@阿西拜-南昌

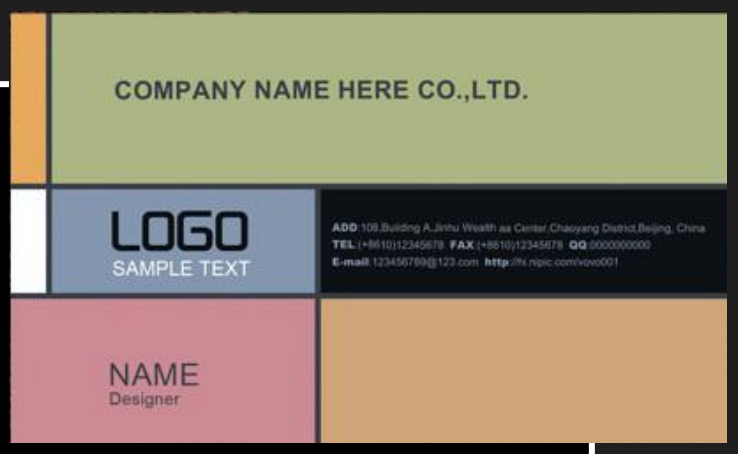
模板与泛型编程

C++ Primer第五版

第16章

模板是C++泛型编程的基础。
为模板提供足够的信息，就能生成特定的类或函数。

定义模板




```
int compare(const string &v1, const string &v2)
{
    if( v1<v2 ) return -1;
    if( v2<v1 ) return 1;
    return 0;
}
int compare(const double &v1, const double &v2)
{
    if( v1<v2 ) return -1;
    if( v2<v1 ) return 1;
    return 0;
}
```

没有办法为用户准备自定义类版本的函数

可以定义一个通用的函数模板，而不是为每个类型都定义一个新函数

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if(v1 < v2) return -1;
    if(v2 < v1) return 1;
    return 0;
}
```

模板参数列表 可以有多个
模板参数列表不能为空
关键字 ： typename 和 class 都是可以的

T的实际类型在编译时根据compare的使用情况来确定

实例化函数模板

```
//实例化出int compare(const int&, const int&)
cout << compare(1,0) << endl; //T为int

//实例化出int compare(const vector<int>&, const vector<int>&)
vector<int> vec1{1,2,3}, vec2{4,5,6};
cout<<compare(vec1,vec2)<<endl; //T为vector<int>
```

编译器生成的版本通常被称为模板的实例

类型参数可以用来指定返回类型或函数的参数类型

```
template<typename T> T foo(T* p)
{
      T tmp = *p; //tmp的类型将是指针p指向的类型
      // ...
      return tmp;
}

//错误：U之前必须加上class或typename
template<typename T,U> T calc(const T&, const U&);
//正确：在目标参数列表中，typename和class没什么不同
template<typename T,class U> calc(const T&, const U&);
```

可以在模板中定义非类型参数，表示一个值而非一个类型

```
template<unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)[M])
{
      return strcmp(p1,p2);        非类型模板参数的模板实参必须是常量表达式
}

compare("hi","mom");       字符串末尾 /0
//int compare(const char (&p)[3],const char (&p2)[4]
```

inline和constexpr的函数模板

```
//正确：inline说明符跟在模板参数列表之后                    注意位置
template <typename T> inline T min(const T&, const T&);
//错误：inline说明符的位置不对
inline template <typename T> T min(const T&, const T&);
```

编写类型无关的代码

```
template <typename T> int compare(const T &v1, const T &v2)
{                    没必要有> 就定义 <
      if(v1 < v2) return -1;
      if(v2 < v1) return 1;
      return 0;
}
//即使用于指针也正确的版本
template <typename T> int compare(const T &v1, const T &v2)
{
      if( less<T>()(v1,v2 )) return -1;
      if( less<T>()(v1,v2 )) return 1;
      return 0;
}
```

模板程序应该尽量减少对实参类型的要求

编写泛型模板的俩个重要原则:
函数参数是 const 引用的
只定义一直 关系操作

函数模板 和 类模板成员的定义通常放在头文件中

类模板

```
template <typename T> class Blob {
public:
    typedef T value_type;
    typedef typename std::vector<T>::size_type size_type;
    //构造函数
    Blob();
    Blob(std::initializer_list<T> il);
    //Blob中的元素数目
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    //添加和删除元素
    void push_back(const T &t) {data->push_back(t);}
    //移动版本
    void push_back(T &&t) { data->push_back(std::move(t)); }
    void pop_back();
    //元素访问
    T& back();
    T& operator[] (size_type i);
private:
    std::shared_ptr<std::vector<T>> data;
    //若data[i]无效，则抛出msg
    void check(size_type i, const std::string &msg) const;
};
```

实例化类模板

```
Blob<int> ia; //空Blob<int>
Blob<int> ia2; = {0,1,2,3,4}; //有5个元素的Blob<int>

//编译器会实例化出一个与下面定义等价的类
template<> class Blob<int> {
    typedef typename std::vector<int>::size_type size_type;
    Blob();
    Blob(std::initializer_list<int> il);
    //...
    int &operator[](size_type i);
private:
    std::shared_ptr<std::vector<int>> data;
    void check(size_type i, const std::string &msg) const;
}
```

类模板的成员函数

```
template <typename T>
void Blob<T>::check(size_type i, const std::string &msg) const
{
      if( i>= data->size())
            throw std::out_of_range(msg);
}

template <typename T>
T& Blob<T>::back()
{
      check(0,"back on empty Blob");
      return data->back();
}
template <typename T>
T& Blob<T>::operator[](size_type i)
{
      //如果i太大，check会抛出异常，阻止访问一个不存在的元素
      check(i,"subscript out of range");
      return (*data)[i];
}
template <typename T> void Blob<T>::pop_back()
{
      check(0,"pop_back on empty Blob");
      data->pop_back();
}
```

Blob构造函数

```
template <typename T>
Blob<T>::Blob():data(std::make_shared<std::vector<T>>()) { }

template <typename T>
Blob<T>::Blob(std::initializer_list<T> il):
      data (std::make_shared<std::vector<T>>(il)) { }

//Blob<string> articles = {"a", "an", "the"};
```

类模板成员函数的实例化

```
//实例化Blob<int>和接受initializer_list<int>的构造函数
Blob<int> squares = {0,1,2,3,4,5,6,7,8,9};
//实例化Blob<int>::size() const
for(size_t i = 0; i!= squares.size(); ++i)
    squares[i] = i*i; //实例化Blob<int>::operator[](size_t)
//如果一个成员函数没被使用，则它不会被实例化
```

默认情况下，对于一个实例化了的类模板，其成员只有在使用时才被实例化

在类代码内简化模板类名的使用

```
//若试图访问一个不存在的元素，BlobPtr抛出一个异常
template <typename T> class BlobPtr {
public:
    BlobPtr():curr(0) { }
    BlobPtr(Blob<T> &a, size_t sz=0):
        wptr(a.data), curr(sz) { }
    T& operator*() const
    {
        auto p = check(curr, "dereference past end");
        return (*p)[curr]; //(*p)为本对象指向的vector
    }
    //递增和递减
    BlobPtr& operator++();//前置运算符
    BlobPtr& operator--();
private:
    //若检查成功，check返回一个指向vecotr的shared_ptr
    std::shared_ptr<std::vector<T>>
        check(std::size_t, const std::string&) const;
    //保存一个weak_ptr，表示底层vector可能被销毁
    std::weak_ptr<std::vector<T>> wptr;
    std::size_t curr; //数组中的当前位置
};
```

```
BlobPtr<T>& operator++();
BlobPtr<T>& operator--();
```

在类模板外使用类模板名

```
//后置：递增/递减对象但返回原值
template <typename T>
BlobPtr<T> BlobPtr<T>::operator++(int)
{
    //此处无须检查；调用前置递增时会进行检查
    BlobPtr ret = *this; //保存当前值
    ++*this;  //推进一个元素；前置++检查递增是否合法
    return ret; //返回保存的状态
}
```

在一个类模板的作用域内，我们可以直接使用模板名而不必指定模板实参。

类模板和友元：一对一友好关系

```
//前置声明，在Blob中声明友元所需要的
template <typename> class BlobPtr;
template <typename> class Blob; //运算符==中的参数所需要的
template <typename T>
bool operator==(const Blob<T>&, const Blob<T>&);

template <typename T> class Blob {
    //每个Blob实例将访问权限授予相同类型实例化的BlobPtr和相等运算符
    friend class BlobPtr<T>;
    friend bool operator==<T>(const Blob<T>&, const Blob<T>&);
    //...
};

Blob<char> ca;    //BlobPtr<char>和operator==<char>都是本对象的友元
Blob<int> ia;     //BlobPtr<int>和operator==<int>都是本对象的友元
```

通用和特定的模板友好关系

```
//前置声明，在将目标的一个特定实例声明为友元时要用到
template<typename T> class Pal;
class C { //C是一个普通的非模板类
    friend class Pal<C>; //用类C实例化的Pal是C的一个友元
    //Pal2的所有实例都是C的友元；这种情况无需前置声明
    template <typename T> friend class Pal2;
};
template <typename T> class C2 { //C2本身是一个类模板
    //C2的每个实例将相同实例化的Pal声明为友元
    friend class Pal<T>; //Pal的模板声明必须在作用域之内
    //Pal2的所有实例都是C2的每个实例的友元，不需要前置声明
    template <typename X> friend class Pal2;
    //Pal3是一个非模板类，它是C2 所有实例的友元
    friend class Pal3; //不需要Pal3的前置声明
};
```

为了让所有实例成为友元，友元声明中必须使用与模板本身不同的模板参数

令模板自己的类型参数成为友元

```
template <typename Type> class Bar {
    friend  Type;  //将访问权限授予用来实例化Bar的类型
    //...
};
```

模板类型别名

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

twin<int> win_loss; //win_loss是一个pair<int,int>
twin<double> area; //

//可以固定一个或多个模板参数
template <typename T> using partNo = pair<T, unsigned>;
partNo<string> book; //pair<string, unsigned>
partNo<Vehicle> cars;
partNo<Student> kids;
```

类模板的static成员

```
template <typename T> class Foo {
public:
    static std::size_t count() { return ctr; }
    //其他接口成员
private:
    static std::size_t ctr;
    //其他实现成员
};

//实例化static成员Foo<string>::ctr和Foo<string>::count
Foo<string> fs;
//所有三个对象共享相同的Foo<int>::ctr和Foo<int>::count成员
Foo<int> fi,fi2,fi3;

//可以将static数据成员定义成模板
template <typename T>
size_t Foo<T>::ctr = 0; //定义并初始化ctr
```

通过类来访问static成员，必须引用一个特定的实例

```
Foo<int> fi; //实例化Foo<int>类和static数据成员ctr
auto ct = Foo<int>::count(); //实例化Foo<int>::count
ct = fi.count(); //使用Foo<int>::count
ct = Foo::count(); //错误：使用哪个版本实例的count？
```

模板参数

```
typedef double A;
template <typename A, typename B> void f(A a, B b)
{
	A tmp = a;	//tmp的类型为A的类型，而非double
	double B;	//错误：重声明模板参数B
}

//错误：非法重用模板参数名V
template <typename V, typename V> //...
```

与函数参数相同，声明中的模板参数的名字不必与定义中相同

```
//3个calc都指向相同的函数模板
template <typename T> T calc(const T&, const T&); //声明
template <typename U> U calc(const U&, const U&); //声明
//模板的定义
template <typename Type>
Type calc(const Type& a, const Type& b) { /*...*/ }
```

使用类的类型成员

```
template <typename T>
typename T::value_type top(const T& c)
{
	if(!c.empty())
		return c.back();
	else
		return typename T::value_type();
}
```

默认模板实参

```
//compare有一个默认模板实参less<T>和一个默认函数实参F()
template<typename T, typename F=less<T>>
int compare(const T &v1, const T v2, F f = F())
{
	if (f(v1,v2)) return -1;
	if (f(v2,v1)) return 1;
	return 0;
}

bool i = compare(0,42); //使用less；i为-1
//结果依赖于item1和item2中的isbn
Sales_data item1(cin),item2(cin);
bool j = compare(item1,item2,compareIsbn);
```

模板默认实参与类模板

```
template <class T = int> class Numbers {//T默认为int
public:
	Numbers(T v=0):val(v) { }
	//对数值的各种操作
private:
	T val;
};
Numbers<long double> lots_of_precision;
NUmbers<> average_precision; //空<>表示我们希望使用默认类型
```

一个类可以包含本身是模板的成员函数：成员模板（不能是虚函数）
普通（非模板）类的成员模板

```
//函数对象类，对给定指针执行delete
class DebugDelete {
public:
	DebugDelete(std::ostream &s = std::cerr): os(s) { }
	//与任何函数模板相同，T的类型由编译器推断
	template<typename T> void operator()(T* p) const
		{ os << "deleting unique_ptr" << std::endl; delete p; }
private:
	std::ostream &os;
};

double *p = new double;
DebugDelete d;   //可像delete表达式一样使用的对象
d(p);                  //调用DebugDelete::operator()(double*),释放p
int* ip = new int;
//在一个临时DebugDelete对象上调用operator()(int*)
DebugDelete()(ip);

//销毁p指向的对象
//实例化DebugDelete::operator()<int>(int *)
unique_ptr<int,DebugDelete> p(new int, DebugDelete());
//销毁sp指向的对象
//实例化DebugDelete::operator()<string>(string*)
unique_ptr<string,DebugDelete> sp(new string, DebugDelete());
```

```
//DebugDelete的成员模板实例化样例
void DebugDelete::operator()(int *p) const { delete p; }
void DebugDelete::operator()(string *p) const { delete p; }
```

类模板的成员模板

```
template <typename T> class Blob {
        template <typename It> Blob(It b, It e);
        //...
};
template <typename T>    //类的类型参数
template <typename It>    //构造函数的类型参数
        Blob<T>::Blob(It b, It e):
                data(std::make_shared<std::vector<T>>(b,e)){ }
```

实例化与成员模板

```
int ia[] = {0,1,2,3,4,5,6,7,8,9};
vector<long> vi = {0,1,2,3,4,5,6,7,8,9};
list<const char*> w = {"now", "is", "the", "time"};
//实例化Blob<int>类及其接受两个int*参数的构造函数
Blob<int> a1(begin(ia),end(ia));
//实例化Blob<int>类接受两个vector<long>::iterator的构造函数
Blob<int> a2(vi.begin(),vi.end());
//实例化Blob<string>及其接受两个list<cosnt char*>::iterator参数的构造函数
Blob<string> a3(w.begin(),w.end());
```

通过显示实例化避免相同的实例出现在多个对象文件中

```
//实例化声明与定义
extern template class Blob<string>; //声明
template int compare(const int&, const int&); //定义

//Application.cc
//这些模板类型必须在程序与其他位置进行实例化
extern template class Blob<string>;
extern template int compare(const int&, const int&);
Blob<string> sa1,sa2; //实例化会出现在其他位置

//Blob<int>及其接受initializer_list的构造函数在本文件中实例化
Blob<int> a1 = {0,1,2,3,4,5,6,7,8,9};
Blob<int> a2(a1); //拷贝构造函数在本文中实例化
int i = compare(a1[0],a2[0])); //实例化出现在其他位置

//templateBuild.cc
template int compare(const int&, const int&);
template class Blob<string>;
```

效率与灵活性

```
//shared_ptr的析构函数必须包含类似下面这样的语句
del?del(p):delete p; //del是一个成员，运行时需要跳转到del的地址

//unique_ptr的析构函数与shared_ptr类似
del(p);  //在编译时del以确定类型，无运行时额外开销
```

模板实参推断

编译器通常不是对实参进行类型转换，而是生成一个新的模板实例

```
template <typename T> T fobj(T,T);                    //实参被拷贝
template <typename T> T fref(const T&, const T&);     //引用
string s1("a value");
const string s2("another value");
fobj(s1,s2); //调用fobj(string,string); const被忽略
fref(s1,s2); //调用fref(const string&, const string&)
//将s1转换为const试允许的
int a[10], b[42];
fobj(a,b); //调用f(int*,int*)
fref(a,b);  //错误：数组类型不匹配
```

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转换及数组或函数到指针的转换

使用相同模板参数类型的函数形参

```
//假设compare函数接受两个const T&参数
long lng;
compare(lng,1024); //错误：不能实例化compare(long,int)

//实参类型可以不同，但必须兼容
template <typename A,typename B>
int flexibleCompare(const A& v1, const B& v2)
{
        if(v1<v2) return -1;
        if(v2<v1) return 1;
        return 0;
}

long lng;
flexibleCompare(lng,1024); //正确
```

正常类型转换应用于普通函数实参

```
template <typename T> ostream &print(ostream &os, const T &obj)
{
        return os << obj;
}

print(cout,42); //实例化print(ostream&, int)
ofstream f("output");
print(f,10); //使用print(ostream&,int);将f转换为ostream&
```

如果函数参数类型不是模板参数，则对实参进行正常的类型转换

指定显式模板实参

```
//编译器无法推断T1,它未出现在函数参数列表中
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
auto val3 = sum<long long>(i,lng); //long long sum(int,long)

//糟糕的设计：用户必须指定所有三个模板参数
template<typename T1,typename T2,typename T3>
T3 alternative_sum(T2,T1);

//错误
auto val3 = alternative_sum<long long>(i,lng);
//正确：显式指定了所有三个参数
auto val2 = alternative_sum<long long, int , long>(i,lng);
```

按由左至右的顺序与对应的模板参数匹配

正常类型转换应用于显式指定的实参

```
long lng;
compare(lng,1024);              //错误：模板参数不匹配
compare<long>(lng,1024);        //正确：实例化compare(long,long)
compare<int>(lng,1024);         //正确：实例化compare(int,int)
```

尾置返回类型与类型转换

```
template <typename It>
??? &fcn(It beg, It end)
{
    //处理序列
    return *beg; //返回序列中一个元素的引用
}

vector<int> vi = { 1,2,3,4,5};
Blob<string> ca = {"hi","bye"};
auto &i = fcn(vi.begin(), vi.end());    //fcn应该返回int&
auto &s = fnc(ca.begin(), ca.end()); //fcn应该返回string&

//尾置返回允许我们在参数列表之后声明返回类型
template <typename It>
auto fcn(It beg,It end) -> decltype(*beg)
{
    //处理序列
    return *beg;//返回序列中一个元素的引用
}
```

在编译器遇到函数的参数列表之前，beg并不存在

进行类型转换的标准库模板类

```
//为了使用模板参数的成员，必须用typename
template<typename It>
auto fcn2(It beg, It end)->typename remove_reference<decltype(*beg)>::type
{
    //处理序列
    return *beg; //返回序列中一个元素的拷贝
}
```

标准类型转换模板

对 Mod<T>，其中 Mod 为	若 T 为	则 Mod<T>::type 为
remove_reference	X&或 X&&	X
	否则	T
add_const	X&、const X 或函数	T
	否则	const T
add_lvalue_reference	X&	T
	X&&	X&
	否则	T&
add_rvalue_reference	X&或 X&&	T
	否则	T&&
remove_pointer	X*	X
	否则	T
add_pointer	X&或 X&&	X*
	否则	T*
make_signed	unsigned X	X
	否则	T
make_unsigned	带符号类型	unsigned X
	否则	T
remove_extent	X[n]	X
	否则	T
remove_all_extents	X[n1][n2]…	X
	否则	T

函数指针和实参推断

```
template<typename T> int compare(const T&, const T&);
//fp1指向实例int compare(const int&, const int&)
int (*pf1)(const int&,const int&) = compare;

//func的重载版本；每个版本接受一个不同的函数指针类型
void func(int(*)(const string&, const string&));
void func(int(*)(const int&, const int&));
func(compare); //错误：使用compare的哪个实例？

//正确：显式指出实例化哪个compare版本
func(compare<int>); //传递compare(const int&,const int&)
```

编译器使用指针的类型来推断模板实参

从左值引用函数参数推断类型

```
template<typename T> void f1(T&); //实参必须是一个左值
//对f1的调用使用实参所引用的类型作为模板的参数类型
f1(i);      //i是一个int；模板参数类型T是int
f1(ci);     //ci是一个const int;模板参数T是const int
f1(5);      //错误：传递给一个&参数的实参必须是一个左值

template<typename T> void f2(const T&); //可以接受一个右值
//f2中的参数是const &;实参中的const是无关的
//在每个调用中，f2的函数参数都被推断为const int&
f2(i);      //i使一个int；模板参数T是int
f2(ci);     //ci是一个const int，但模板参数T是int
f2(5);      //一个const &参数可以绑定到一个右值；T是int
```

底层const

从右值引用函数参数推断类型

```
template <typename T> void f3(T&&);
f3(42);  //实参是一个int类型的右值；模板参数T是int
```

引用折叠和右值引用参数：通常不能将一个右值引用绑定到一个左值上

左值传递给函数的右值引用参数（模板类型参数），编译器推断模板类型参数为实参的左值应用类型

- X& & 、 X& &&和X&& &都折叠成类型X&
- 类型X&& &&折叠成X&&

引用折叠只能应用于间接创建的引用的引用，如类型别名或模板参数

```
f3(i);      //实参是一个左值；模板参数T是int&
f3(ci);     //实参是一个左值；模板参数T是一个const int&
```

如果将一个左值传递给 右值的函数参数 那么它会被实例化为 左值的类型

```
//无效代码，只用于演示
void f3<int&>(int& &&);    //当T是int&时，函数参数为int& &&
void  f3<int&>(int&);      //当T是int&时，函数参数折叠为int&
```

编写接受右值引用参数的模板函数

```
template <typename T> void f3(T&& val)
{
    T t = val;      //拷贝还是绑定一个引用？
    t = fcn(t);     //赋值只改变t还是即改变t又改变val?
    if(val == t){/*...*/} //若T是引用类型，则一直为true
}
```

std::move是如何定义的

move告诉编译器：我们有一个左值，但我们希望像一个右值一样处理它

```
//标准库是这样定义move的
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
{
    return static_cast<typename remove_reference<T>::type&&>(t);
}
```

从一个左值satic_cast到一个右值引用是允许的

```
string s1("hi"),s2;
s2 = std::move(string("bye!"));      //正确：从一个右值移动数据
//string&& move(string &&t)

s2 = std::move(s1);                  //正确：但在赋值之后，s1的值不确定
//string&& move(string &t)
```

- 推断出T的类型为string&
- 因此，remove_reference用string&进行实例化
- remove_reference<string&>的type成员是string
- move的返回类型仍是string&&
- move的函数参数t实例化为string& &&,会折叠为string&

一个左值static_cast 到一个右值引用是允许的。

转发

```
//flip1是一个不完整的实现：顶层const和引用丢失了
template<typename F, typename T1, typename T2>
void flip1(F f, T1 t1, T2 t2)
{
	f(t2,t1);
}

void f(int v1, int &v2) //注意v2是一个引用
{
	cout<<v1<<" "<<++v2<<endl;
}

f(42,i);		//f改变了实参i
flip1(f,j,42);		//f不会改变j  t2 被实例化为  int  没有引用 所以不会修改j的值
//实例化：
//void flip1(void(*fcn)(int, int&), int t1, int t2);
```

定义能保持类型信息的函数参数

```
template<typename F, typename T1, typename T2>
void flip2(F f, T1 &&t1, T2 &&t2)
{
	f(t2,t1);
}

void g(int &&i, int& j)
{
	cout<<i<<" "<<j<<endl;
}
flip1(f,j,42); //f会改变j，void flip1(void(*fcn)(int, int&), int& t1, int&& t2);
flip2(g,i,42); //错误：不能从一个左值实例化int&&
```

右值表达式是左值

在调用中使用std::forward保持类型信息

```
template <typename F, typename T1, typename T2>
void flip(F f, T1 &&t1, T2 &&t2)
{
	f(std::forward<T2>(t2),std::forward<T1>(t1));
}
flip(g,i,42);  //i将以int&类型传递给g,42将以int&&类型传递给g
```

forward<T>的返回类型是T&& 返回该实参的右值引用

当用于一个指向目标参数类型的右值引用函数参数(T&&)时，forward会保持实参类型的所有细节

函数模板可以被另一个模板或一个普通非模板函数重载

```
#include <string>
#include <iostream>
#include <sstream>
using namespace std;

//打印任何我们不能处理的类型
template<typename T> string debug_rep(const T &t)
{
	ostringstream ret;
	ret << t; //使用T的输出运算符打印t的一个表示形式
	return ret.str();
}

//打印指针的值，后跟指针指向的对象
//注意：此函数不能用于char*;
template<typename T> string debug_rep(T *p)
{
	ostringstream ret;
	ret << "pointer: " << p;		//打印指针本身的值
	if(p)
		ret <<" "<<debug_rep(*p); //打印p指向的值
	else
		ret << " null pointer";		//或指出p为空
	return ret.str();				//返回ret绑定的string的一个副本
}

int main()
{
	string s("hi");
	cout<<debug_rep(s)<<endl ;	//只能匹配第一个版本
	cout<<debug_rep(&s)<<endl;	//第二个版本更精确
	const string *sp = &s;
	cout<<debug_rep(sp)<< endl; //都是精确匹配，但第二个版本更特别
}
```

```
C:\WINDOWS\system32\cmd.exe
hi
pointer: 008FFA9C hi
pointer: 008FFA9C hi
```

非模板和模板重载

```
//非模板函数
string debug_rep(const string &s)
{
	return ""+s+"“;
}

string s("hi");
cout<<debug_rep(s)<<endl;
```

非模板版本优先

重载模板和类型转换

```
cout<<debug_rep("hi world!")<<endl; //debug_rep(T*)
//有三个可行的版本
//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的转换
```

如果希望将字符指针按string处理：

```
//将字符指针按指针转换为string，并调用string版本的debug_reg
string debug_rep(char* p)
{
	return debug_rep(string(p));
}
string debug_rep(const char *p)
{
	return debug_rep(string(p));
}
```

缺少声明可能导致程序行为异常

```
template <typename T> string debug_rep(const T &t);
template <typename T> string debug_rep(T *p);
//为了使debug_rep(char*)的定义正确工作，下面的声明必须在作用域中
string debug_rep(const string&);
string debug_rep(char *p)
{
	//如果接受一个const string&的版本的声明不在作用域中，
	//返回语句将调用debug_rep(const T&)的T实例化为string的版本
	return debug_rep(string(p));
}
```

可变参数模板

接受可变数目参数的模板函数或模板类

```
//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);

//编译器会推断包中参数的数目
int i = 0; double d = 3.14; string s = "how now brown cow";
foo(i,s,42,d);          //包中有三个参数
foo(s,42,"hi");         //包中有两个参数
foo(d,s);               //包中有一个参数
foo("hi");              //空包

//编译器会为foo实例化出四个不同的版本
void foo(const int&, const string&, const int&, const double&);
void foo(const string&, const int&, const char[3]&);
void foo(const double&, const string&);
void foo(const char[3]&);
```

sizeof...运算符

```
template<typename ...Args> void g(Args ...args){
    cout<<sizeof...(Args)<<endl; //类型参数的数目
    cout<<sizeof...(args)<<endl; //函数参数的数目
}
```

编写可变参数函数模板

```
//用来终止递归并打印最后一个元素的函数
//此函数必须在可变参数版本的print定义之前声明
tempalte<typename T>
ostream &print(ostream&os,const T &t)
{
    return os << t; //包中最有一个元素之后不打印分隔符
}
//包中除了最后一个元素之外的其他元素都会调用这个版本的print
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
}
print(cout, i, s, 42); //包中有两个参数
```

包扩展：分解为构成的元素，对每个元素应用模式

```
template<typename T, typename... Args>
ostream &
print(ostream &os, const T &t, const Args&... rest) //扩展Args
{
      os << t << ", ";
      return print(os, rest...); //扩展rest
}

print(cout,i,s,42); //包中有两个参数
//此调用被实例化为
//osgream&
//print(ostream&, cosnt int&, const string&, const int&);
//print(os,s,42);
```

理解包扩展

```
//在print调用中对每个实参调用debug_rep
template<typename... Args>
ostream &errorMsg(ostream &os, const Args&... rest)
{
      //print(os, debug_rep(a1),debug_rep(a2),...,debug_rep(an)
      return print(os,debug_rep(rest)...);
}

errorMsg(cerr,fncName,code,num(),otherData,"other",item);
//等价于
//print(cerr,debug_rep(fcnName),debug_rep(code,num()),
//          debug_rep(otherData),debug_rep("otherData"),
//          debug_rep(item));

print(os,debug_rep(rest...)); //错误：此调用无匹配函数
print(cerr,debug_rep(fcnName,code.num(),otherData,"otherData",item));
```

转发参数包：组合使用可变参数模板与forward机制

```
class StrVec{
public:
    template<class... Args> void emplace_back(Args&&...);
    //...
};

template<class... Args>
inline
void StrVec::emplace_back(Args&&... args)
{
    chk_n_alloc(); //如果需要的话重新分配StrVec内存空间
    alloc.construct(first_free++,std::forward<Args>(args)...);
}

//假定svec是一个StrVec的对象
svec.emplace_back(10,'c'); //将ccccccccc添加为新的尾元素
//std::forward<int>(10),std::forward<char>(c)
svec.emplace_back(s1+s2); //使用移动构造函数
//std::forward<string>(string("the end"))
```

模板特例化

当我们不能（或不希望）使用模板版本时，可以定义一个特例化版本

```
//第一个版本；可以比较任意两个类型
template<typename T> int compare(const T&, const T&);
//第二个版本；处理字符串字面常量
template<size_t N, size_t M>
int compare(const char (&)[N],const char (&)[M]);

const char *p1 = "hi", *p2 = "mom";
//无法将一个指针转换为数组的引用
compare(p1,p2);              //调用第一个模板
compare("hi","mom");         //调用有两个非类型参数的版本
```

定义函数模板特例化

```
//compare的特殊版本，处理字符数组的指针
template<>
int compare(const char* const &p1, const char* const &p2)
{
      return strcmp(p1,p2);
}
```

函数重载与模板特例化

```
compare("hi","mom");
//由于是特例化版本，并不是独立的非模板函数
//所以还是调用数组引用的版本
```

类模板特例化:将自定义类放到无序容器


```
//打开std命名空间，以便特例化std::hash
namespace std {
    template<>//我们正在定义一个特例化版本，模板参数为Sales_data
    struct hash<Sales_data>
    {
        //用来散列一个无需容器的类型必须要定义下列类型
        typedef size_t result_type;
        typedef Sales_data argument_type; //默认情况下，此类型需要==
        size_t oeprator()(const Sales_data& s) const;
        //我们的类使用合成的口碑控制成员和默认构造函数
    };
    size_t hash<Sales_data>::operator()(const Sales_data& s) const
    {
        return hash<string>()(s.bookNo)^
        hash<unsigned>()(s.units_sold)^
        hash<double>()(s.revenue);
    }
}//关闭std命名空间；注意：右花括号之后没有分号

//使用hash<Sales_data>和Sales_data中的operator==
unordered_multiset<Sales_data> SDset;

template<class T> class std::hash; //友元声明所需要的
class Sales_data{
    friend class std::hash<Sales_data>;
    //其他成员定义...
} ;
```

类模板部分特例化

```
//原始的、最通用的版本
template<class T> struct remove_reference{
    typedef T type;
};
//部分特例化版本，将用于左值引用和右值引用
template<class T> struct remove_reference<T&> //左值引用
    { typedef T type; };
template<class T> struct remove_reference<T&&> //右值引用
    { typedef T type; };

int i;
remove_reference<decltype(42)>::type a;
remove_reference<decltype(i)>::type b;
remove_reference<decltype(std::move(i))>::type c;
```

只能部分特例化类模板，而不能部分特例化函数模板

特例化成员而不是类


```
template<typename T> struct Foo{
        Foo(const T &t = T()):mem(t) { }
        void Bar() {/*...*/}
        T mem;
        //Foo的其他成员
};
template<>                      //我们正在特例化一个模板
void Foo<int>::Bar()            //我们正在特例化Foo<int>的成员Bar
{
        //进行应用于int的特例化处理
}
//我们只特例化Foo<int>类的一个成员，其他成员将由Foo模板提供
Foo<string> fs;     //实例化Foo<string>::Foo()
fs.Bar();           //实例化Foo<string>::Bar()
Foo<int> fi;        //实例化Foo<int>::Foo()
fi.Bar();           //使用我们特例化版本的Foo<int>::Bar()
```