectb area: 30 |  |

The results of this example are identical to those of the previous example. But now, class Rectangle has no member function set\_values, and has instead a constructor that performs a similar action: it initializes the values of width and height with the arguments passed to it.  
  
Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

|  |  |  |
| --- | --- | --- |
| 1 2 | Rectangle rect (3,4);  Rectangle rectb (5,6); |  |

Constructors cannot be called explicitly as if they were regular member functions. They are only executed once, when a new object of that class is created.  
  
Notice how neither the constructor prototype declaration (within the class) nor the latter constructor definition, have return values; not even void: Constructors never return values, they simply initialize the object.

**Overloading constructors**

Like any other function, a constructor can also be overloaded with different versions taking different parameters: with a different number of parameters and/or parameters of different types. The compiler will automatically call the one whose parameters match the arguments:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 | // overloading class constructors  #include <iostream>  using namespace std;  class Rectangle {  int width, height;  public:  Rectangle ();  Rectangle (int,int);  int area (void) {return (width\*height);}  };  Rectangle::Rectangle () {  width = 5;  height = 5;  }  Rectangle::Rectangle (int a, int b) {  width = a;  height = b;  }  int main () {  Rectangle rect (3,4);  Rectangle rectb;  cout << "rect area: " << rect.area() << endl;  cout << "rectb area: " << rectb.area() << endl;  return 0;  } | rect area: 12  rectb area: 25 |  |

In the above example, two objects of class Rectangle are constructed: rect and rectb. rect is constructed with two arguments, like in the example before.  
  
But this example also introduces a special kind constructor: the *default constructor*. The *default constructor* is the constructor that takes no parameters, and it is special because it is called when an object is declared but is not initialized with any arguments. In the example above, the *default constructor* is called for rectb. Note how rectb is not even constructed with an empty set of parentheses - in fact, empty parentheses cannot be used to call the default constructor:

|  |  |  |
| --- | --- | --- |
| 1 2 | Rectangle rectb; // ok, default constructor called  Rectangle rectc(); // oops, default constructor NOT called |  |

This is because the empty set of parentheses would make of rectc a function declaration instead of an object declaration: It would be a function that takes no arguments and returns a value of type Rectangle.