**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 declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, the functionset\_values in the previous example has access to the variables width and height, which are private members of classRectangle, 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 objectrect:

|  |  |  |
| --- | --- | --- |
| 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 membersset\_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 andheight 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.

**Uniform initialization**

The way of calling constructors by enclosing their arguments in parentheses, as shown above, is known as *functional form*. But constructors can also be called with other syntaxes:  
  
First, constructors with a single parameter can be called using the variable initialization syntax (an equal sign followed by the argument):  
  
class\_name object\_name = initialization\_value;   
  
More recently, C++ introduced the possibility of constructors to be called using *uniform initialization*, which essentially is the same as the functional form, but using braces ({}) instead of parentheses (()):  
  
class\_name object\_name { value, value, value, ... }   
  
Optionally, this last syntax can include an equal sign before the braces.  
  
Here is an example with four ways to construct objects of a class whose constructor takes a single parameter:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // classes and uniform initialization  #include <iostream>  using namespace std;  class Circle {  double radius;  public:  Circle(double r) { radius = r; }  double circum() {return 2\*radius\*3.14159265;}  };  int main () {  Circle foo (10.0); // functional form  Circle bar = 20.0; // assignment init.  Circle baz {30.0}; // uniform init.  Circle qux = {40.0}; // POD-like  cout << "foo's circumference: " << foo.circum() << '\n';  return 0;  } | foo's circumference: 62.8319 |

An advantage of uniform initialization over functional form is that, unlike parentheses, braces cannot be confused with function declarations, and thus can be used to explicitly call default constructors:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | Rectangle rectb; // default constructor called  Rectangle rectc(); // function declaration (default constructor NOT called)  Rectangle rectd{}; // default constructor called |  |

The choice of syntax to call constructors is largely a matter of style. Most existing code currently uses functional form, and some newer style guides suggest to choose uniform initialization over the others, even though it also has its potential pitfalls for its preference of [initializer\_list](http://www.cplusplus.com/initializer_list) as its type.

**Member initialization in constructors**

When a constructor is used to initialize other members, these other members can be initialized directly, without resorting to statements in its body. This is done by inserting, before the constructor's body, a colon (:) and a list of initializations for class members. For example, consider a class with the following declaration:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 | class Rectangle {  int width,height;  public:  Rectangle(int,int);  int area() {return width\*height;}  }; |  |

The constructor for this class could be defined, as usual, as:

|  |  |  |
| --- | --- | --- |
|  | Rectangle::Rectangle (int x, int y) { width=x; height=y; } |  |

But it could also be defined using *member initialization* as:

|  |  |  |
| --- | --- | --- |
|  | Rectangle::Rectangle (int x, int y) : width(x) { height=y; } |  |

Or even:

|  |  |  |
| --- | --- | --- |
|  | Rectangle::Rectangle (int x, int y) : width(x), height(y) { } |  |

Note how in this last case, the constructor does nothing else than initialize its members, hence it has an empty function body.  
  
For members of fundamental types, it makes no difference which of the ways above the constructor is defined, because they are not initialized by default, but for member objects (those whose type is a class), if they are not initialized after the colon, they are default-constructed.  
  
Default-constructing all members of a class may or may always not be convenient: in some cases, this is a waste (when the member is then reinitialized otherwise in the constructor), but in some other cases, default-construction is not even possible (when the class does not have a default constructor). In these cases, members shall be initialized in the member initialization list.

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 | // member initialization  #include <iostream>  using namespace std;  class Circle {  double radius;  public:  Circle(double r) : radius(r) { }  double area() {return radius\*radius\*3.14159265;}  };  class Cylinder {  Circle base;  double height;  public:  Cylinder(double r, double h) : base (r), height(h) {}  double volume() {return base.area() \* height;}  };  int main () {  Cylinder foo (10,20);  cout << "foo's volume: " << foo.volume() << '\n';  return 0;  } | foo's volume: 6283.19 |

In this example, class Cylinder has a member object whose type is another class (base's type is Circle). Because objects of class Circle can only be constructed with a parameter, Cylinder's constructor needs to call base's constructor, and the only way to do this is in the *member initializer list*.  
  
These initializations can also use uniform initializer syntax, using braces {} instead of parentheses ():

|  |  |  |
| --- | --- | --- |
|  | Cylinder::Cylinder (double r, double h) : base{r}, height{h} { } |  |

**Pointers to classes**

Objects can also be pointed to by pointers: Once declared, a class becomes a valid type, so it can be used as the type pointed to by a pointer. For example:

|  |  |  |
| --- | --- | --- |
|  | Rectangle \* prect; |  |

is a pointer to an object of class Rectangle.  
  
Similarly as with plain data structures, the members of an object can be accessed directly from a pointer by using the arrow operator (->). Here is an example with some possible combinations:

|  |  |
| --- | --- |
| 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 | // pointer to classes example  #include <iostream>  using namespace std;  class Rectangle {  int width, height;  public:  Rectangle(int x, int y) : width(x), height(y) {}  int area(void) { return width \* height; }  };  int main() {  Rectangle obj (3, 4);  Rectangle \* foo, \* bar, \* baz;  foo = &obj;  bar = new Rectangle (5, 6);  baz = new Rectangle[2] { {2,5}, {3,6} };  cout << "obj's area: " << obj.area() << '\n';  cout << "\*foo's area: " << foo->area() << '\n';  cout << "\*bar's area: " << bar->area() << '\n';  cout << "baz[0]'s area:" << baz[0].area() << '\n';  cout << "baz[1]'s area:" << baz[1].area() << '\n';  delete bar;  delete[] baz;  return 0;  } |

This example makes use of several operators to operate on objects and pointers (operators \*, &, ., ->, []). They can be interpreted as:

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

Most of these expressions have been introduced in earlier chapters. Most notably, the chapter about arrays introduced the offset operator ([]) and the chapter about plain data structures introduced the arrow operator (->).

**Classes defined with struct and union**

Classes can be defined not only with keyword class, but also with keywords struct and union.  
  
The keyword struct, generally used to declare plain data structures, can also be used to declare classes that have member functions, with the same syntax as with keyword class. The only difference between both is that members of classes declared with the keyword struct have public access by default, while members of classes declared with the keyword class have private access by default. For all other purposes both keywords are equivalent in this context.  
  
Conversely, the concept of *unions* is different from that of classes declared with struct and class, since unions only store one data member at a time, but nevertheless they are also classes and can thus also hold member functions. The default access in union classes is public.

**Classes (II)**

**Overloading operators**

Classes, essentially, define new types to be used in C++ code. And types in C++ not only interact with code by means of constructions and assignments. They also interact by means of operators. For example, take the following operation on fundamental types:

|  |  |  |
| --- | --- | --- |
| 1 2 | int a, b, c;  a = b + c; |  |

Here, different variables of a fundamental type (int) are applied the addition operator, and then the assignment operator. For a fundamental arithmetic type, the meaning of such operations is generally obvious and unambiguous, but it may not be so for certain class types. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | struct myclass {  string product;  float price;  } a, b, c;  a = b + c; |  |

Here, it is not obvious what the result of the addition operation on b and c does. In fact, this code alone would cause a compilation[ERROR[http://cdncache-a.akamaihd.net/items/it/img/arrow-10x10.png](http://www.cplusplus.com/doc/tutorial/templates/#82633523)](http://www.cplusplus.com/doc/tutorial/templates/#82633523), since the type myclass has no defined behavior for additions. However, C++ allows most operators to be overloaded so that their behavior can be defined for just about any type, including classes.

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

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

Operators are overloaded by means of operator functions, which are regular functions with special names: their name begins by the operator keyword followed by the *operator sign* that is overloaded. The syntax is:  
  
type operator sign (parameters) { /\*... body ...\*/ }   
For example, *cartesian vectors* are sets of two coordinates: x and y. The addition operation of two *cartesian vectors* is defined as the addition both x coordinates together, and both y coordinates together. For example, adding the *cartesian vectors* (3,1) and (1,2) together would result in (3+1,1+2) = (4,3). This could be implemented in C++ with the following code:

|  |  |  |
| --- | --- | --- |
| 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 | // overloading operators example  #include <iostream>  using namespace std;  class CVector {  public:  int x,y;  CVector () {};  CVector (int a,int b) : x(a), y(b) {}  CVector operator + (const CVector&);  };  CVector CVector::operator+ (const CVector& param) {  CVector temp;  temp.x = x + param.x;  temp.y = y + param.y;  return temp;  }  int main () {  CVector foo (3,1);  CVector bar (1,2);  CVector result;  result = foo + bar;  cout << result.x << ',' << result.y << '\n';  return 0;  } | 4,3 |

If confused about so many appearances of CVector, consider that some of them refer to the class name (i.e., the type)CVector and some others are functions with that name (i.e., constructors, which must have the same name as the class). For example:

|  |  |  |
| --- | --- | --- |
| 1 2 | CVector (int, int) : x(a), y(b) {} // function name CVector (constructor)  CVector operator+ (const CVector&); // function that returns a CVector |  |

The function operator+ of class CVector overloads the addition operator (+) for that type. Once declared, this function can be called either implicitly using the operator, or explicitly using its functional name:

|  |  |  |
| --- | --- | --- |
| 1 2 | c = a + b;  c = a.operator+ (b); |  |

Both expressions are equivalent.  
  
The operator overloads are just regular functions which can have any behavior; there is actually no requirement that the operation performed by that overload bears a relation to the mathematical or usual meaning of the operator, although it is strongly recommended. For example, a class that overloads operator+ to actually subtract or that overloads operator==to fill the object with zeros, is perfectly valid, although using such a class could be challenging.  
  
The parameter expected for a member function overload for operations such as operator+ is naturally the operand to the right hand side of the operator. This is common to all binary operators (those with an operand to its left and one operand to its right). But operators can come in diverse forms. Here you have a table with a summary of the parameters needed for each of the different operators than can be overloaded (please, replace @ by the operator in each case):

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Operator** | **Member function** | **Non-member function** |
| @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->b | -> | A::operator->() | - |
| (TYPE) a | TYPE | A::operator TYPE() | - |

Where a is an object of class A, b is an object of class B and c is an object of class C. TYPE is just any type (that operators overloads the conversion to type TYPE).  
  
Notice that some operators may be overloaded in two forms: either as a member function or as a non-member function: The first case has been used in the example above for operator+. But some operators can also be overloaded as non-member functions; In this case, the operator function takes an object of the proper class as first argument.

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 | // non-member operator overloads  #include <iostream>  using namespace std;  class CVector {  public:  int x,y;  CVector () {}  CVector (int a, int b) : x(a), y(b) {}  };  CVector operator+ (const CVector& lhs, const CVector& rhs) {  CVector temp;  temp.x = lhs.x + rhs.x;  temp.y = lhs.y + rhs.y;  return temp;  }  int main () {  CVector foo (3,1);  CVector bar (1,2);  CVector result;  result = foo + bar;  cout << result.x << ',' << result.y << '\n';  return 0;  } | 4,3 |

**The keyword this**

The keyword this represents a pointer to the object whose member function is being executed. It is used within a class's member function to refer 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 | // example on this  #include <iostream>  using namespace std;  class Dummy {  public:  bool isitme (Dummy& param);  };  bool Dummy::isitme (Dummy& param)  {  if (&param == this) return true;  else return false;  }  int main () {  Dummy a;  Dummy\* b = &a;  if ( b->isitme(a) )  cout << "yes, &a is b\n";  return 0;  } | yes, &a is b |

It is also frequently used in operator= member functions that return objects by reference. Following with the examples on *cartesian vector* seen before, its operator= function could have been defined as:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 | CVector& CVector::operator= (const CVector& param)  {  x=param.x;  y=param.y;  return \*this;  } |  |

In fact, this function is very similar to the code that the compiler generates implicitly for this class for operator=.

**Static members**

A class can contain static members, either data or functions.  
  
A static data member of a class is also known as a "class variable", because there is only one common variable for all the objects of that same class, sharing the same value: i.e., its value 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 | // static members in classes  #include <iostream>  using namespace std;  class Dummy {  public:  static int n;  Dummy () { n++; };  };  int Dummy::n=0;  int main () {  Dummy a;  Dummy b[5];  cout << a.n << '\n';  Dummy \* c = new Dummy;  cout << Dummy::n << '\n';  delete c;  return 0;  } | 6  7 |

In fact, static members have the same properties as non-member variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, they cannot be initialized directly in the class, but need to be initialized somewhere outside it. As in the previous example:

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

Because it is a common 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 << Dummy::n; |  |

These two calls above are referring to the same variable: the static variable n within class Dummy shared by all objects of this class.  
  
Again, it is just like a non-member variable, but with a name that requires to be accessed like a member of a class (or an object).  
  
Classes can also have static member functions. These represent the same: members of a class that are common to all object of that class, acting exactly as non-member functions but being accessed like members of the class. Because they are like non-member functions, they cannot access non-static members of the class (neither member variables nor member functions). They neither can use the keyword this.

**Const member functions**

When an object of a class is qualified as a const object:

|  |  |  |
| --- | --- | --- |
|  | const MyClass myobject; |  |

The access to its data members from outside the class is restricted to read-only, as if all its data members were const for those accessing them from outside the class. Note though, that the constructor is still called and is allowed to initialize and modify these data members:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | // constructor on const object  #include <iostream>  using namespace std;  class MyClass {  public:  int x;  MyClass(int val) : x(val) {}  int get() {return x;}  };  int main() {  const MyClass foo(10);  // foo.x = 20; // not valid: x cannot be modified  cout << foo.x << '\n'; // ok: data member x can be read  return 0;  } | 10 |

The member functions of a const object can only be called if they are themselves specified as const members; in the example above, member get (which is not specified as const) cannot be called from foo. To specify that a member is aconst member, the const keyword shall follow the function prototype, after the closing parenthesis for its parameters:

|  |  |  |
| --- | --- | --- |
|  | int get() const {return x;} |  |

Note that const can be used to qualify the type returned by a member function. This const is not the same as the one which specifies a member as const. Both are independent and are located at different places in the function prototype:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | int get() const {return x;} // const member function  const int& get() {return x;} // member function returning a const&  const int& get() const {return x;} // const member function returning a const& |  |

Member functions specified to be const cannot modify non-static data members nor call other non-const member functions. In essence, const members shall not modify the state of an object.  
  
const objects are limited to access only member functions marked as const, but non-const objects are not restricted and thus can access both const and non-const member functions alike.

You may think that anyway you are seldom going to declare const objects, and thus marking all members that don't modify the object as const is not worth the effort, but const objects are actually very common. Most functions taking classes as parameters actually take them by const reference, and thus, these functions can only access their const members:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // const objects  #include <iostream>  using namespace std;  class MyClass {  int x;  public:  MyClass(int val) : x(val) {}  const int& get() const {return x;}  };  void print (const MyClass& arg) {  cout << arg.get() << '\n';  }  int main() {  MyClass foo (10);  print(foo);  return 0;  } | 10 |

If in this example, get was not specified as a const member, the call to arg.get() in the print function would not be possible, because const objects only have access to const member functions.  
  
Member functions can be overloaded on their constness: i.e., a class may have two member functions with identical signatures except that one is const and the other is not: in this case, the const version is called only when the object is itself const, and the non-const version is called when the object is itself non-const.

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // overloading members on constness  #include <iostream>  using namespace std;  class MyClass {  int x;  public:  MyClass(int val) : x(val) {}  const int& get() const {return x;}  int& get() {return x;}  };  int main() {  MyClass foo (10);  const MyClass bar (20);  foo.get() = 15; // ok: get() returns int&  // bar.get() = 25; // not valid: get() returns const int&  cout << foo.get() << '\n';  cout << bar.get() << '\n';  return 0;  } | 15  20 |

**Class templates**

Just like we can create function templates, we can also create class templates, allowing classes to have members that use template parameters as types. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 | template <class T>  class mypair {  T values [2];  public:  mypair (T first, T second)  {  values[0]=first; values[1]=second;  }  }; |  |

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

|  |  |  |
| --- | --- | --- |
|  | mypair<int> myobject (115, 36); |  |

This same class could also be used to create an object to store any other type, such as:

|  |  |  |
| --- | --- | --- |
|  | mypair<double> myfloats (3.0, 2.18); |  |

The constructor is the only member function in the previous class template and it has been defined inline within the class definition itself. In case that a member function is defined outside the defintion of the class template, it shall be preceded with the template <...> prefix:

|  |  |  |
| --- | --- | --- |
| 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 | // class templates  #include <iostream>  using namespace std;  template <class T>  class mypair {  T a, b;  public:  mypair (T first, T second)  {a=first; b=second;}  T getmax ();  };  template <class T>  T mypair<T>::getmax ()  {  T retval;  retval = a>b? a : b;  return retval;  }  int main () {  mypair <int> myobject (100, 75);  cout << myobject.getmax();  return 0;  } | 100 |

Notice the syntax of the definition of member function getmax:

|  |  |  |
| --- | --- | --- |
| 1 2 | template <class T>  T mypair<T>::getmax () |  |

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

**Template specialization**

It is possible to define a different implementation for a template when a specific type is passed as template argument. This is called a *template specialization*.  
  
For example, let's suppose that we have a very simple class called mycontainer that can store one element of any type and that has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function memberuppercase, so we decide to declare a class template specialization for that type:

|  |  |  |
| --- | --- | --- |
| 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 | // template specialization  #include <iostream>  using namespace std;  // class template:  template <class T>  class mycontainer {  T element;  public:  mycontainer (T arg) {element=arg;}  T increase () {return ++element;}  };  // class template specialization:  template <>  class mycontainer <char> {  char element;  public:  mycontainer (char arg) {element=arg;}  char uppercase ()  {  if ((element>='a')&&(element<='z'))  element+='A'-'a';  return element;  }  };  int main () {  mycontainer<int> myint (7);  mycontainer<char> mychar ('j');  cout << myint.increase() << endl;  cout << mychar.uppercase() << endl;  return 0;  } | 8  J |

This is the syntax used for the class template specialization:

|  |  |  |
| --- | --- | --- |
|  | template <> class mycontainer <char> { ... }; |  |

First of all, notice that we precede the class name with template<> , including an empty parameter list. This is because all types are known and no template arguments are required for this specialization, but still, it is the specialization of a class template, and thus it requires to be noted as such.  
  
But more important than this prefix, is the <char> specialization parameter after the class template name. This specialization parameter itself identifies the type for which the template class is being specialized (char). Notice the differences between the generic class template and the specialization:

|  |  |  |
| --- | --- | --- |
| 1 2 | template <class T> class mycontainer { ... };  template <> class mycontainer <char> { ... }; |  |

The first line is the generic template, and the second one is the specialization.  
  
When we declare specializations for a template class, we must also define all its members, even those identical to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

**Special members**

[NOTE: This chapter requires proper understanding of *dynamically allocated memory*]  
  
*Special member functions* are member functions that are implicitly defined as member of classes under certain circumstances. There are six:

|  |  |
| --- | --- |
| **Member function** | **typical form for class C:** |
| Default constructor | C::C(); |
| Destructor | C::~C(); |
| Copy constructor | C::C (const C&); |
| Copy assignment | C& operator= (const C&); |
| Move constructor | C::C (C&&); |
| Move assignment | C& operator= (C&&); |

Let's examine each of these:

**Default constructor**

The *default constructor* is the constructor called when objects of a class are declared, but are not initialized with any arguments.  
  
If a class definition has no constructors, the compiler assumes the class to have an implicitly defined *default constructor*. Therefore, after declaring a class like this:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | class Example {  public:  int total;  void accumulate (int x) { total += x; }  }; |  |

The compiler assumes that Example has a *default constructor*. Therefore, objects of this class can be constructed by simply declaring them without any arguments:

|  |  |  |
| --- | --- | --- |
|  | Example ex; |  |

But as soon as a class has some constructor taking any number of parameters explicitly declared, the compiler no longer provides an implicit default constructor, and no longer allows the declaration of new objects of that class without arguments.

For example, the following class:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 | class Example2 {  public:  int total;  Example2 (int initial\_value) : total(initial\_value) { };  void accumulate (int x) { total += x; };  }; |  |

Here, we have declared a constructor with a parameter of type int. Therefore the following object declaration would be correct:

|  |  |  |
| --- | --- | --- |
|  | Example2 ex (100); // ok: calls constructor |  |

But the following:

|  |  |  |
| --- | --- | --- |
|  | Example2 ex; // not valid: no default constructor |  |

Would not be valid, since the class has been declared with an explicit constructor taking one argument and that replaces the implicit *default constructor* taking none.  
  
Therefore, if objects of this class need to be constructed without arguments, the proper *default constructor* shall also be declared in the class. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // classes and default constructors  #include <iostream>  #include <string>  using namespace std;  class Example3 {  string data;  public:  Example3 (const string& str) : data(str) {}  Example3() {}  const string& content() const {return data;}  };  int main () {  Example3 foo;  Example3 bar ("Example");  cout << "bar's content: " << bar.content() << '\n';  return 0;  } | bar's content: Example |

Here, Example3 has a *default constructor* (i.e., a constructor without parameters) defined as an empty block:

|  |  |  |
| --- | --- | --- |
|  | Example3() {} |  |

This allows objects of class Example3 to be constructed without arguments (like foo was declared in this example). Normally, a default constructor like this is implicitly defined for all classes that have no other constructors and thus no explicit definition is required. But in this case, Example3 has another constructor:

|  |  |  |
| --- | --- | --- |
|  | Example3 (const string& str); |  |

And when any constructor is explicitly declared in a class, no implicit *default constructors* is automatically provided.

**Destructor**

Destructors fulfill the opposite functionality of *constructors*: They are responsible for the necessary cleanup needed by a class when its lifetime ends. The classes we have defined in previous chapters did not allocate any resource and thus did not really require any CLEAN UP.  
  
But now, let's imagine that the class in the last example allocates dynamic memory to store the string it had as data member; in this case, it would be very useful to have a function called automatically at the end of the object's life in charge of releasing this memory. To do this, we use a *destructor*. A destructor is a member function very similar to a*default constructor*: it takes no arguments and returns nothing, not even void. It also uses the class name as its own name, but preceded with a tilde sign (~):

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | // destructors  #include <iostream>  #include <string>  using namespace std;  class Example4 {  string\* ptr;  public:  // constructors:  Example4() : ptr(new string) {}  Example4 (const string& str) : ptr(new string(str)) {}  // destructor:  ~Example4 () {delete ptr;}  // access content:  const string& content() const {return \*ptr;}  };  int main () {  Example4 foo;  Example4 bar ("Example");  cout << "bar's content: " << bar.content() << '\n';  return 0;  } | bar's content: Example |

On construction, Example4 allocates storage for a string. Storage that is later released by the destructor.  
  
The destructor for an object is called at the end of its lifetime; in the case of foo and bar THIS HAPPENS AT THE END OF FUNCTION main.

**Copy constructor**

When an object is passed a named object of its own type as argument, its *copy constructor* is invoked in order to construct a copy.  
  
A *copy constructor* IS A CONSTRUCTOR WHOSE FIRST PARAMETER IS OF TYPE *reference to the class* itself (possibly const qualified) and which can be invoked with a single argument of this type. For example, for a class MyClass, the *copy constructor* may have the following signature:

|  |  |  |
| --- | --- | --- |
|  | MyClass::MyClass (const MyClass&); |  |

If a class has no custom *copy* nor *move* constructors (or assignments) defined, an implicit *copy constructor* is provided. This copy constructor simply performs a copy of its own members. For example, for a class such as:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | class MyClass {  public:  int a, b; string c;  }; |  |

An implicit *copy constructor* is automatically defined. The definition assumed for this function performs a *shallow copy*, roughly equivalent to:

|  |  |  |
| --- | --- | --- |
|  | MyClass::MyClass(const MyClass& x) : a(x.a), b(x.b), c(x.c) {} |  |

This default *copy constructor* may suit the needs of many classes. But *shallow copies* only copy the members of the class themselves, and this is probably not what we expect for classes like class Example4 we defined above, because it contains pointers of which it handles its storage. For that class, performing a *shallow copy* means that the pointer value is copied, but not the content itself; This means that both objects (the copy and the original) would be sharing a single stringobject (they would both be pointing to the same object), and at some point (on destruction) both objects would try to delete the same block of memory, probably causing the program to crash on runtime. This can be solved by defining the following custom *copy constructor* that performs a *deep copy*:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | // copy constructor: deep copy  #include <iostream>  #include <string>  using namespace std;  class Example5 {  string\* ptr;  public:  Example5 (const string& str) : ptr(new string(str)) {}  ~Example5 () {delete ptr;}  // copy constructor:  Example5 (const Example5& x) : ptr(new string(x.content())) {}  // access content:  const string& content() const {return \*ptr;}  };  int main () {  Example5 foo ("Example");  Example5 bar = foo;  cout << "bar's content: " << bar.content() << '\n';  return 0;  } | bar's content: Example |

The *deep copy* performed by this *copy constructor* allocates storage for a new string, which is initialized to contain a copy of the original object. In this way, both objects (copy and original) have distinct copies of the content stored in different locations.

**Copy assignment**

Objects are not only copied on construction, when they are initialized: They can also be copied on any assignment operation. See the difference:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | MyClass foo;  MyClass bar (foo); // object initialization: copy constructor called  MyClass baz = foo; // object initialization: copy constructor called  foo = bar; // object already initialized: copy assignment called |  |

Note that baz is initialized on construction using an *equal sign*, but this is not an assignment operation! (although it may look like one): The declaration of an object is not an assignment operation, it is just another of the syntaxes to call single-argument constructors.  
  
THE ASSIGNMENT ON foo is an assignment operation. No object is being declared here, but an operation is being performed on an existing object; foo.  
  
The *copy assignment operator* is an overload of operator= which takes a *value* or *reference* of the class itself as parameter. The return value is generally a reference to \*this (although this is not required). For example, for a class MyClass, the *copy assignment* may have the following signature:

|  |  |  |
| --- | --- | --- |
|  | MyClass& operator= (const MyClass&); |  |

The *copy assignment operator* is also a *special function* and is also defined implicitly if a class has no custom *copy* nor*move* assignments (nor move constructor) defined.  
  
But again, the *implicit* version performs a *shallow copy* which is suitable for many classes, but not for classes with pointers to objects they handle its storage, as is the case in Example5. In this case, not only the class incurs the risk of deleting the pointed object twice, but the assignment creates MEMORY LEAKS by not deleting the object pointed by the object before the assignment. These issues could be solved with a *copy assignment* that deletes the previous object and performs a *deep copy*:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 | Example5& operator= (const Example5& x) {  delete ptr; // delete currently pointed string  ptr = new string (x.content()); // allocate space for new string, and copy  return \*this;  } |  |

Or even better, since its string member is not constant, it could re-utilize the same string object:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | Example5& operator= (const Example5& x) {  \*ptr = x.content();  return \*this;  } |  |

**Move constructor and assignment**

Similar to copying, moving also uses the value of an object to set the value to another object. But, unlike copying, the content is actually transferred from one object (the source) to the other (the destination): the source loses that content, which is taken over by the destination. This moving only happens when the source of the value is an *unnamed object*.   
  
*Unnamed objects* are objects that are temporary in nature, and thus haven't even been given a name. Typical examples of *unnamed objects* are return values of functions or type-casts.  
  
Using the value of a temporary object such as these to initialize another object or to assign its value, does not really require a copy: the object is never going to be used for anything else, and thus, its value can be *moved into* the destination object. These cases trigger the *move constructor* and *move assignments*:  
  
The *move constructor* is called when an object is initialized on construction using an unnamed temporary. Likewise, the*move assignment* is called when an object is assigned the value of an unnamed temporary:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 | MyClass fn(); // function returning a MyClass object  MyClass foo; // default constructor  MyClass bar = foo; // copy constructor  MyClass baz = fn(); // move constructor  foo = bar; // copy assignment  baz = MyClass(); // move assignment |  |

Both the value returned by fn and the value constructed with MyClass are unnamed temporaries. In these cases, there is no need to make a copy, because the unnamed object is very short-lived and can be acquired by the other object when this is a more efficient operation.  
  
The move constructor and move assignment are members that take a parameter of type *rvalue reference to the class*itself:

|  |  |  |
| --- | --- | --- |
| 1 2 | MyClass (MyClass&&); // move-constructor  MyClass& operator= (MyClass&&); // move-assignment |  |

An *rvalue reference* is specified by following the type with two ampersands (&&). As a parameter, an *rvalue reference* matches arguments of temporaries of this type.  
  
The concept of moving is most useful for objects that manage the storage they use, such as objects that allocate storage with new and delete. In such objects, copying and moving are really different operations:  
- Copying from A to B means that new memory is allocated to B and then the entire content of A is copied to this new memory allocated for B.  
- Moving from A to B means that the memory already allocated to A is transferred to B without allocating any new storage. It involves simply copying the pointer.

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 31 32 33 34 35 36 37 | // move constructor/assignment  #include <iostream>  #include <string>  using namespace std;  class Example6 {  string\* ptr;  public:  Example6 (const string& str) : ptr(new string(str)) {}  ~Example6 () {delete ptr;}  // move constructor  Example6 (Example6&& x) : ptr(x.ptr) {x.ptr=nullptr;}  // move assignment  Example6& operator= (Example6&& x) {  delete ptr;  ptr = x.ptr;  x.ptr=nullptr;  return \*this;  }  // access content:  const string& content() const {return \*ptr;}  // addition:  Example6 operator+(const Example6& rhs) {  return Example6(content()+rhs.content());  }  };  int main () {  Example6 foo ("Exam");  Example6 bar = Example6("ple"); // move-construction    foo = foo + bar; // move-assignment  cout << "foo's content: " << foo.content() << '\n';  return 0;  } | foo's content: Example |

Compilers already OPTIMIZE many cases that formally require a move-construction call in what is known as *Return Value OPTIMIZATION*. Most notably, when the value returned by a function is used to initialize an object. In these cases, the *move constructor* may actually never get called.  
  
NOTE THAT EVEN THOUGH *rvalue references* can be used for the type of any function parameter, it is seldom useful for uses other than the *move constructor*. Rvalue references are tricky, and unnecessary uses may be the source of ERRORS quite difficult to track.

**Implicit members**

The six *special members functions* described above are members implicitly declared on classes under certain circumstances:

|  |  |  |
| --- | --- | --- |
| **Member function** | **implicitly defined:** | **default definition:** |
| Default constructor | if no other constructors | does nothing |
| Destructor | if no destructor | does nothing |
| Copy constructor | if no move constructor and no move assignment | copies all members |
| Copy assignment | if no move constructor and no move assignment | copies all members |
| Move constructor | if no destructor, no copy constructor and no copy nor move assignment | moves all members |
| Move assignment | if no destructor, no copy constructor and no copy nor move assignment | moves all members |

Notice how not all *special member functions* are implicitly defined in the same cases. This is mostly due to backwards compatibility with C structures and earlier C++ versions, and in fact some include deprecated cases. Fortunately, each class can select explicitly which of these members exist with their default definition or which are deleted by using the keywords default and delete, respectively. The syntax is either one of:

|  |
| --- |
| function\_declaration = default;  function\_declaration = delete; |

For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | // default and delete implicit members  #include <iostream>  using namespace std;  class Rectangle {  int width, height;  public:  Rectangle (int x, int y) : width(x), height(y) {}  Rectangle() = default;  Rectangle (const Rectangle& other) = delete;  int area() {return width\*height;}  };  int main () {  Rectangle foo;  Rectangle bar (10,20);  cout << "bar's area: " << bar.area() << '\n';  return 0;  } | bar's area: 200 |

Here, Rectangle can be constructed either with two int arguments or be *default-constructed* (with no arguments). It cannot however be *copy-constructed* from another Rectangle object, because this function has been deleted. Therefore, assuming the objects of the last example, the following statement would not be valid:

|  |  |  |
| --- | --- | --- |
|  | Rectangle baz (foo); |  |

It could, however, be made explicitly valid by defining its copy constructor as:

|  |  |  |
| --- | --- | --- |
|  | Rectangle::Rectangle (const Rectangle& other) = default; |  |

Which would be essentially equivalent to:

|  |  |  |
| --- | --- | --- |
|  | Rectangle::Rectangle (const Rectangle& other) : width(other.width), height(other.height) {} |  |

Note that, the keyword default does not define a member function equal to the *default constructor* (i.e., where *default constructor* means constructor with no parameters), but equal to the constructor that would be implicitly defined if not deleted.  
  
In general, and for future compatibility, classes that explicitly define one copy/move constructor or one copy/move assignment but not both, are encouraged to specify either delete or default on the other special member functions they don't explicitly define.

**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 apply to *"friends"*.  
  
*Friends* are functions or classes declared with the friend keyword.  
  
A non-member function can access the private and PROTECTED members of a class if it is declared a *friend* of that class. That is done by including a declaration 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 | // friend functions  #include <iostream>  using namespace std;  class Rectangle {  int width, height;  public:  Rectangle() {}  Rectangle (int x, int y) : width(x), height(y) {}  int area() {return width \* height;}  friend Rectangle duplicate (const Rectangle&);  };  Rectangle duplicate (const Rectangle& param)  {  Rectangle res;  res.width = param.width\*2;  res.height = param.height\*2;  return res;  }  int main () {  Rectangle foo;  Rectangle bar (2,3);  foo = duplicate (bar);  cout << foo.area() << '\n';  return 0;  } | 24 |

The duplicate function is a *friend* of class Rectangle. Therefore, function duplicate is able to access the members widthand height (which are private) of different objects of type Rectangle. Notice though that neither in the declaration of duplicate nor in its later use in main, function duplicate is considered a member of class Rectangle. It isn't! It simply has access to its private and PROTECTED members without being a member.  
  
Typical use cases of friend functions are operations that are conducted between two different classes accessing private or protected members of both.

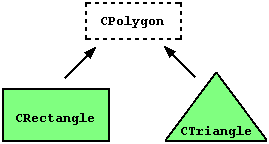
**Friend classes**

Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another 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 34 | // friend class  #include <iostream>  using namespace std;  class Square;  class Rectangle {  int width, height;  public:  int area ()  {return (width \* height);}  void convert (Square a);  };  class Square {  friend class Rectangle;  private:  int side;  public:  Square (int a) : side(a) {}  };  void Rectangle::convert (Square a) {  width = a.side;  height = a.side;  }    int main () {  Rectangle rect;  Square sqr (4);  rect.convert(sqr);  cout << rect.area();  return 0;  } | 16 |

In this example, class Rectangle is a friend of class Square allowing Rectangle's member functions to access private and protected members of Square. More concretely, Rectangle accesses the member variable Square::side, which describes the side of the square.  
  
There is something else new in this example: at the beginning of the program, there is an empty declaration of class Square. This is necessary because class Rectangle uses Square (as a parameter in member convert), and Square uses Rectangle (declaring it a friend).   
  
Friendships are never corresponded unless specified: In our example, Rectangle is considered a friend class by Square, but Square is not considered a friend by Rectangle. Therefore, the member functions of Rectangle can access the protected and private members of Square but not the other way around. Of course, Square could also be declared friend ofRectangle, if needed, granting such an access.  
  
Another property of friendships is that they are not transitive: The friend of a friend is not considered a friend unless explicitly specified.

**Inheritance between classes**

Classes in C++ can be extended, creating new classes which retain characteristics of the base class. This process, known as inheritance, involves a *base class* and a *derived class*: The *derived class* inherits the members of the *base class*, on top of which it can add its own members.  
  
For example, let's imagine a series of classes to describe two kinds of polygons: rectangles and triangles. These two polygons have certain common properties, such as the values needed to calculate their areas: they both can be described simply with a height and a width (or base).  
  
This could be represented in the world of classes with a class Polygon from which we would derive the two other ones:Rectangle and Triangle:  
  
   
The Polygon class would contain members that are common for both types of polygon. In our case: width and height. And Rectangle and Triangle 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 a class from it with another member called B, the derived class will contain both member A and member B.  
  
The inheritance relationship of two classes is declared in the derived class. Derived classes definitions use the following syntax:  
  
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 or private). 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>  using namespace std;  class Polygon {  protected:  int width, height;  public:  void set\_values (int a, int b)  { width=a; height=b;}  };  class Rectangle: public Polygon {  public:  int area ()  { return width \* height; }  };  class Triangle: public Polygon {  public:  int area ()  { return width \* height / 2; }  };    int main () {  Rectangle rect;  Triangle trgl;  rect.set\_values (4,5);  trgl.set\_values (4,5);  cout << rect.area() << '\n';  cout << trgl.area() << '\n';  return 0;  } | 20  10 |

The objects of the classes Rectangle and Triangle each contain members inherited from Polygon. These are: width,height and set\_values.  
  
The PROTECTED access specifier used in class Polygon is similar to private. Its only difference occurs in fact with inheritance: When a class inherits another one, the members of the derived class can access the PROTECTED members inherited from the base class, but not its private members.  
  
By declaring width and height as PROTECTED instead of private, these members are also accessible from the derived classes Rectangle and Triangle, instead of just from members of Polygon. If they were public, they could be accessed just from anywhere.

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

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

Where "not members" represents any access from outside the class, such as from main, from another class or from a function.  
  
In the example above, the members inherited by Rectangle and Triangle have the same access permissions as they had in their base class Polygon:

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

This is because the inheritance relation has been declared using the public keyword on each of the derived classes:

|  |  |  |
| --- | --- | --- |
|  | class Rectangle: public Polygon { /\* ... \*/ } |  |

This public keyword after the colon (:) denotes the most accessible level the members inherited from the class that follows it (in this case Polygon) will have from the derived class (in this case Rectangle). 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.  
  
With protected, all public members of the base class are inherited as protected in the derived class. Conversely, if the most restricting access level is specified (private), all the base class members are inherited as private.  
  
For example, if daughter were a class derived from mother that we defined as:

|  |  |  |
| --- | --- | --- |
|  | class Daughter: protected Mother; |  |

This would set protected as the less restrictive 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 restrictDaughter from declaring its own public members. That *less restrictive access level* is only set for the members inherited from Mother.  
  
If no access level is specified for the inheritance, the compiler assumes private for classes declared with keyword class and public for those declared with struct.  
  
Actually, most use cases of inheritance in C++ should use public inheritance. When other access levels are needed for base classes, they can usually be better represented as member variables instead.

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

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

* its constructors and its destructor
* its assignment operator members (operator=)
* its friends
* its private members

Even though access to the constructors and destructor of the base class is not inherited as such, they are automatically called by the constructors and destructor of the derived class.  
  
Unless otherwise specified, the constructors of a derived class calls the default constructor of its base classes (i.e., the constructor taking no arguments). Calling a different constructor of a base class is possible, using the same syntax used to initialize member variables in the initialization list:  
  
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 kelly(0);  Son bud(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 due to the different constructor declarations of Daughter and Son:

|  |  |  |
| --- | --- | --- |
| 1 2 | Daughter (int a) // nothing specified: call default constructor  Son (int a) : Mother (a) // constructor specified: call this specific constructor |  |

**Multiple inheritance**

A class may inherit from more than one class by simply specifying more base classes, separated by commas, in the list of a class's base classes (i.e., after the colon). For example, if the program had a specific class to print on screen calledOutput, and we wanted our classes Rectangle and Triangle to also inherit its members in addition to those of Polygon we could write:

|  |  |  |
| --- | --- | --- |
| 1 2 | class Rectangle: public Polygon, public Output;  class Triangle: public Polygon, public Output; |  |

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 | // multiple inheritance  #include <iostream>  using namespace std;  class Polygon {  protected:  int width, height;  public:  Polygon (int a, int b) : width(a), height(b) {}  };  class Output {  public:  static void print (int i);  };  void Output::print (int i) {  cout << i << '\n';  }  class Rectangle: public Polygon, public Output {  public:  Rectangle (int a, int b) : Polygon(a,b) {}  int area ()  { return width\*height; }  };  class Triangle: public Polygon, public Output {  public:  Triangle (int a, int b) : Polygon(a,b) {}  int area ()  { return width\*height/2; }  };    int main () {  Rectangle rect (4,5);  Triangle trgl (4,5);  rect.print (rect.area());  Triangle::print (trgl.area());  return 0;  } | 20  10 |

**Polymorphism**

Before getting any deeper into this chapter, you should have a proper understanding of pointers and class inheritance. If you are not really sure of the meaning of any of the following expressions, you should review the indicated sections:

|  |  |
| --- | --- |
| **Statement:** | **Explained in:** |
| int A::b(int c) { } | Classes |
| a->b | Data structures |
| class A: public B {}; | Friendship and inheritance |

**Pointers to base class**

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class. *Polymorphism* is the art of taking advantage of this simple but powerful and versatile feature.  
  
The example about the rectangle and triangle classes can be rewritten using pointers taking this feature into account:

|  |  |  |
| --- | --- | --- |
| 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 | // pointers to base class  #include <iostream>  using namespace std;  class Polygon {  protected:  int width, height;  public:  void set\_values (int a, int b){ width=a; height=b; }  };  class Rectangle: public Polygon {  public:  int area(){ return width\*height; }  };  class Triangle: public Polygon {  public:  int area(){ return width\*height/2; }  };  int main () {  Rectangle rect;  Triangle trgl;  Polygon \* ppoly1 = &rect;  Polygon \* ppoly2 = &trgl;  ppoly1->set\_values (4,5);  ppoly2->set\_values (4,5);  cout << rect.area() << '\n';  cout << trgl.area() << '\n';  return 0;  } | 20  10 |

Function main declares two pointers to Polygon (named ppoly1 and ppoly2). These are assigned the addresses of rect andtrgl, respectively, which are objects of type Rectangle and Triangle. Such assignments are valid, since both Rectangleand Triangle are classes derived from Polygon.  
  
Dereferencing ppoly1 and ppoly2 (with \*ppoly1 and \*ppoly2) is valid and allows us to access the members of their pointed objects. For example, the following two statements would be equivalent in the previous example:

|  |  |  |
| --- | --- | --- |
| 1 2 | ppoly1->set\_values (4,5);  rect.set\_values (4,5); |  |

But because the type of ppoly1 and ppoly2 is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and not those of the derived classes Rectangle and Triangle. That is why the program above accesses the area members of both objects using rect and trgl directly, instead of the pointers; the pointers to the base class cannot access the area members.  
  
Member area could have been accessed with the pointers to Polygon if area were a member of Polygon instead of a member of its derived classes, but the problem is that Rectangle and Triangle implement different versions of area, therefore there is not a single common version that could be implemented in the base class. 

**Virtual members**

A virtual member is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the virtual keyword:

|  |  |  |
| --- | --- | --- |
| 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 | // virtual members  #include <iostream>  using namespace std;  class Polygon {  protected:  int width, height;  public:  void set\_values (int a, int b)  { width=a; height=b; }  virtual int area ()  { return 0; }  };  class Rectangle: public Polygon {  public:  int area ()  { return width \* height; }  };  class Triangle: public Polygon {  public:  int area ()  { return (width \* height / 2); }  };  int main () {  Rectangle rect;  Triangle trgl;  Polygon poly;  Polygon \* ppoly1 = &rect;  Polygon \* ppoly2 = &trgl;  Polygon \* ppoly3 = &poly;  ppoly1->set\_values (4,5);  ppoly2->set\_values (4,5);  ppoly3->set\_values (4,5);  cout << ppoly1->area() << '\n';  cout << ppoly2->area() << '\n';  cout << ppoly3->area() << '\n';  return 0;  } | 20  10  0 |

In this example, all three classes (Polygon, Rectangle and Triangle) have the same members: width, height, and functions set\_values and area.  
  
The member function area has been declared as virtual in the base class because it is later redefined in each of the derived classes. Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of area in the example above, all three calls to area would return zero, because in all cases, the version of the base class would have been called instead.  
  
Therefore, essentially, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class that is pointing to an object of the derived class, as in the above example.  
  
A class that declares or inherits a virtual function is called a *polymorphic class*.  
  
Note that despite of the virtuality of one of its members, Polygon was a regular class, of which even an object was instantiated (poly), with its own definition of member area that always returns 0.

**Abstract base classes**

Abstract base classes are something very similar to the Polygon class in the previous example. They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (known as pure virtual functions). The syntax is to replace their definition by =0 (an equal sign and a zero):  
  
An abstract base Polygon class could look like this:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 | // abstract class CPolygon  class Polygon {  protected:  int width, height;  public:  void set\_values (int a, int b)  { width=a; height=b; }  virtual int area () =0;  }; |  |

Notice that area has no definition; this has been replaced by =0, which makes it a *pure virtual function*. Classes that contain at least one *pure virtual function* are known as *abstract base classes*.  
  
Abstract base classes cannot be used to instantiate objects. Therefore, this last abstract base class version of Polygoncould not be used to declare objects like:

|  |  |  |
| --- | --- | --- |
|  | Polygon mypolygon; // not working if Polygon is abstract base class |  |

But an *abstract base class* is not totally useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities. For example, the following pointer declarations would be valid:

|  |  |  |
| --- | --- | --- |
| 1 2 | Polygon \* ppoly1;  Polygon \* ppoly2; |  |

And can actually be dereferenced when pointing to objects of derived (non-abstract) classes. Here is the entire 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 | // abstract base class  #include <iostream>  using namespace std;  class Polygon {  protected:  int width, height;  public:  void set\_values (int a, int b)  { width=a; height=b; }  virtual int area (void) =0;  };  class Rectangle: public Polygon {  public:  int area (void)  { return (width \* height); }  };  class Triangle: public Polygon {  public:  int area (void)  { return (width \* height / 2); }  };  int main () {  Rectangle rect;  Triangle trgl;  Polygon \* ppoly1 = &rect;  Polygon \* ppoly2 = &trgl;  ppoly1->set\_values (4,5);  ppoly2->set\_values (4,5);  cout << ppoly1->area() << '\n';  cout << ppoly2->area() << '\n';  return 0;  } | 20  10 |

In this example, objects of different but related types are referred to using a unique type of pointer (Polygon\*) and the proper member function is called every time, just because they are virtual. This can be really useful in some circumstances. For example, it is even possible for a member of the abstract base class Polygon to use the special pointerthis to access the proper virtual members, even though Polygon itself has no implementation for this function:

|  |  |  |
| --- | --- | --- |
| 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 | // pure virtual members can be called  // from the abstract base class  #include <iostream>  using namespace std;  class Polygon {  protected:  int width, height;  public:  void set\_values (int a, int b)  { width=a; height=b; }  virtual int area() =0;  void printarea()  { cout << this->area() << '\n'; }  };  class Rectangle: public Polygon {  public:  int area (void)  { return (width \* height); }  };  class Triangle: public Polygon {  public:  int area (void)  { return (width \* height / 2); }  };  int main () {  Rectangle rect;  Triangle trgl;  Polygon \* ppoly1 = &rect;  Polygon \* ppoly2 = &trgl;  ppoly1->set\_values (4,5);  ppoly2->set\_values (4,5);  ppoly1->printarea();  ppoly2->printarea();  return 0;  } | 20  10 |

Virtual members and abstract classes grant C++ polymorphic characteristics, most useful for object-oriented projects. Of course, the examples above are very simple use cases, but these features can be applied to arrays of objects or dynamically allocated objects.  
  
Here is an example that combines some of the features in the latest chapters, such as dynamic memory, constructor initializers and polymorphism:

|  |  |  |
| --- | --- | --- |
| 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 | // dynamic allocation and polymorphism  #include <iostream>  using namespace std;  class Polygon {  protected:  int width, height;  public:  Polygon (int a, int b) : width(a), height(b) {}  virtual int area (void) =0;  void printarea()  { cout << this->area() << '\n'; }  };  class Rectangle: public Polygon {  public:  Rectangle(int a,int b) : Polygon(a,b) {}  int area()  { return width\*height; }  };  class Triangle: public Polygon {  public:  Triangle(int a,int b) : Polygon(a,b) {}  int area()  { return width\*height/2; }  };  int main () {  Polygon \* ppoly1 = new Rectangle (4,5);  Polygon \* ppoly2 = new Triangle (4,5);  ppoly1->printarea();  ppoly2->printarea();  delete ppoly1;  delete ppoly2;  return 0;  } | 20  10 |

Notice that the ppoly pointers:

|  |  |  |
| --- | --- | --- |
| 1 2 | Polygon \* ppoly1 = new Rectangle (4,5);  Polygon \* ppoly2 = new Triangle (4,5); |  |

are declared being of type "pointer to Polygon", but the objects allocated have been declared having the derived class type directly (Rectangle and Triangle).