模板与泛型编程

函数模板

1. 在泛型编程中，在编译的时候就能获知类型（编译器使用实参的类型来确定绑定到模板参数的类型）
2. 定义模板类型参数：

```
template <typename T,U> //错误 每个模板类型参数都要用typename声明
template <typename T,typename U>    //正确
template <typename T,class U>       //typename和class没有什么不同
```

3. 在模板中还可以定义非类型参数，表示一个值而非一个类型，通过特定的类型名指定非类型参数。绑定到非类型整型参数的实参必须是一个常量表达式，绑定到指针或引用非类型参数的实参必须具有**静态的生存期**。（指针参数也可以用nullptr）

```
template<unsigned N,unsigned M>
int compare(const char (&p1)[N],const char (&p2)[M]){
    return strcmp(p1,p2);
}
compare("hi","mom");
//编译器会用字面常量的大小来替换N和M，最终实例化的版本为：
int compare(const char (&p1)[3],const char (&p2)[4])
//因为编译器会在一个字符串字面常量的末尾插入一个空字符作为终结符
```

   上面的代码中，当函数形参为引用时，传递的是整个数组，因此需要指定数组的大小。
4. 为了生成一个实例化版本，编译器需要掌握函数模板或类模板成员函数的定义，因此模板的头文件通常既包括声明也包括定义。
5. 模板知道实例化时才会生成代码：
   - 编译模板本身时，编译器只是检查语法错误
   - 编译器遇到模板使用时，通常会检查实参数目是否正确，参数类型是否匹配
   - 模板实例化时，只有此时才能发现类型相关的错误，这类错误可能在链接时才报告
6. 一个类模板的每个实例都形成一个独立的类。类型class<int>与任何其他class类型，如class<double>没有关联，也不会对任何其他class类型的成员有特殊访问权限。

```
template<typename N>
class Test{
    public:
        static void func(){
            cout<<num++<<' ';
        }
    private:
        static int num;
};
template<typename N>
int Test<N>::num=0;     //注意静态成员的定义

//测试
int main(){
    Test<int> T1,T3;
    T1.func();          //输出0
    T1.func();          //输出1
    Test<double> T2;
    T2.func();          //输出0
    T3.func();          //输出2
}
```

注意类的静态成员需要在类外定义，当类为类模板时定义的时候也要声明为是属于类模板的作用域class<T>::

7. 由于模板不是类型，所以我们不能顶一个typedef引用一个模板，但我们可以用typedef引用一个模板实例。但新标准允许使用using为类模板定义一个类型别名：

```
template<typename T> using twin=pair<T,T>
twin<string> authors;   //则authors是一个pair<string,string>

//我们页可也固定一个或多个模板参数
template<typename T> using twin=pair<T,unsigned>
twin<string> authors;   //则authors是一个pair<string,unsigned>
```

8. 假如T是一个模板类型参数，当编译器遇到类似T::mem这样的代码时，它不知道mem是一个类型成员还是一个static数据成员，直到实例化时才会知道。但是为了处理模板，编译器必须知道名字是否表示一个类型。例如，假定T是一个类型参数的名字，当编译器遇到如下形式的语句时：
T::size_type *p; 它需要知道我们是在定义一个名为p的变量还是将一个名为size_type的static数据成员与名为p的变量相乘。
默认情况下C++假定通过作用域运算符访问的名字不是类型，因为如果我们希望使用一个模板类型参数的类型成员，就必须显示告诉编译器该名字是一个类型，可以用关键字**typename**实现：

```
template<typename T>
//表明返回值是 T类型的作用域下的value_type类型
typename T::value_type top(const T& c);
```

9. 默认模板实参：

```
//默认模板实参指出cmp将使用less函数对象类
//默认函数实参指出f将是类型F的一个默认初始化对象
template <typename T,typename F=less<T>>
int cmp(const T &v1,const T &v2, F f=F());
```

当一个类模板为其所有模板参数都提供了默认实参且我们希望使用这些默认实参时，必须在模板名之后跟一个空的尖括号

```
template <class T=int>
class Test{

};
Test<double> t1;
Test<> t;   //表示Test<int>
```

10. 一个类可以包含本身是模板的成员函数，这种成员被称为成员模板，成员模板不能是虚函数。
11. 模板被使用时才会进行实例化，这意味着相同的实例可能出现在多个对象文件中。在大系统中，在多个文件中实例化相同模板的额外开销可能非常严重。新标准中我们可以通过显式实例化来避免这种开销。形式如下：

```
//declaration是一个类或者函数声明，其中所有模板参数已被替换为模板实参
extern template declaration;    //实例化声明
templat declaration;            //实例化定义
```

编译器遇到extern模板声明时，不会在本文件中生成实例化代码，因为该关键字表示承诺在程序其他位置有该实例化的一个非extern声明。对于一个给定的实例化版本，可能有多个extern声明，但必须只有一个定义。

12. 将实参传递给带模板类型的函数形参时，能够自动应用的类型转换只有const转换以及数组或函数到指针的转换

```
template <typename T>
void cmp(const T&,const T&);
long lng;
cmp(lng,1024);  //错误 1024是int型 因此cmp实参类型不同 long 和 int
```

13. 指定显式模板实参：当无法推断模板的类型时，调用者必须为模板提供显式模板实参。

```
template<typename T1,typename T2,typename T3>
T1 sum(T2,T3);
//T1是显式指定的 T2和T3是从函数实参类型推断而来的
int i=1;
long lng=2;
auto val=sum<long long>(i,lng); // long long sum(int,long)
```

显式模板实参按由左到右的顺序与对应的模板参数匹配：第一个模板实参与第一个模板参数匹配，依此类推。只有尾部参数的显式模板实参才可以忽略，而且前提是它们可以从函数参数推断出来。**即上面代码中的“long long”对应了T1在template中的位置**。

```
template<typename T1,typename T2,typename T3>
T3 sum(T2,T1);
//若这样定义 则用户必须指定所有三个模板参数
int i=1;
long lng=2;
//下面的long long指定的是T1 T2和T3还未知且无法推断
auto val=sum<long long>(i,lng); //错误
//下面代码等价于 long long sum(int,long)
auto val2=sum<long long,int,long>(i,lng); //正确
```

14. 对于模板类型参数已经显式指定了的函数，其实参可以进行正常的类型转换：

```
template<typename T>
void cmp(T,T);

long lng;
cmp(lng,1024);      //错误 long 和 int(1024) 不同类型
cmp<long>(lng,1024) //正确 因为已经显式指定了模板类型参数 1024可以转为long
```

15. 尾置返回类型：尾置返回出现在参数列表之后，**可以使用函数的参数**。用于不确定返回结果的准确类型，又无法显示指定。

```
template<typename It>
//指定func的返回类型为beg所指向的元素的类型(的引用)
auto func(It beg,It end)->decltype(*beg){
    return *beg;
}
```

上面代码中，编译器在遇到函数的参数列表之前，beg是不存在的，所以为了定义此函数，需要使用尾置返回类型

16. 进行类型转换的标准库模板类：所有迭代器操作都不会生成元素，只能生成元素的引用。为了获得元素类型（而非引用），可以使用标准库的类型转换模板（定义在头文件type_traits中）。
可以使用**remove_reference**来获得元素类型。remove_reference模板有一个模板类型参数和一个名为type的public类型成员。若我们用一个引用类型实例化remove_reference，则type将表示被引用的类型。如实例化remove_reference<int&>时type表示的是int。将上面代码改为返回的是元素类型而非引用：

```
template<typename It>
//指定func的返回类型为beg所指向的元素的类型
auto func(It beg,It end)->
    typename remove_reference<decltype(*beg)>::type
{
    return *beg;
}
```

用typename显式指明type是remove_reference类中定义的类型，而不是数据成员

17. 模板实参推断和引用：当一个函数参数是模板类型参数的一个普通引用时（形如T&），规定我们只能传递给它一个左值。

```
template<typename T> void func(T&);
func(i);    //i是一个int，T也是一个int
func(ci);   //ci是一个const int，T也是const int
func(5);    //错误

template<typenam T> void func(const T&);
func(5);    //正确，一个const&参数可以绑定到一个右值 因此T是int
```

18. 当我们将一个左值传递给函数的右值引用参数且此右值引用指向模板类型参数（如T&&）时，**编译器推断模板类型参数为实参的左值引用类型**。因此当func(T&&)，调用func((int)i)时，编译器推断T的类型为int&。所以，如果我们间接创建一个引用的引用，则这些引用形成了“折叠”。对于一个给定类型X：
    - X& &、X& &&和X&& & 都将折叠成类型X&
    - X&& &&折叠成X&&

    因此**如果一个函数参数是指向模板参数类型的右值引用，则可以传递给它任意类型的实参**。如果将一个左值传递给这样的参数，则函数参数被实例化为一个普通的左值引用。
19. 理解std::move：move函数的定义如下

```
template<typename T>
typename remove_reference<T>::type&& move(T&& t){
    return static_cast<typename remove_reference<T>::type&&>(t);
}
```

首先move函数的参数T&&是一个指向模板类型参数的右值引用，通过引用折叠此参数可以与任何类型的实参匹配。

```
string s1("hi"),s2;
s2=std::move(string("bye"));    //正确，从一个右值移动数据
s2=std::move(s1);               //正确，但赋值之后s1的值是不确定的
```

对于s2=std::move(string("bye")); T被推断为string，因此remove_reference<T>::type的类型是string，因此move返回的类型是string&&，而函数参数t的类型为string&&。因此move返回的是static_cast<string&&>(t)，t的类型已经是string&&，于是类型转换什么都不做。

对于s2=std::move(s1); T被推断为string&，因此remove_reference<T>::type的类型是string，move的返回类型是string&&，move的参数t实例化为string& &&，折叠为string&，因此这个调用被实例化为 string&& move(string &t)。在此情况下，t的类型为string&，cast将其转换为string&&。
这也告诉了我们，可以用static_cast显式地将一个左值转换为一个右值引用

20. 如果一个函数参数是指向模板类型参数的右值引用，则它对应的实参的const属性和左值/右值属性将得到保持。
21. forward函数（定义在头文件utility中）用于在函数参数传递过程中保持原始实参的类型。**forward必须通过显式模板实参调用，其返回的是该显式实参类型的右值引用**，即forward<T>的返回类型是T&&(**此时若T是左值引用 则由于引用折叠，forward的返回值也是左值引用**)。通常我们用forward传递哪些定义为模板类型参数的右值引用的函数参数，通过其返回类型上的**引用折叠**，forward可以保持给定实参的左值/右值属性。

```
template<typename Type>
void intermediary(Typr &&arg){
    finalFcn(std::forward<Type>(arg))
}
```

模板与重载

1. 对于一个调用，如果一个非函数模板与一个函数模板提供同样好的匹配，则选择非模板版本
2. 考虑调用debug_rep("hi world!")，有如下三种debug_rep的版本可以匹配：
   - debug_rep(const T&),T被绑定到char[10]
   - debug_rep(T*),T被绑定到const char
   - debug_rep(const string&),要求从const char*到string的类型转换

   两个模板都提供了精确匹配，其中第二个模板需要进行一次数组到指针的转换，这种转换被认为是精确匹配。而非模板版本需要进行一次用户定义的类型转换。所以T*更加特例化，故编译器最终会选择第二个版本调用

可变参数模板

1. 可变参数模板就是一个接受可变数目参数的模板函数或模板类。可变数目的参数被称为参数包。存在两种参数包：模板参数包，表示零个或多个模板参数；函数参数包，表示零个或多个函数参数。**用一个省略号(...)来指出一个模板参数或函数参数表示一个包**。在一个模板参数列表中，**calss...或者typename**指出接下来的参数表示零个或多个类型的列表。在函数参数列表中，如果一个参数的类型是一个模板参数包，则此参数也是一个函数参数包。

```
//Args是一个模板参数包；rest是一个函数参数包
//Args表示零个或多个模板类型参数
//rest表示零个或多个函数参数
template<typename T,typename... Args>
void foo(const T &t,const Args&... rest);

//给定下面的调用
int i=0; double d=3.14; string s="hello";
foo(i,s,42,d);  //正确 表明Args中有三个参数
foo(s,422,"hi");//正确 表明Args中有两个参数
foo(d,s);       //正确 表明Args中有一个参数
foo("hi");      //正确 表明Args中没有参数

//要知道包中有多少元素，可以使用sizeof...运算符
template<typename... Args>
void g(Args... args){
    cout<<sizeof...(Args)<<endl;
    cout<<sizeof...(args)<<endl;
}
```

2. 对可变参数的递归调用：

```
template<typename T,typename... Args>
ostream& print(ostream &os,const T &t,const Args&... rest){
    os<<t<<", ";    //打印第一个实参
    return print(os,rest...);   //递归调用 打印其他实参
}
template<typename T>
ostream& print(ostream &os,const T &t){
    return os<<t;
}
```

可变参数函数通常是递归的，第一步调用处理包中的第一个实参，然后剩余实参调用自身。上面的print就是这样，每次递归调用将第二个实参打印到第一个实参表示的流中。为了终止递归，还定义了一个非可变参数的print函数，负责打印初始调用中的最后一个实参。
print(os,rest...)中rest的第一个参数被绑定为T，剩余参数形成下一个print调用的参数包。如调用print(cout,i,s,42)时，递归执行如下：

- print(cout,i,s,42)： t=i， rest...=s,42
- print(cout,s,42)： t=s， rest...=42
- print(cout,42)： 调用非可变参数版本的print 终止递归

3. 对于一个参数包，除了获取其大小之外，还可以对它做扩展。扩展一个包就是将它分解为构成的元素，对每个元素应用模式，获得扩展后的列表。通过在模式右边放一个省略号(...)来触发扩展操作。

```
template <typename... Args>
void func(const Args& rest);     //错误 参数包为扩展
void func(const Args&... rest); //正确
```

4. 理解包扩展：**扩展结果将是一个逗号分隔的调用列表**。

```
//其中print1、print2和f是其他函数
template<typename... Args>
void func(ostream &os,const Args&... rest){
    print1(os,f(rest)...);
    print2(os,f(rest...));
}
//若有下面调用：
func(cout,i,j,k);
//则等价于：
print1(os,f(i),f(j),f(k));
print2(os,f(i,j,k));
```

5. 可变参数函数通常和forward配合使用来传递参数

模板特例化

1. 模板特例化是模板的一个独立的定义，在其中一个或多个模板参数被指定为特定的类型。当特例化一个函数模板时，必须为原模板中的每个模板参数都提供实参，且为了指出我们正在实例化一个模板，应该使用关键字**template<>**。一个特例化版本本质上是一个实例，而非函数名的一个重载版本。