Overloads and templates

Overloaded functions

In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. For example:

在c++中，如果两个不同的函数的形参不同，它们可以有相同的名称;要么是因为它们有不同数量的参数，要么是因为它们的任何参数都是不同类型的。例如:

```
// overloading functions
#include <iostream>
using namespace std;

int operate (int a, int b)
{
  return (a*b);
}

double operate (double a, double b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  double n=5.0,m=2.0;
  cout << operate (x,y) << '\n';
  cout << operate (n,m) << '\n';
  return 0;
}
```

In this example, there are two functions called `operate`, but one of them has two parameters of type `int`, while the other has them of type `double`. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two `int` arguments, it calls to the function that has two `int` parameters, and if it is called with two `double`s, it calls the one with two `double`s.

在这个例子中，有两个叫做operate的函数，但是其中一个有两个int类型的形参，而另一个有double类型的形参。编译器通过检查函数调用时作为参数传递的类型，知道在每种情况下调用哪个类型。如果使用两个int形参调用它，则调用具有两个int形参的函数;如果使用两个double形参调用它，则调用具有两个double形参的函数。

In this example, both functions have quite different behaviors, the `int` version multiplies its arguments, while the `double` version divides them. This is generally not a good idea. Two functions with the same name are generally expected to have -at least- a similar behavior, but this example demonstrates that is entirely possible for them not to. Two overloaded functions (i.e., two functions with the same name) have entirely different definitions; they are, for all purposes, different functions, that only happen to have the same name.

在这个例子中，两个函数有完全不同的行为，int版本将其实参相乘，而double版本将实参除。这通常不是一个好主意。两个具有相同名称的函数通常会有—至少—相似的行为，但是这个例子证明了它们完全有可能没有。两个重载函数(即两个同名函数)具有完全不同的定义;无论如何，它们都是不同的函数，只是碰巧有相同的名称。

Note that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

注意，函数不能仅由其返回类型重载。它的参数中至少有一个必须具有不同的类型。

Function templates

Overloaded functions may have the same definition. For example:

重载函数可以具有相同的定义。例如:

```
// overloaded functions
#include <iostream>
using namespace std;

int sum (int a, int b)
{
  return a+b;
}

double sum (double a, double b)
{
  return a+b;
}

int main ()
{
  cout << sum (10,20) << '\n';
  cout << sum (1.0,1.5) << '\n';
  return 0;
}
```

Here, `sum` is overloaded with different parameter types, but with the exact same body.

这里，使用不同的形参类型重载sum，但使用完全相同的函数体。

The function `sum` could be overloaded for a lot of types, and it could make sense for all of them to have the same body. For cases such as this, C++ has the ability to define functions with generic types, known as *function templates*. Defining a function template follows the same syntax as a regular function, except that it is preceded by the `template` keyword and a series of template parameters enclosed in angle-brackets <>:

对于许多类型，函数sum都可以重载，并且所有类型都具有相同的函数体是有意义的。对于这种情况，c++能够使用泛型类型定义函数，称为函数模板。定义函数模板遵循与普通函数相同的语法，除了它的前面有template关键字和一系列用尖括号<>括起来的模板形参:

`template <template-parameters> function-declaration`

The template parameters are a series of parameters separated by commas. These parameters can be generic template types by specifying either the `class` or `typename` keyword followed by an identifier. This identifier can then be used in the function declaration as if it was a regular type. For example, a generic `sum` function could be defined as:

模板参数是由逗号分隔的一系列参数。这些参数可以是泛型模板类型，只要指定class或typename关键字，后跟一个标识符。然后可以在函数声明中使用该标识符，就像它是一个常规类型一样。例如，一个泛型求和函数可以定义为:

```
template <class SomeType>
SomeType sum (SomeType a, SomeType b)
{
  return a+b;
}
```

It makes no difference whether the generic type is specified with keyword `class` or keyword `typename` in the template argument list (they are 100% synonyms in template declarations).

在模板参数列表中，泛型类型是用关键字class指定还是用关键字typename指定都没有区别(它们在模板声明中100%是同义词)。

In the code above, declaring `SomeType` (a generic type within the template parameters enclosed in angle-brackets) allows `SomeType` to be used anywhere in the function definition, just as any other type; it can be used as the type for parameters, as return type, or to declare new variables of this type. In all cases, it represents a generic type that will be determined on the moment the template is instantiated.

在上面的代码中，声明SomeType(用尖括号括起来的模板形参中的泛型类型)允许SomeType在函数定义的任何地方使用，就像任何其他类型一样;它可以用作参数的类型，作为返回类型，或者声明这种类型的新变量。在所有情况下，它都表示一个泛型类型，该类型将在模板实例化时确定。

Instantiating a template is applying the template to create a function using particular types or values for its template parameters. This is done by calling the *function template*, with the same syntax as calling a regular function, but specifying the template arguments enclosed in angle brackets:

实例化模板就是应用模板来创建一个函数，该函数使用模板参数的特定类型或值。这可以通过调用函数模板来实现，语法与调用普通函数相同，但要指定用尖括号括起来的模板参数:

`name <template-arguments> (function-arguments)`

For example, the `sum` function template defined above can be called with:

```
x = sum<int>(10,20);
```

The function `sum<int>` is just one of the possible instantiations of function template `sum`. In this case, by using `int` as template argument in the call, the compiler automatically instantiates a version of `sum` where each occurrence of `SomeType` is replaced by `int`, as if it was defined as:

函数sum只是函数模板sum的一个可能实例化。在本例中，通过在调用中使用int作为模板实参，编译器自动实例化sum的一个版本，其中SomeType每次出现都会被int替换，就好像它定义为:

```
int sum (int a, int b)
{
  return a+b;
}
```

Let's see an actual example:

```
// function template
#include <iostream>
using namespace std;

template <class T>
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6, k;
  double f=2.0, g=0.5, h;
  k=sum<int>(i,j);
  h=sum<double>(f,g);
```

```
    cout << k << '\n';
    cout << h << '\n';
    return 0;
}
```

In this case, we have used `T` as the template parameter name, instead of `SomeType`. It makes no difference, and `T` is actually a quite common template parameter name for generic types.

在本例中，我们使用了T作为模板参数名，而不是SomeType。这没有区别，T实际上是泛型类型的一个非常常见的模板参数名。

In the example above, we used the function template `sum` twice. The first time with arguments of type `int`, and the second one with arguments of type `double`. The compiler has instantiated and then called each time the appropriate version of the function.

在上面的例子中，我们使用了两次函数模板sum。第一次使用int类型的参数，第二次使用double类型的参数。编译器已经实例化了函数，然后每次调用相应版本的函数。

Note also how `T` is also used to declare a local variable of that (generic) type within `sum`:

还请注意T是如何在sum中声明该(泛型)类型的局部变量的:

```
T result;
```

Therefore, result will be a variable of the same type as the parameters `a` and `b`, and as the type returned by the function.
In this specific case where the generic type `T` is used as a parameter for `sum`, the compiler is even able to deduce the data type automatically without having to explicitly specify it within angle brackets. Therefore, instead of explicitly specifying the template arguments with:

因此，result将是与参数a和b相同类型的变量，并且是函数返回的类型。在这种特殊情况下，泛型类型T被用作sum的参数，编译器甚至可以自动推断数据类型，而不必在尖括号中显式指定它。因此，不用显式指定模板参数:

```
k = sum<int> (i,j);
h = sum<double> (f,g);
```

It is possible to instead simply write:

```
k = sum (i,j);
h = sum (f,g);
```

without the type enclosed in angle brackets. Naturally, for that, the type shall be unambiguous. If `sum` is called with arguments of different types, the compiler may not be able to deduce the type of `T` automatically.

没有用尖括号括起来的类型。当然，为此，类型应该是明确的。如果使用不同类型的参数调用' sum '，编译器可能无法自动推断' T '的类型。

Templates are a powerful and versatile feature. They can have multiple template parameters, and the function can still use regular non-templated types. For example:

模板是一种功能强大且通用的特性。它们可以有多个模板参数，函数仍然可以使用常规的非模板类型。例如:

```
// function templates
#include <iostream>
using namespace std;

template <class T, class U>
bool are_equal (T a, U b)
{
  return (a==b);
}

int main ()
{
  if (are_equal(10,10.0))
    cout << "x and y are equal\n";
  else
    cout << "x and y are not equal\n";
  return 0;
}
```

Note that this example uses automatic template parameter deduction in the call to `are_equal`:

注意，这个例子在are_equal的调用中使用了自动的模板参数推断:

```
are_equal(10,10.0)
```

Is equivalent to:

```
are_equal<int,double>(10,10.0)
```

There is no ambiguity possible because numerical literals are always of a specific type: Unless otherwise specified with a suffix, integer literals always produce values of type `int`, and floating-point literals always produce values of type `double`. Therefore `10` has always type `int` and `10.0` has always type `double`.

不可能有歧义，因为数字字面值总是特定的类型:除非用后缀指定，整型字面值总是产生int类型的值，浮点型字面值总是产生double类型的值。因此10的类型总是int, 10.0的类型总是double。

Non-type template arguments

The template parameters can not only include types introduced by `class` or `typename`, but can also include expressions of a particular type:

模板形参不仅可以包含由' class '或' typename '引入的类型，还可以包含特定类型的表达式:

```
// template arguments
#include <iostream>
using namespace std;

template <class T, int N>
T fixed_multiply (T val)
{
  return val * N;
}

int main() {
  std::cout << fixed_multiply<int,2>(10) << '\n';
  std::cout << fixed_multiply<int,3>(10) << '\n';
}
```

The second argument of the `fixed_multiply` function template is of type `int`. It just looks like a regular function parameter, and can actually be used just like one.

' fixed_multiply '函数模板的第二个参数类型为' int '。它只是看起来像一个普通的函数参数，实际上可以像一个一样使用。

But there exists a major difference: the value of template parameters is determined on compile-time to generate a different instantiation of the function `fixed_multiply`, and thus the value of that argument is never passed during runtime: The two calls to `fixed_multiply` in `main` essentially call two versions of the function: one that always multiplies by two, and one that always multiplies by three. For that same reason, the second template argument needs to be a constant expression (it cannot be passed a variable).

但存在一个重大的区别:确定模板参数的值在编译时生成一个不同的的函数实例化“fixed_multiply”,因此这一观点的价值从来都不是通过在运行时:两个调用“fixed_multiply””主要”实际上是调用函数的两个版本:一个总是乘以2，一个总是乘以3。出于同样的原因，第二个模板参数需要是常量表达式(不能传递变量)。