**Classes (II)**
- Overloading operators
- The keyword this
- Static members
- Const member functions
- Class templates
- Template specialization

# 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:

类本质上定义了要在c++代码中使用的新类型。c++中的类型不仅通过构造和赋值与代码交互。它们还通过操作符进行交互。例如，在基本类型上执行以下操作:

```
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:

在这里，基本类型(int)的不同变量被应用于加法操作符，然后是赋值操作符。对于基本算术类型，这种操作的含义通常是明显和明确的，但对于某些类类型可能不是这样。例如:

```
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:

在这里，对' b '和' c '的加法操作的结果并不明显。事实上，这段代码本身会导致编译错误，因为类型' myclass '没有为添加而定义的行为。然而，c++允许重载大多数操作符，以便它们的行为可以定义为几乎任何类型，包括类。下面是所有可以重载的操作符的列表:

**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:

操作符是通过操作符函数重载的，操作符函数是具有特殊名称的常规函数:其名称以operator关键字开头，后面跟着重载的操作符符号。的语法是:

`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:

例如，笛卡儿向量是两个坐标x和y的集合，两个笛卡儿向量的加法运算定义为两个x坐标和两个y坐标的加法。例如，将笛卡儿向量(3,1)和(1,2)相加将得到(3+1,1+2)=(4,3)。这可以通过以下代码在c++中实现

```
// 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;
}
```

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:

如果对CVector的这么多外观感到困惑，请考虑它们中的一些引用类名(即类型)CVector，而其他一些则是具有该名称的函数(即构造函数，它必须与类具有相同的名称)。例如:

```
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:

CVector类的函数操作符+重载该类型的加法操作符(+)。一旦声明，这个函数可以使用操作符隐式调用，也可以使用函数名显式调用:

```
c = a + b;
c = a.operator+ (b);
```

Both expressions are equivalent.

两个表达式是等价的。

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.

操作符重载只是普通函数，可以有任何行为;实际上，没有要求由该重载执行的操作与运算符的数学或通常含义有关，尽管强烈建议这样做。例如，重载' operator+ '以实际减去或重载' operator== '以将对象填充为0的类是完全有效的，尽管使用这样的类可能很有挑战性。

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):

operator+等操作的成员函数重载所需的形参自然是操作符右侧的操作数。这对所有的二元操作符都是常见的(在其左边有一个操作数，在其右边有一个操作数)。但运营商可以有多种形式。这里你有一个表，其中汇总了每个可以重载的不同操作符所需的参数(请用每个情况下的操作符替换' @ '):

| 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`).

其中a是类a的对象，b是类b的对象，c是类c的对象。TYPE是任意类型(操作符重载TYPE类型的转换)。

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.

注意，有些操作符可能以两种形式重载:作为成员函数或作为非成员函数:第一种情况已在上面的例子中用于操作符+。但有些操作符也可以作为非成员函数重载;在这种情况下，操作符函数接受一个适当类的对象作为第一个实参。

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;
}
```

## 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.

关键字this表示一个指向其成员函数正在执行的对象的指针。在类的成员函数中使用它来引用对象本身。

One of its uses can be to check if a parameter passed to a member function is the object itself. For example:

它的用途之一是检查传递给成员函数的参数是否为对象本身。例如:

```
// 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;
}
```

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:

它也经常用于通过引用返回对象的operator=成员函数中。下面是之前看到的关于笛卡尔向量的例子，它的operator=函数可以定义为:

```
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=`.

事实上，这个函数与编译器为`operator= '类隐式生成的代码非常相似。

## Static members

A class can contain static members, either data or functions.

类可以包含静态成员(数据或函数)。

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.

一个类的静态数据成员也被称为“类变量”，因为同一个类的所有对象只有一个公共变量，共享相同的值:也就是说，它的值在这个类的一个对象和另一个对象之间没有不同。

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:

例如，它可以用于类中的一个变量，该类可以包含一个计数器，该计数器包含当前已分配的类对象的数量，如下例所示:

```
// 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;
}
```

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:

事实上，静态成员具有与非成员变量相同的属性，但它们具有类作用域。因此，为了避免多次声明它们，不能在类中直接初始化它们，而是需要在类外部进行初始化。就像前面的例子:

```
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):

因为它是同一个类的所有对象的通用变量值，它可以被作为该类的任何对象的成员引用，甚至可以直接通过类名引用(当然，这只对静态成员有效):

```
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.

上面的两个调用都指向同一个变量:静态变量' n '在类' Dummy '中，由该类的所有对象共享。

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).

同样，它类似于一个非成员变量，但是它的名称需要像访问类(或对象)的成员一样访问。

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`.

类也可以有静态成员函数。它们表示的是相同的:类的成员对该类的所有对象都是通用的，它们的作用完全像非成员函数，但被访问时像类的成员一样。因为它们类似于非成员函数，它们不能访问类的非静态成员(既不是成员变量也不是成员函数)。他们也不能使用关键字“this”。

## Const member functions

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

当类的对象限定为const对象时:

```
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:

从类外部访问其数据成员的权限被限制为只读，就好像从类外部访问其数据成员的所有成员都是const一样。但是请注意，构造函数仍然被调用，并且允许初始化和修改这些数据成员:

```
// 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;
}
```

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:

const对象的成员函数只有在自身被指定为const成员时才能被调用;在上面的例子中，成员get(没有指定为const)不能从foo中调用。要指定一个成员为const成员，const关键字必须紧跟在函数原型的形参的右括号之后:

```
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:

注意，const可用于限定成员函数返回的类型。该const与将成员指定为const的const不同。两者都是独立的，位于函数原型的不同位置:

```
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.

指定为const的成员函数不能修改非静态数据成员，也不能调用其他非const成员函数。本质上，const成员不能修改对象的状态。

`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.

Const对象只能访问标记为Const的成员函数，而非Const对象则不受限制，因此可以同时访问Const和非Const成员函数。

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:

您可能会认为，无论如何，您很少会声明const对象，因此将所有不修改对象的成员标记为const是不值得的，但const对象实际上非常常见。大多数接受类作为形参的函数实际上是通过const引用接受类的，因此，这些函数只能访问它们的const成员:

```
// 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;
}
```

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.

如果在本例中，没有将get指定为const成员，则不能在print函数中调用arg.get()，因为const对象只能访问const成员函数。

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`.

成员函数可以重载的constness:即,一个类可以有两个成员函数具有相同签名的,除了一个是常量,另一个是:在这种情况下,const版本叫做只有当对象本身是常量,non-const版本叫做当non-const对象本身。

```
// 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;
}
```

## 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:

就像我们可以创建函数模板一样，我们也可以创建类模板，允许类拥有使用模板形参作为类型的成员。例如:

```
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:

我们刚刚定义的类用于存储任意有效类型的两个元素。例如，如果要声明该类的一个对象来存储两个int类型的整型值，分别为115和36，可以这样写:

```
mypair<int> myobject (115, 36);
```

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

这个类也可以用来创建存储任何其他类型的对象，例如:

```
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:

构造函数是前一个类模板中唯一的成员函数，它是在类定义本身内内联定义的。如果成员函数是在类模板的定义之外定义的，则它的前面应该加上template <...>前缀:

```
// 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;
}
```

Notice the syntax of the definition of member function `getmax`:

注意成员函数' getmax '的定义语法:

```
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.

被这么多T弄糊涂了?在这个声明中有三个T:第一个是模板参数。第二个T指向函数返回的类型。第三个T(尖括号之间的那个)也是必须的:它指定这个函数的模板形参也是类模板形参。

## 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*.

当将特定类型作为模板参数传递时，可以为模板定义不同的实现。这被称为模板专门化。

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:

例如，假设我们有一个非常简单的类mycontainer，它可以存储任何类型的元素，并且只有一个名为increase的成员函数，它会增加元素的值。但我们发现，当存储char类型的元素时，使用完全不同的大写函数成员实现会更方便，因此决定声明该类型的类模板专门化:

```
// 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;
}
```

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.

首先，请注意在类名前面加上template<>，其中包括一个空的形参列表。这是因为所有类型都是已知的，这种专门化不需要模板参数，但它仍然是类模板的专门化，因此需要这样标出。

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:

但是比这个前缀更重要的是类模板名后面的特化参数。这个特化参数本身标识模板类要特化的类型(char)。注意泛型类模板和特化之间的区别:

```
template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

第一行是通用模板，第二行是专门化。

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.

当为模板类声明专门化时，还必须定义它的所有成员，即使是那些与泛型模板类相同的成员，因为泛型模板的成员没有“继承”到专门化。