C++
Information
Tutorials
Reference
Articles
Forum

Tutorials
C++ Language
Ascii Codes
Boolean Operations
Numerical Bases

C++ Language
Introduction:
Compilers
Basics of C++:
Structure of a program
Variables and types
Constants
Operators
Basic Input/Output
Program structure:
Statements and flow control
Functions
Overloads and templates
Name visibility
Compound data types:
Arrays
Character sequences
Pointers
Dynamic memory
Data structures
Other data types
Classes:
Classes (I)
Classes (II)
Special members
Friendship and inheritance
Polymorphism
Other language features:
Type conversions
Exceptions
Preprocessor directives
Standard library:
Input/output with files

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:

```
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:

```
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, 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:

```
// 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:

```
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:

```
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:

```
// 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:

```
// 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:

```
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:

```
// 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):

```
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:

```
// 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 a

const 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:

```
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:

```
// 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.

```
// 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:

```
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:

```
// 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`:

```
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 member `uppercase`, so we decide to declare a class template specialization for that type:

```
// 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 template <class T> class mycontainer { ... };
2 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.

Previous:
Classes (I)

Index

Next:
Special members