**Chapter Eight**

**Fundamental of Classes**

**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: 

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | 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 ahave 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: 

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

|  |  |  |
| --- | --- | --- |
| 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>  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;  } | area: 12 |

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:

The most important new thing in this code is the operator of scope (::, two colons) included in the definition ofset\_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 definition, 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 any 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: 

|  |  |  |
| --- | --- | --- |
| 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>  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;  } | 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.

**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.  
  
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: 

|  |  |  |
| --- | --- | --- |
| 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 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;  } | rect area: 12  rectb area: 30 |

As you can see, the result of this example is identical to the previous one. But now we have removed the member function set\_values(), and have included instead a constructor that performs a similar action: it initializes the values of width and height 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:

|  |  |
| --- | --- |
| 1 2 | 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.  
  
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.

|  |  |  |
| --- | --- | --- |
| 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 30 | // example 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;  } | 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:

|  |  |  |
| --- | --- | --- |
| 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 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;  } | 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 ():

|  |  |
| --- | --- |
| 1 2 | 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:

|  |  |
| --- | --- |
| 1 2 3 4 5 | 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; |

But 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:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 | 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.  
  
But the compiler not only creates a default constructor for you if you do not specify your own. It provides three special member functions in total that are implicitly declared if you do not declare your own. These are the *copy constructor*, the *copy assignment operator*, and the default destructor.  
  
The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object. For CExample, the copy constructor implicitly declared by the compiler would be something similar to:

|  |  |
| --- | --- |
| 1 2 3 | CExample::CExample (const CExample& rv) {  a=rv.a; b=rv.b; c=rv.c;  } |

Therefore, the two following object declarations would be correct:

|  |  |
| --- | --- |
| 1 2 | CExample ex (2,3);  CExample ex2 (ex); // copy constructor (data copied from ex) |

**Pointers to classes**

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example: 

|  |  |
| --- | --- |
|  | CRectangle \* prect; |

|  |  |  |
| --- | --- | --- |
| 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 30 31 32 33 34 | // pointer to classes example  #include <iostream>  using namespace std;  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];  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;  delete[] d;  delete b;  return 0;  } | a area: 2  \*b area: 12  \*c area: 2  d[0] area: 30  d[1] area: 56 |

is a pointer to an object of class CRectangle.  
  
As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (->) of indirection. Here is an example with some possible combinations:

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

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

Be sure that you understand the logic under all of these expressions before proceeding with the next sections. If you have doubts, read again this section and/or consult the previous sections about pointers and data structures.

**The keyword this**

The keyword this represents a pointer to the object whose member function is being executed. It is a pointer to the object itself.  
  
One of its uses can be to check if a parameter passed to a member function is the object itself. For example,

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // this  #include <iostream>  class CDummy {  public:  int isitme (CDummy& param);  };  int CDummy::isitme (CDummy& param)  {  if (&param == this) return true;  else return false;  }  int main () {  CDummy a;  CDummy\* b = &a;  if ( b->isitme(a) )  cout << "yes, &a is b";  return 0;  } | yes, &a is b |

**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 there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another.   
  
For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // static members in classes  #include <iostream>  class CDummy {  public:  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;  } | 7  6 |

In fact, static members have the same properties as global variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, we can only include the prototype (its declaration) in the class declaration but not its definition (its initialization). In order to initialize a static data-member we must include a formal definition outside the class, in the global scope, as in the previous example:

|  |  |
| --- | --- |
|  | int CDummy::n=0; |

Because it is a unique variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

|  |  |
| --- | --- |
| 1 2 | cout << a.n;  cout << CDummy::n; |

These two calls included in the previous example are referring to the same variable: the static variable n within class CDummy shared by all objects of this class.  
  
Once again, I remind you that in fact it is a global variable. The only difference is its name and possible access restrictions outside its class.  
  
Just as we may include static data within a class, we can also include static functions. They represent the same: they are global functions that are called as if they were object members of a given class. They can only refer to static data, in no case to non-static members of the class, as well as they do not allow the use of the keywordthis, since it makes reference to an object pointer and these functions in fact are not members of any object but direct members of the class.

**Friendship and inheritance**

**Friend functions**

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 with the friend keyword.  
  
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:

|  |  |  |
| --- | --- | --- |
| 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 30 31 32 | // friend functions  #include <iostream>  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();  return 0;  } | 24 |

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 classCRectangle. 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.

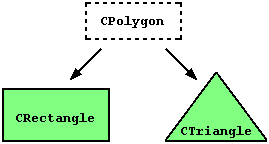
**Friend classes**

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

|  |  |  |
| --- | --- | --- |
| 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 30 31 32 33 34 35 36 | // friend class  #include <iostream>  using namespace std;  class CSquare;  class CRectangle {  int width, height;  public:  int area ()  {return (width \* height);}  void convert (CSquare a);  };  class CSquare {  private:  int side;  public:  void set\_side (int a)  {side=a;}  friend class CRectangle;  };  void CRectangle::convert (CSquare a) {  width = a.side;  height = a.side;  }    int main () {  CSquare sqr;  CRectangle rect;  sqr.set\_side(4);  rect.convert(sqr);  cout << rect.area();  return 0;  } | 16 |

In this example, we have declared CRectangle as a friend of CSquare so that CRectangle member functions could have access to the protected and private members of CSquare, more concretely to CSquare::side, which describes the side width of the square.  
  
You may also see something new at the beginning of the program: an empty declaration of class CSquare. This is necessary because within the declaration of CRectangle we refer to CSquare (as a parameter in convert()). The definition of CSquare is included later, so if we did not include a previous empty declaration for CSquare this class would not be visible from within the definition of CRectangle.  
  
Consider that friendships are not corresponded if we do not explicitly specify so. In our example, CRectangle is considered as a friend class by CSquare, but CRectangle does not consider CSquare to be a friend, so CRectanglecan access the protected and private members of CSquare but not the reverse way. Of course, we could have declared also CSquare as friend of CRectangle if we wanted to.  
  
Another property of friendships is that they are *not transitive*: The friend of a friend is not considered to be a friend unless explicitly specified.

**Inheritance between classes**

A key feature of C++ classes is inheritance. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own. For example, we are going to suppose that we want to declare a series of classes that describe polygons like our CRectangle, or likeCTriangle. They have certain common properties, such as both can be described by means of only two sides: height and base.  
  
This could be represented in the world of classes with a class CPolygon from which we would derive the two other ones: CRectangle and CTriangle.  
  
   
The class CPolygon would contain members that are common for both types of polygon. In our case: width andheight. And CRectangle and CTriangle would be its derived classes, with specific features that are different from one type of polygon to the other.  
  
Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive it to another class with another member called B, the derived class will contain both members A and B.  
  
In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:   
  
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. The public access specifier may be replaced by any one of the other access specifiers protected andprivate. This access specifier limits the most accessible level for the members inherited from the base class: The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

|  |  |  |
| --- | --- | --- |
| 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 30 31 32 33 | // derived classes  #include <iostream>  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 ()  { return (width \* height); }  };  class CTriangle: public CPolygon {  public:  int area ()  { 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 |

The objects of the classes CRectangle and CTriangle each contain members inherited from CPolygon. These are:width, height and set\_values().  
  
The protected access specifier is similar to private. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.  
  
Since we wanted width and height to be accessible from members of the derived classes CRectangle andCTriangle and not only by members of CPolygon, we have used protected access instead of private.  
  
We can summarize the different access types according to who 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 |

Where "not members" represent any access from outside the class, such as from main(), from another class or from a function.  
  
In our example, the members inherited by CRectangle and CTriangle have the same access permissions as they had in their base class CPolygon:

|  |  |
| --- | --- |
| 1 2 3 4 5 | CPolygon::width // protected access  CRectangle::width // protected access  CPolygon::set\_values() // public access  CRectangle::set\_values() // public access |

This is because we have used the public keyword to define the inheritance relationship on each of the derived classes:

|  |  |
| --- | --- |
|  | class CRectangle: public CPolygon { ... } |

This public keyword after the colon (:) denotes the most accessible level the members inherited from the class that follows it (in this case CPolygon) will have. Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.  
  
If we specify a more restrictive access level like protected, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: private, all the base class members are inherited as private.  
  
For example, if daughter was a class derived from mother that we defined as:

|  |  |
| --- | --- |
|  | class daughter: protected mother; |

This would set protected as the maximum access level for the members of daughter that it inherited from mother. That is, all members that were public in mother would become protected in daughter. Of course, this would not restrict daughter to declare its own public members. That maximum access level is only set for the members inherited from mother.  
  
If we do not explicitly specify any access level for the inheritance, the compiler assumes private for classes declared with class keyword and public for those declared with struct.

**What is inherited from the base class?**

In principle, a derived class inherits every member of a base class except:

* its constructor and its destructor
* its operator=() members
* its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.  
  
If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:  
  
derived\_constructor\_name (parameters) : base\_constructor\_name (parameters) {...}  
  
For example: 

|  |  |  |
| --- | --- | --- |
| 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 30 | // constructors and derived classes  #include <iostream>  using namespace std;  class mother {  public:  mother ()  { cout << "mother: no parameters\n"; }  mother (int a)  { cout << "mother: int parameter\n"; }  };  class daughter : public mother {  public:  daughter (int a)  { cout << "daughter: int parameter\n\n"; }  };  class son : public mother {  public:  son (int a) : mother (a)  { cout << "son: int parameter\n\n"; }  };  int main () {  daughter cynthia (0);  son daniel(0);    return 0;  } | mother: no parameters  daughter: int parameter    mother: int parameter  son: int parameter |

Notice the difference between which mother's constructor is called when a new daughter object is created and which when it is a son object. The difference is because the constructor declaration of daughter and son:

|  |  |
| --- | --- |
| 1 2 | daughter (int a) // nothing specified: call default  son (int a) : mother (a) // constructor specified: call this |

**Multiple inheritances**

In C++ it is perfectly possible that a class inherits members from more than one class. This is done by simply separating the different base classes with commas in the derived class declaration. For example, if we had a specific class to print on screen (COutput) and we wanted our classes CRectangle and CTriangle to also inherit its members in addition to those of CPolygon we could write: 

|  |  |
| --- | --- |
| 1 2 | class CRectangle: public CPolygon, public COutput;  class CTriangle: public CPolygon, public COutput; |

here is the complete example: 

|  |  |  |
| --- | --- | --- |
| 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 30 31 32 33 34 35 36 37 38 39 40 41 42 | // multiple inheritance  #include <iostream>  class CPolygon {  protected:  int width, height;  public:  void set\_values (int a, int b)  { width=a; height=b;}  };  class COutput {  public:  void output (int i);  };  void COutput::output (int i) {  cout << i << endl;  }  class CRectangle: public CPolygon, public COutput {  public:  int area ()  { return (width \* height); }  };  class CTriangle: public CPolygon, public COutput {  public:  int area ()  { return (width \* height / 2); }  };    int main () {  CRectangle rect;  CTriangle trgl;  rect.set\_values (4,5);  trgl.set\_values (4,5);  rect.output (rect.area());  trgl.output (trgl.area());  return 0;  } | 20  10 |