Variadic Template

可变参数模板（variadic template）就是接受可变数目的模板函数或模板类。其中可变数目的参数被称为**参数包（parameter packtet）**。

- 模板参数包，表示0个或多个模板参数
- 函数参数包，表示0个或多个函数参数

和C一样，我们用省略号指出一个模板参数或函数参数表示一个包（可变数目的参数）。在一个模板参数列表中，`class...`或`typename...`指出接下来的参数表示零个或多个**类型**的列表；一个类型后面跟着一个省略号表示零个或多个给定类型的**非类型参数**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。例如，

```
// Args是一个模板参数包；rest是一个函数参数包
// Args表示0个或多个模板类型参数
// rest表示0个或多个函数参数
template<typename T,typename ... Args>
void foo(const T &t,const Args&... rest);
```

声明了一个`foo`，是一个可变参数函数模板，它有一个`T`的模板类型参数，和一个名为`Args`的模板参数包。这个包表示0个或多个额外的类型参数。`foo`的函数参数列表包含一个`const &`类型的参数，指向`T`的类型，还包含一个名`rest`的函数参数包，此包表示0个或多个函数参数。

对于一个可变参数模板，编译器会推断包中参数的数目，例如，给定下面的调用：

```
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]&);
```

其中，`T`从第一个实参的类型推断出来，其余的（如果有的话），提供函数额外实参的数目和类型。

Operator `sizeof...`

我们可以通过`sizeof...`运算符，返回包中的元素个数。与`sizeof`一样，`sizeof...`返回也是一个常量表达式。

```
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
}
```

Writing a Variadic Function Template

可变参数函数通常是递归的。我们先给出C++ primer中的例子。

```
// 必须先定义一个递归终止的函数,也就是包中的元素为0时的情况
// 此函数必须在可变参数版本的print之前声明
template <typename T>
ostream &print(ostream &os,const T &t){
    return os<<t<<; // 包中最后一个元素之后不打印分隔符
}
// 包中除了最后一个元素之外(此时可变参数包没有元素)。
template<typename T,typename ... Args>
ostream &print(ostream &os,const T &t,const Args &... rest){
    os<<t<<",";
    return print(os,rest...);// 递归调用
}
```

这段程序的关键部分时可变参数函数中对print的调用:

```
return print(os,rest...);// 递归调用,打印其他实参
```

我们注意到可变参数版本的print接受三个参数,而此调用只传递了两个实参。结果就是rest中的第一个实参被绑定到了t,剩余的实参形成下一个包。

这里给出调用

```
print(cout,i,s,42);//包中有两个参数
```

的调用过程:

调用过程	t	rest...
print(cout,i,s,42)	i	s,42
print(cout,s,42)	s	42
print(cout,42)调用非可变参数版本的print		

Pack Expansion

当扩展一个包时，我们还要提供用于每个扩展元素的**模式（pattern）**。扩展一个包就是将它分解成构成的元素，对每个元素应用模式，获得扩展后的列表。我们通过在模式右边放一个省略号来触发扩展操作。

例如，我们的print函数包含两个扩展

```
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
}
```

对Args的扩展中，编译器将模式const Args &应用到模板参数包Args中的每个元素。因此，此模式的扩展结果时一个以逗号分隔的零个或多个类型的列表，每个类型都形如const type &。例如，print(cout,i,s,42)最后两个实参的类型和模式一起确定了尾置参数的类型。此调用被实例化为：

```
ostream & print(ostream &os,const int &,const string &,const int &);
```

第二个扩展的模式是函数参数包的名字(rest)。此模式扩展出一个由包中元素组成的，逗号分隔的列表。因此，这个调用等价于：

```
print(os,s,42);
```

Understanding Package Expansion

print中的函数参数包扩展仅仅将包扩展为其构成元素，C++语言还允许更复杂的扩展模式。

```
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));
}
```

这里使用了模式debug_rep(rest)。此模式表示我们希望对函数参数包rest中的每个元素调用debug_rep。扩展结果是一个逗号分隔的debug_rep调用列表。即，如下调用

```
errorMsg(cerr,fcnNam,code.num(),otherData,"oter",iter);
```

就好像编写如下代码：

```
print(cerr,debug_rep(fcnName),debug_rep(code.num()),
            debug_rep(otherData),debug_rep("otherData"),
            debug_rep(iter);
);
```

与之相对，下面的模式会编译失败：

```
// 将包传递给debug_rep; print(os,debug_rep(a1,a2,...,an))
print(os,debug_rep(rest...));// error,该调用无匹配函数
```

这端代码的问题在于我们debug_rep调用中扩展了rest，它等价于

```
print(cerr,debug_rep(fcnName,code.num(),
        otherData,"otherData",iter));
```

Forwading Parameter Packs

可变参数模板和forward机制配合编写，一个例子，实现emplace_back成员函数。

为StrVec设计的emplace_back应该需要保持传递的实参的所有类型信息，也就是需要使用forward。

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

模板参数包扩展中的模式是&&，意味着每个函数参数将是一个指向其对应实参的右值引用。

当emplace_back将这些实参传递给construct时，我们必须使用forward来保持实参的原始类型。

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

扩展中，既扩展了模板参数包Args，也扩展了函数参数包args。此模式生成如下形式元素

```
std::forward<Ti> (ti)
```

其中，Ti表示模板参数包中的第i个元素的类型，ti表示函数参数包中第i个元素。例如，调用

```
svec.emplace_back(10,'c');// 将ccccccccccc添加到尾部
```

construct调用的模式会扩展出

```
std::forward<int>(10),std::forward<char> (c)
```

通过forward, 可以保证如果用一个右值调用emplace_back, 则construct也会得到一个右值。例如,

```
svec.emplace_back(s1+s2);// 使用移动构造函数
```

传递的实参是一个右值, 传递给construct的是

```
std::forward<string>(string("the end"));
```

forward<string>的结果类型是string &&, 因此construct将得到一个右值引用的实参。construct会继续将此实参传递给string的移动构造函数来创建新元素。