C++学习笔记十六-模板和泛型编程(一)

概述:所谓泛型编程就是以独立于任何特定类型的方式编写代码。使用泛型程序时，我们需要提供具体程序实例所操作的类型或值。第二部分中描述的标准库的容器、迭代器和算法都是泛型编程的例子。**在 C++ 中，模板是泛型编程的基础。模板是创建类或函数的蓝图或公式。**

一 、模板定义

1. 定义函数模板：

compare 的模板版本：

```
// implement strcmp-like generic compare function
// returns 0 if the values are equal, 1 if v1 is larger, -1 if v1 is smaller
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

a. 模板定义以关键字 template 开始，后接**模板形参表**，模板形参表是用尖括号括住的一个或多个**模板形参**的列表，形参之间以逗号分隔。 模板形参表不能为空。

b. 模板形参表很像函数形参表，函数形参表定义了特定类型的局部变量但并不初始化那些变量，在运行时再提供实参来初始化形参。

c. 模板形参可以是表示类型的**类型形参**，也可以是表示常量表达式的**非类型形参**。非类型形参跟在类型说明符之后声明,类型形参跟在关键字 class 或 typename 之后定义，例如，class T 是名为 T 的类型形参，在这里 class 和 typename 没有区别。

2.使用函数模板:**使用函数模板时，编译器会推断哪个（或哪些）模板实参绑定到模板形参。一旦编译器确定了实际的模板实参，就称它实例化了函数模板的一个实例。**实质上，编译器将确定用什么类型代替每个类型形参，以及用什么值代替每个非类型形参。推导出实际模板实参后，编译器使用实参代替相应的模板形参产生编译该版本的函数。**编译器承担了为我们使用的每种类型而编写函数的单调工作。**

2.类模板:类模板也是模板，因此必须以关键字 template 开头，后接模板形参表。

a.除了模板形参表外，类模板的定义看起来与任意其他类问相似。类模板可以定义数据成员、函数成员和类型成员，也可以使用访问标号控制对成员的访问，还可以定义构造函数和析构函数等等。在类和类成员的定义中，可以使用模板形参作为类型或值的占位符，在使用类时再提供那些类型或值。

b.**与调用函数模板形成对比，使用类模板时，必须为模板形参显式指定实参.编译器使用实参来实例化这个类的特定类型版本。**

3.模板形参:

a.像函数形参一样，程序员为模板形参选择的名字没有本质含义。

b.可以给模板形参赋予的唯一含义是区别形参是类型形参还是非类型形参。如果是类型形参，我们就知道该形参表示未知类型，如果是非类型形参，我们就知道它是一个未知值。

4.模板形参作用域:

a.模板形参的名字可以在声明为模板形参之后直到模板声明或定义的末尾处使用。

b.模板形参遵循常规名字屏蔽规则。与全局作用域中声明的对象、函数或类型同名的模板形参会屏蔽全局名字.

c.使用模板形参名字的限制：用作模板形参的名字不能在模板内部重用(不能再次作为类型来使用)。

```
template <class T> T calc(const T &a, const T &b)
{
    typedef double T; // error: redeclares template parameter T
    T tmp = a;
    // ...
    return tmp;
}
```

d.这一限制还意味着模板形参的名字只能在同一模板形参表中使用一次：

```
// error: illegal reuse of template parameter name V
template <class V, class V> V calc(const V&, const V&) ;
```

e.当然，正如可以重用函数形参名字一样，模板形参的名字也能在不同模板中重用.

f.同一模板的声明和定义中，模板形参的名字不必相同。

```
// all three uses of calc refer to the same function template
// forward declarations of the template
template <class T> T calc(const T&, const T&) ;
template <class U> U calc(const U&, const U&) ;
```

```
// actual definition of the template

template <class Type>

Type calc(const Type& a, const Type& b) { /* ... */ }
```

g.每个模板类型形参前面必须带上关键字 class 或 typename，每个非类型形参前面必须带上类型名字，省略关键字或类型说明符是错误的

5.typename 与 class 的区别:

在函数模板形参表中，关键字 typename 和 class 具有相同含义，可以互换使用，两个关键字都可以在同一模板形参表中使用

6.在模板定义内部指定类型:在类型之前指定 typename 没有害处，因此，即使 typename 是不必要的，也没有关系。

如果希望编译器将 size_type 当作类型，则必须显式告诉编译器这样做：

```
template <class Parm, class U>

Parm fcn(Parm* array, U value)

{

    typename Parm::size_type * p; // ok: declares p to be a pointer

}
```

7.非类型模板形参:

a.在调用函数时非类型形参将用值代替，值的类型在模板形参表中指定。

b.**模板非类型形参是模板定义内部的常量值，在需要常量表达式的时候，可使用非类型形参（例如，像这里所做的一样）指定数组的长度.**

8.编写泛型程序：

a.在函数模板内部完成的操作限制了可用于实例化该函数的类型。程序员的责任是，保证用作函数实参的类型实际上支持所用的任意操作，以及保证在模板使用哪些操作的环境中那些操作运行正常。

b.编写模板代码时，对实参类型的要求尽可能少是很有益的。

c.编写泛型代码的两个重要原则：

- 模板的形参是 const 引用。
-
-
- 函数体中的测试只用 < 比较。
-
-

9.警告：链接时的编译时错误：

-

重要的是，要认识到编译模板定义的时候，对程序是否有效所知不多。类似地，甚至可能会在已经成功编译了使用模板的每个文件之后出现编译错误。只在实例化期间检测错误的情况很少，错误检测可能发生在链接时。

-

二 、实例化

概述:模板是一个蓝图，它本身不是类或函数。编译器用模板产生指定的类或函数的特定类型版本。产生模板的特定类型实例的过程称为实例化，这个术语反映了创建模板类型或模板函数的新“实例”的概念。

模板在使用时将进行实例化，类模板在引用实际模板类类型时实例化，函数模板在调用它或用它对函数指针进行初始化或赋值时实例化。

1.类的实例化:

a.类模板的每次实例化都会产生一个独立的类类型。为 int 类型实例化的 Queue 与任意其他 Queue 类型没有关系，对其他 Queue 类型成员也没有特殊的访问权。

b.想要使用类模板，就必须显式指定模板实参。

c.类模板不定义类型，只有特定的实例才定义了类型。特定的实例化是通过提供模板实参与每个模板形参匹配而定义的。模板实参在用逗号分隔并用尖括号括住的列表中指定：

```
Queue<int> qi;          // ok: defines Queue that holds ints
```

d.用模板类定义的类型总是包含模板实参。例如，`Queue` 不是类型，而 `Queue<int>` 或 `Queue<string>` 是类型。

2.函数模板实例化:

A.使用函数模板时，编译器通常会为我们推断模板实参：

```
int main()
{
```

```
        compare(1, 0);              // ok: binds template parameter to int

        compare(3.14, 2.7);         // ok: binds template parameter to double

        return 0;

    }
```

B.模板实参推断:要确定应该实例化哪个函数，编译器会查看每个实参。如果相应形参声明为类型形参的类型，则编译器从实参的类型推断形参的类型。

C.类型形参的实参的受限转换:**一般而论，不会转换实参以匹配已有的实例化，相反，会产生新的实例。除了产生新的实例化之外，编译器只会执行两种转换：**

a. const 转换:接受 const 引用或 const 指针的函数可以分别用非 const 对象的引用或指针来调用，无须产生新的实例化。如果函数接受非引用类型，形参类型实参都忽略 const，即，无论传递 const 或非 const 对象给接受非引用类型的函数，都使用相同的实例化。

b.数组或函数到指针的转换:如果模板形参不是引用类型，则对数组或函数类型的实参应用常规指针转换。数组实参将当作指向其第一个元素的指针，函数实参当作指向函数类型的指针。

D.类型转换的限制只适用于类型为模板形参的那些实参。

用普通类型定义的形参可以使用常规转换,下面的函数模板 sum 有两个形参:

```
template <class Type> Type sum(const Type &op1, int op2)
{
    return op1 + op2;
```

```
    }
```

E.模板实参推断与函数指针:**可以使用函数模板对函数指针进行初始化或赋值,这样做的时候，编译器使用指针的类型实例化具有适当模板实参的模板版本。获取函数模板实例化的地址的时候，上下文必须是这样的：它允许为每个模板形参确定唯一的类型或值。**

```
  // overloaded versions of func; each take a different function pointer type

    void func(int(*) (const string&, const string&));

    void func(int(*) (const int&, const int&));

    func(compare); // error: which instantiation of compare?
```

F.在返回类型中使用类型形参:

a.**指定返回类型的一种方式是引入第三个模板形参,它必须由调用者显式指定,也就是说返回类型不能推断,必须显示指定：**

```
    // T1 cannot be deduced: it doesn't appear in the function parameter list

    template <class T1, class T2, class T3>

    T1 sum(T2, T3);
```

b.显式模板实参从左至右对应模板形参相匹配，第一个模板实参与第一个模板形参匹配，第二个实参与第二个形参匹配，以此类推。假如可以从函数形参推断，则结尾（最右边）形参的显式模板实参可以省略。

G.显式实参与函数模板的指针:通过使用显式模板实参能够消除二义性：

```
    template <typename T> int compare(const T&, const T&);
```

```
    // overloaded versions of func; each take a different
function pointer type

    void func(int(*) (const string&, const string&));

    void func(int(*) (const int&, const int&));

    func(compare<int>); // ok: explicitly specify which
version of compare
```

C++学习笔记十六-模板和泛型编程(二)

一、类模板成员

1.模板作用域中模板类型的引用：

A.在类模板的作用域内部，可以用它的非限定名字引用该类。

B.通常，当使用类模板的名字的时候，必须指定模板形参。这一规则有个例外：在类本身的作用域内部，可以使用类模板的非限定名。例如，在默认构造函数和复制构造函数的声明中，名字 Queue 是 Queue<Type> 缩写表示。实质上，编译器推断，当我们引用类的名字时，引用的是同一版本。因此，复制构造函数定义其实等价于：

```
    Queue<Type>(const Queue<Type> &Q): head(0), tail(0)

                { copy_elems(Q); }
```

C.编译器不会为类中使用的其他模板的模板形参进行这样的推断，因此，在声明伙伴类 QueueItem 的指针时，必须指定类型形参：

```
    QueueItem<Type> *head;    // pointer to first element
in Queue
```

2. 类模板成员函数 : 类模板成员函数的定义具有如下形式：

A.必须以关键字 template 开关，后接类的模板形参表。

B.必须指出它是哪个类的成员。

C.类名必须包含其模板形参。

从这些规则可以看到，在类外定义的 Queue 类的成员函数的形式应该是：

```
template <class T> ret-type Queue<T>::member-name
```

3.类模板成员函数的实例化：类模板的成员函数本身也是函数模板。像任何其他函数模板一样，需要使用类模板的成员函数产生该成员的实例化。**与其他函数模板不同的是，在实例化类模板成员函数的进修，编译器不执行模板实参推断，相反，类模板成员函数的模板形参由调用该函数的对象的类型确定。**

对象的模板实参能够确定成员函数模板形参，这一事实意味着，调用类模板成员函数比调用类似函数模板更灵活。用模板形参定义的函数形参的实参允许进行常规转换：

```
Queue qi; // instantiates class Queue
short s = 42;
int i = 42;
// ok: s converted to int and passed to push
qi.push(s); // instantiates Queue::push(const int&)
qi.push(i); // uses Queue::push(const int&)
f(s);       // instantiates f(const short&)
f(i);       // instantiates f(const int&)
```

4.何时实例化类和成员：类模板的成员函数只有为程序所用才进行实例化。如果某函数从未使用，则不会实例化该成员函数。这一行为意味着，用于实例化模板的类型只需满足实际使用的操作的

要求。只接受一个容量形参的顺序容器构造函数就是这样的例子，该构造函数使用元素类型的默认构造函数。如果有一个没有定义默认构造函数的类型，仍然可以定义容器来保存该类型，但是，不能使用只接受一个容量的构造函数。

5. 定义模板类型的对象时，该定义导致实例化类模板。定义对象也会实例化用于初始化该对象的任一构造函数，以及该构造函数调用的任意成员：

```
// instantiates Queue<int> class and
Queue<int>::Queue()
Queue<string> qs;
qs.push("hello"); // instantiates
Queue<int>::push</int></string></int></int>
```

Queue 类中的 QueueItem 成员是指针。类模板的指针定义不会对类进行实例化，只有用到这样的指针时才会对类进行实例化。因此，在创建 Queue 对象进不会实例化 QueueItem 类，相反，在使用诸如 front、push 或 pop 这样的 Queue 成员时才实例化 QueueItem 类。

6. 非类型形参的模板实参：这个模板有两个形参，均为非类型形参。当用户定义 Screen 对象时，必须为每个形参提供常量表达式以供使用。类在默认构造函数中使用这些形参设置默认 Screen 的尺寸。

像任意类模板一样，使用 Screen 类型时必须显式声明形参值：

```
Screen<24,80>
hp2621; //
screen 24
lines by 80
characters
```

非类型模板实参必须是编译时常量表达式。

二、类模板中的友元声明：

1.在类模板中可以出现三种友元声明，每一种都声明了与一个或多个实体友元关系：

A.普通非模板类或函数的友元声明，将友元关系授予明确指定的类或函数。

非模板类或非模板函数可以是类模板的友元：

```
template <class Type> class Bar {
        // grants access to ordinary, nontemplate class and function
        friend class FooBar;
        friend void fcn();
        // ...
};
```

这个声明是说，FooBar 的成员和 fcn 函数可以访问 Bar 类的任意实例的 private 成员和 protected 成员。

B.类模板或函数模板的友元声明，授予对友元所有实例的访问权。

友元可以是类模板或函数模板：

```
template <class Type> class Bar {
    // grants access to Foo1 or templ_fcn1 parameterized by any type
    template <class T> friend class Foo1;
    template <class T> friend void templ_fcn1(const T&);
    // ...
```

```
};
```

这些友元声明使用与类本身不同的类型形参，该类型形参指的是 Foo1 和 templ_fcn1 的类型形参。在这两种情况下，都将没有数目限制的类和函数设为 Bar 的友元。Foo1 **的友元声明是说，**Foo1 **的友元声明是说，** Foo1 **的任意实例都可以访问** Bar **的任意实例的私有元素，类似地，** temp_fcn1 **的任意实例可以访问** Bar **的任意实例。**

这个友元声明在 Bar 与其友元 Foo1 和 templ_fcn1 的每个实例之间建立了一对多的映射。对 Bar 的每个实例而言，Foo1 或 templ_fcn1 的所有实例都是友元。

C.只授予对类模板或函数模板的特定实例的访问权的友元声明。

```
template <class T> class Foo3;
template <class T> void templ_fcn3(const T&);
template <class Type> class Bar {
    // each instantiation of Bar grants access to the
    // version of Foo3 or templ_fcn3 instantiated with the same type
    friend class Foo3<Type>;
    friend void templ_fcn3<Type>(const Type&);
    // ...
};
```

这些友元定义了 Bar 的特定实例与使用同一模板实参的 Foo3 或 templ_fcn3 的实例之间的友元关系。每个 Bar 实例有一个相关的 Foo3 和 templ_fcn3 友元：

```
Bar<int> bi;    // Foo3<int> and templ_fcn3<int> are friends
```

```
    Bar<string> bs; // Foo3<string>, templ_fcn3<string>
are friends
```

D.声明依赖性：

当授予对给定模板的所有实例的访问权时候，在作用域中不需要存在该类模板或函数模板的声明。实质上，编译器将友元声明也当作类或函数的声明对待。

想要限制对特定实例化的友元关系时，必须在可以用于友元声明之前声明类或函数：

```
     template <class T> class A;
    template <class T> class B {
    public:
        friend class A<T>;       // ok: A is known to be a
template
        friend class C;          // ok: C must be an ordinary,
nontemplate class
        template <class S> friend class D; // ok: D is a
template
        friend class E<T>;       // error: E wasn't declared
as a template
        friend class F<int>;     // error: F wasn't declared
as a template
     };
```

如果没有事先告诉编译器该友元是一个模板，则编译器将认为该友元是一个普通非模板类或非模板函数。

三、成员模板

1.任意类（模板或非模板）可以拥有本身为类模板或函数模板的成员，这种成员称为**成员模板**，成员模板不能为虚。定义成员模板模板成员声明看起来像任意模板的声明一样：

```
template <class Type> class Queue {
public:
    // construct a Queue from a pair of iterators on some sequence
    template <class It>
    Queue(It beg, It end):
        head(0), tail(0) { copy_elems(beg, end); }
    // replace current Queue by contents delimited by a pair of iterators
    template <class Iter> void assign(Iter, Iter);
    // rest of Queue class as before
private:
    // version of copy to be used by assign to copy elements from iterator range
    template <class Iter> void copy_elems(Iter, Iter);
};
```

成员声明的开关是自己的模板形参表。构造函数和 assign 成员各有一个模板类型形参，这些函数使用该类型形参作为其函数形参的类型，它们的函数形参是指明要复制元素范围的迭代器。

2.在类外部定义成员模板：

当成员模板是类模板的成员时，它的定义必须包含类模板形参以及自己的模板形参。首先是类模板形参表，后面接着成员自己的模板形参表。assign 函数定义的形式为

```
template <class T> template <class Iter>
```

第一个模板形参表 template<class T> 是类模板的，第二个模板形参表 template<class Iter> 是成员模板的。

3.成员模板遵循常规访问控制：成员模板遵循与任意其他类成员一样的访问规则。如果成员模板为私有的，则只有该类的成员函数和友元可以使用该成员模板。

4.成员模板和实例化：与其他成员一样，成员模板只有在程序中使用时才实例化。类模板的成员模板的实例化比类模板的普通成员函数的实例化要复杂一点。**成员模板有两种模板形参：由类定义的和由成员模板本身定义的。类模板形参由调用函数的对象的类型确定，成员定义的模板形参的行为与普通函数模板一样。这些形参都通过常规模板实参推断而确定。**

四、类模板的 static 成员

1.类模板可以像任意其他类一样声明 static 成员。以下代码：

```
template <class T> class Foo {
public:
    static std::size_t count() { return ctr; }
    // other interface members
private:
    static std::size_t ctr;
    // other implementation members
```

```
};

// Each object shares the same Foo<int>::ctrand
Foo<int>::count members

Foo<int> fi, fi2, fi3;

// has static members Foo<string>::ctrand
Foo<string>::count

Foo<string> fs;
```

每个实例化表示截然不同的类型，所以给定实例外星人所有对象都共享一个 static 成员。因此,Foo<int> 类型的任意对象共享同一 static 成员ctr，Foo<string> 类型的对象共享另一个不同的 ctr 成员。

2.使用类模板的 static 成员：

通常，可以通过类类型的对象访问类模板的 static 成员，或者通过使用作用域操作符直接访问成员。当然，当试图通过类使用 static 成员的时候，必须引用实际的实例化：

```
Foo<int> fi, fi2;              // instantiates
Foo<int> class

size_t ct = Foo<int>::count(); // instantiates
Foo<int>::count

ct = fi.count();               // ok: uses
Foo<int>::count

ct = fi2.count();              // ok: uses
Foo<int>::count

ct = Foo::count();             // error: which
template instantiation?
```

与任意其他成员函数一样，static 成员函数只有在程序中使用时才进行实例化。

3.定义 static 成员：

像使用任意其他 static 数据成员一样，必须在类外部出现数据成员的定义。在类模板含有 static 成员的情况下，成员定义必须指出它是类模板的成员：

```
template <class T>
size_t Foo<T>::ctr = 0; // define and initialize ctr
```

static 数据成员像定义在类外部的任意其他类成员一样定义，它用关键字 template 开头，后面接着类模板形参表和类名。在这个例子中，static 数据成员的名字以 Foo<T>:: 为前缀，表示成员属于类模板 Foo。

五、模板特化

引言：我们并不总是能够写出对所有可能被实例化的类型都最合适的模板。某些情况下，通用模板定义对于某个类型可能是完全错误的，通用模板定义也许不能编译或者做错误的事情；另外一些情况下，可以利用关于类型的一些特殊知识，编写比从模板实例化来的函数更有效率的函数。

1.函数模板的特化：模板特化（template specialization）是这样的一个定义，该定义中一个或多个模板形参的实际类型或实际值是指定的。特化的形式如下：

A.关键字 template 后面接一对空的尖括号（<>）；

B.再接模板名和一对尖括号，尖括号中指定这个特化定义的模板形参；

C.函数形参表；

D.函数体。

2.当模板形参类型绑定到 const char* 时，compare 函数的特化：

```
// special version of compare to handle C-style
character strings
```

```
template <>
int compare<const char*>(const char* const &v1,
                         const char* const &v2)
{
    return strcmp(v1, v2);
}
```

现在，**当调用 compare 函数的时候，传给它两个字符指针，编译器将调用特化版本**。编译器将为任意其他实参类型（包括普通 char*）调用泛型版本：

2.声明特化：

与任意函数一样，函数模板特化可以声明而无须定义。模板特化声明看起来与定义很像，但省略了函数体：

```
// declaration of function template explicit specialization
template<>
int compare<const char*>(const char* const&,
                         const char* const&);
```

这个声明由一个后接返回类型的空模板形参表（template<>），后接一对尖括号中指定的显式模板实参的函数名（可选），以及函数形参表构成。模板特化必须总是包含空模板形参说明符，即 template<>，而且，还必须包含函数形参表。如果可以从函数形参表推断模板实参，则不必显式指定模板实参：

```
// error: invalid specialization declarations
// missing template<>
int compare<const char*>(const char* const&,
                         const char* const&);

// error: function parameter list missing
template<> int compare<const char*>;
```

```
        // ok: explicit template argument const char* deduced from
parameter types
        template<> int compare(const char* const&,
                                                    const char*
const&);
```

3.函数重载与模板特化：当定义非模板函数的时候，对实参应用常规转换；当特化模板的时候，对实参类型不应用转换。在模板特化版本的调用中，**实参类型必须与特化版本函数的形参类型完全匹配，如果不完全匹配，编译器将为实参从模板定义实例化一个实例。**

4.不是总能检测到重复定义：如果程序由多个文件构成，模板特化的声明必须在使用该特化的每个文件中出现。不能在一些文件中从泛型模板定义实例化一个函数模板，而在其他文件中为同一模板实参集合特化该函数模板。

与其他函数声明一样，应在一个头文件中包含模板特化的声明，然后使用该特化的每个源文件包含该头文件。

六.类模板的特化：

1.为 C 风格字符串的 Queue 提供正确行为的一种途径，是为 const char* 定义整个类的特化版本：

```
    /* definition of specialization for const char*
     * this class forwards its work to Queue<string>;
     * the push function translates the const char*
parameter to a string
     * the front functions return a string rather than a
const char*
     */
    template<> class Queue<const char*> {
    public:
```

```
        // no copy control: Synthesized versions work for
this class

        // similarly, no need for explicit default
constructor either

        void push(const char*);

        void pop()                      {real_queue.pop();}

        bool empty() const              {return
real_queue.empty();}

        // Note: return type does not match template
parameter type

        std::string front()             {return
real_queue.front();}

        const std::string &front() const

                                        {return
real_queue.front();}

    private:

        Queue<std::string> real_queue; // forward calls to
real_queue

    };
```

A.值得注意的是，特化可以定义与模板本身完全不同的成员。如果一个特化无法从模板定义某个成员，该特化类型的对象就不能使用该成员。类模板成员的定义不会用于创建显式特化成员的定义。

B.类模板特化应该与它所特化的模板定义相同的接口，否则当用户试图使用未定义的成员时会感到奇怪。

2.类特化定义:在类特化外部定义成员时，成员之前不能加 template<> 标记。

我们的类只在类的外部定义了一个成员：

```
void Queue<const char*>::push(const char* val)
{
    return real_queue.push(val);
}
```

3.特化成员而不特化类:成员特化的声明与任何其他函数模板特化一样，必须以空的模板形参表开头：

```
// push and pop specialized for const char*
template <>
void Queue<const char*>::push(const char* const &);
template <> void Queue<const char*>::pop();
```

4.类模板的部分特化:

如果类模板有一个以上的模板形参，我们也许想要特化某些模板形参而非全部。使用类模板的部分特化可以做到这一点：

```
template <class T1, class T2>
class some_template {
    // ...
};
// partial specialization: fixes T2 as int and allows T1 to vary
template <class T1>
class some_template<T1, int> {
    // ...
};
```

类模板的部分特化本身也是模板。部分特化的定义看来像模板定义，这种定义以关键字 template 开头，接着是由尖括号（<>）括住的模板形参表。部分特化的模板形参表是对应的类模板定义形参表的子集。some_template 的部分特化只有一个名为 T1 的模板类型形参，第二个模板形参 T2 的实参已知为 int。部分特化的模板形参表只列出未知模板实参的那些形参。

5.使用类模板的部分特化：

部分特化与对应类模板有相同名字，即这里的 some_template。类模板的名字后面必须接着模板实参列表，前面例子中，模板实参列表是 <T1,int>。因为第一个模板形参的实参值未知，实参列表使用模板形参名 T1 作为占位符，另一个实参是类型 int，为 int 而部分特化模板。

像任何其他类模板一样，部分特化是在程序中使用时隐式实例化：

```
some_template<int, string> foo; // uses template

some_template<string, int> bar; // uses partial specialization
```

注意第二个变量的类型，形参为 string 和 int 的 some_template，既可以从普通类模板定义实例化，也可以从部分特化实例化。为什么选择部分特化来实例化该模板呢？**当声明了部分特化的时候，编译器将为实例化选择最特化的模板定义，当没有部分特化可以使用的时候，就使用通用模板定义。**foo 的实例化类型与提供的部分特化不匹配，因此，foo 的类型必然从通用类模板实例化，将 int 绑定到 T1 并将 string 绑定到 T2。部分特化只用于实例化第二个类型为 int 的 some_template 类型。

部分特化的定义与通用模板的定义完全不会冲突。部分特化可以具有与通用类模板完全不同的成员集合。类模板成员的通用定义永远不会用来实例化类模板部分特化的成员。

七、重载与函数模板

函数模板可以重载：可以定义有相同名字但形参数目或类型不同的多个函数模板，也可以定义与函数模板有相同名字的普通非模板函数。

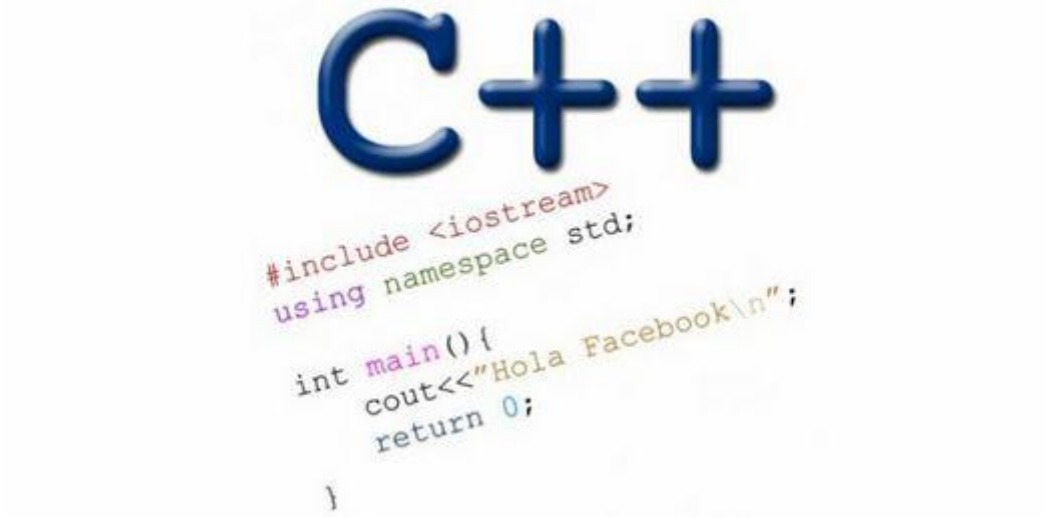
当然，声明一组重载函数模板不保证可以成功调用它们，重载的函数模板可能会导致二义性。

1.如果重载函数中既有普通函数又有函数模板，确定函数调用的步骤如下：

A. 为这个函数名建立候选函数集合，包括：
与被调用函数名字相同的任意普通函数。
任意函数模板实例化，在其中，模板实参推断发现了与调用中所用函数实参相匹配的模板实参。
B. 确定哪些普通函数是可行的（第 7.8.2 节）（如果有可行函数的话）。候选集合中的每个模板实例都 可行的，因为模板实参推断保证函数可以被调用。
C. 如果需要转换来进行调用，根据转换的种类排列可靠函数，记住，调用模板函数实例所允许的转换是有限的。
如果只有一个函数可选，就调用这个函数。
如果调用有二义性，从可行函数集合中去掉所有函数模板实例。
D. 重新排列去掉函数模板实例的可行函数。
如果只有一个函数可选，就调用这个函数。
否则，调用有二义性。

现代 C++设计更倾向于强大的泛型编程方法，但是在使用之前你可能需要花费很大代价去学习。这也多亏了 Andrei Alexandrescu 和 Scott Meyers 等人让泛型编程更好理解，并简化了应用。只是现代 C++设计还没有被广泛使用。

正如 Biarne Stroustrup 指出的那样，“C++ 是一种多范型的语言。”它支持不同风格的程序和范例，面向对象的编程只是其中之一，还有结构化编程和泛型编程。在过去几年 C++ 方面专家 Andrei Alexandrescu，Scott Meyers 和 Herb Sutter 推荐用户使用泛型编程，他们将它成为现代 C++设计。



关于现代 C++ 设计，Andrei Alexandrescu 这样说：

> 现代 C++ 设计定义并系统地使用通用组件——高度灵活的设计品是少量的 orthogonal body of code 和丰富的行为的融合和匹配。

他提出了如下三个观点：

- 现代 C++ 设计定义并系统地使用通用组件
- 高灵活度
- 使用少量 orthogonal body of code 得到丰富的行为

泛型的灵活性

为了更加清楚的了解泛型的灵活性，下面比较一下在 OOP 和泛型中税收计算类的实现。

OOP 实现：

```
class CTaxCalculator{

    double CalculateTax(double param1, double param2){

        double tax = 0;

        // use m_calculator  to calculate tax

        return tax;

    }

    ICalculator*  m_calculator;
    public:
    void SetCalculator(ICalculator* calculator)
    {
        m_calculator = calculator;
    }

};
```

CTaxCalculator 类能告诉我们什么？

我是 CTaxCalculator 类，我知道如何计算税收，但我只与 ICalculator 类型的类协作。我拒绝与任何其他任何非 ICalculator 类合作，即使它可以帮我计算税。

泛型实现：

```
template<typename T>
class CGenericTaxCalculator
{
    double CalculateTax(double param1, double param2){

        double tax = 0;

        //  algo using m_calculator
        //  to calculate the tax

        return tax;

    }
    T*  m_calculator;

};
```

CGenericTacCalculator 类知道如何计算税收，并且能够和任何类型合作进行税收计算。

这使得泛型编程更自然的和灵活，第一种方法就想是在现实世界中，一个公司只接受一个从特定学校毕业的开发人员，拒绝其他人即使他们有相应的技术。

但代码变得灵活的代价是变得难以理解。事实上在 OOP 中我可以直接去看 ICalculator 类的定义，就可以知道我们我们能从这个类型中获得什么。而在泛型编程中我们很难确切知道能从参数模板中得到什么，哪些参数是必须包含的?是否必须来自一个特定的类型?

的确 C++ 特性和 SFINAE 机制能帮助我们定义从模板参数中得到什么，但这项先进的特性是很少 C++ 开发者掌握的。

C++ 设计师们意识到这个问题，并在新的 C++ 标准中，添加了许多模板编程的改进。大家能看到非常有趣 “Concept-lite” 特性，它用来定义参数模板的限制。

泛型编程产生的代码更加简洁

拷贝代码使得维护变得困难,而且可能产生各种错误。使用泛型编程可以避免有害的重复。

第二个小例子：计算两个数字之和。下面是不使用模板完成的代码：

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

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

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

使用模板函数，代码变得更加简洁：

```
template<typename T>
T sum(T a, T b)
{
    return a + b;
}
```

这只是展示使用模板获得简洁代码的一个小例子，STL 和 boost 包含更高级的算法，使得泛型编程个更加强大能够取代 boilerplate code。

但是为什么泛型编程没有得到广泛的使用？

泛型编程听起来更加自然和灵活，它可以提供比 OOP 更多的用法，然而许多开发人员发现它非常复杂，很难学习和使用。

- 代码变得不清晰，更难维护。
- 编译错误更难懂。

如果你看一下知名的 C++ 开源设计，就会知道泛型编程大部分只用在 C++ 库里，像 stl，boost，loki 和 folly，但是很少有使用反省编程的应用，虽然他们大部分都用了模板库。

总结

现代 C++ 设计更倾向于强大的泛型编程方法，但是在使用之前你可能需要花费很大代价去学习。感谢 Andrei Alexandrescu、Scott Meyers 和 Herb Sutter 等人的努力让泛型编程更好理解，并简化了其应用。但是现代 C++ 设计还没有被 C++ 社区广泛使用。