第16章 模板与泛型编程

16.1 定义模板

16.1.1 函数模板

1. 我们可以定义一个通用的**函数模板（function template）**，而不是为每个类型都定义一个新函数。一个函数模板就是一个公式，可用来生成针对特定类型的函数版本。
2. 模板定义以关键字 `template` 开始，后跟一个**模板参数列表（template parameter list）**，模板参数列表不能为空。

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

3. 编译器用推断出的模板参数来为我们**实例化（instantiate）**一个特定版本的函数。使用模板实参代替对应的模板参数来创建出模板的一个新"实例"。
4. 我们可以将**类型参数（type parameter）**看作类型说明符，就像内置类型或类类型说明符一样使用。
5. 类型参数可以用来指定返回类型或函数的参数类型，以及在函数体内用于变量声明或类型转换：

```
// 类型参数为 T
template <typename T> T foo(T* p)
{
    T tmp = *p;
    // ...
    return tmp;
}
```

6. 除了定义类型参数，还可以在模板中定义**非类型参数（nontype parameter）**。一个非类型参数表示一个值而非一个类型。通过一个**特定的类型名**而非关键字 `class` 或 `typename` 来指定非类型参数。
7. 当一个模板被实例化时，非类型参数被一个用户提供的或编译器推断出的值所代替。这些值必须是常量表达式，从而允许编译器在编译时实例化模板。
8. 一个非类型参数可以是一个整型，或者是一个指向对象或函数类型的指针或（左值）引用。

9. 绑定到非类型整型参数的实参必须是一个常量表达式。绑定到指针或引用的非类型参数的实参必须具有静态的生存期。
10. 不能用一个普通（非 `static` ）局部变量或动态对象作为指针或引用非类型模板参数的实参。指针参数也可以用 `nullptr` 或一个值为0的常量表达式来实例化。

```
template <unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)[M])
{
    return strcmp(p1, p2);
}

// 调用模板：
compare("hi", "mom");

// 模板将会实例化成：
int compare(const char (&p1)[3], const char (&p2)[4])
```

11. 模板程序应该尽量减少对实参类型的要求。
12. 函数模板和类模板成员函数的定义通常放在头文件中。
13. 大多数编译错误在实例化期间报告。
14. 保证传递给模板的实参支持模板所要求的操作，以及这些操作在模板中能正确工作，是调用者的责任。

16.1.2 类模板

1. **类模板（class template）是用来生成类的蓝图的。
2. 与函数模板的不同之处是，编译器不能为类模板推断模板参数类型。为了使用类模板，我们必须在模板名后的尖括号中提供额外信息-用来代替模板参数的模板实参列表。
3. 使用模板时，需要提供显式模板实参（explicit template argument）列表作为额外信息，它们被绑定到模板参数。编译器使用这些模板参数来实例化特定的类。

```
template <typename T> class Blob {
public:
    Blob();
    Blob(std::initializer_list<T> il);
    // ... ...
private:
    std::shared_ptr<std::vector<T>> data;
    // ... ...
}

Blob<int> ia;                // 空Blob<int>
Blob<int> ia2 = {0, 1, 2}    // 有3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> {
public:
    Blob();
    Blob(std::initializer_list<int> il);
    // ... ...
private:
    std::shared_ptr<std::vector<int>> data;
    // ... ...
}
```

4. 一个类模板的每个实例都形成一个独立的类。类型 `Blob<string>` 与任何其他的 `Blob` 类型都没有关联，也不会对其他任何 `Blob` 类型的成员有特殊访问权限。
5. 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。这一特征是的即使某种类型不能完全符合模板操作的要求，我们仍然能用该类型实例化类。
6. 在类模板自己的作用域内，可以直接使用模板名而不提供实参，编译器处理模板自身的引用时就好像我们已经提供了与模板参数匹配的实参一样。
7. 当我们在类模板外定义其他成员时，必须记住，我们并不在类的作用域中，直到遇到类名才表示进入类的作用域。
8. 当一个类包含一个友元声明时，类与友元各自是否是模板是相互无关的。
9. 如果一个类模板包含一个非模板友元，则友元被授权可以访问所有模板实例。如果友元自身是模板，类可以授权给所有友元模板实例，也可以只授权给特定实例。
10. 模板类与另一个（类或函数）模板间友好关系的最常见的形式是建立对应实例及其友元间的友好关系。
11. 一个类也可以将另一个模板的每一个实例都声明为自己的友元，或者限定特定的实例为友元。
12. 在新标准中，可以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
    // ...
}
```

13. 类模板的一个实例定义了一个类类型，与任何其他类类型一样，我们可以定义一个 `typedef` 来引用实例化的类：

```
typedef Blob<string> StrBlob;

// 由于模板不是类型，我们无法定义一个typedef引用一个模板：
typedef Blob<T> TBlob;

// 新标准允许我们为类模板定义一个类型别名：
template <typename T> using twin = pair<T, T>;
twin<string> authors;   // 一个pair<string, string>

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

14. 类模板可以声明 `static` 成员。
15. 类似任何其他的成员函数，一个static成员函数只有在使用时才会实例化。

```
template <typename T> class Foo {
public:
    static std::size_t count() { return ctr; }
private:
    static std::size_t ctr;
};

// 将static数据成员也定义为模板
template <typename T>
size_t Foo<T>::ctr = 0; // 定义并初始化ctr

// 访问静态成员函数：
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：使用哪个模板实例的count?
```

16.1.3 模板参数

1. 一个模板参数名的可用范围是在其声明之后，至模板声明或定义结束之前。
2. 与其他任何名字一样，模板参数会隐藏外层作用域中声明的相同名字。但是，与大多数其他上下文不同，在模板内不能重用模板参数名：

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

// 由于参数名不能重用，所以一个模板参数名在一个特定模板参数列表中只能出现一次：
// 错误：非法重用模板参数名V
template <typename V, typename V> // ...
```

3. 一个特定文件所需要的所有模板的声明通常一起放置在文件开始位置，出现于任何使用这些模板的代码之前。
4. c++语言假定通过作用域运算符访问的名字不是类型。因此，如果我们希望使用一个模板类型参数的类型成员，就必须显式告诉编译器该名字是一个类型。我们通过关键字 `typename` 来实现这一点。
5. 当我们希望通知编译器一个名字表示类型时，必须使用关键字 `typename` ，而不能使用 `class` 。

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

6. 可以给函数的模板参数提供**默认实参（default template argument）。
7. 与函数默认实参一样，对于一个模板参数，只有当它右侧的所有参数都有默认实参时，它才可以有默认实参。

```
// 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;
}
```

8. 无论何时使用一个类模板，我们都必须在模板名之后接上尖括号。尖括号指出类必须从一个模板实例化而来。
9. 特别是，如果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，就必须在模板名之后跟一个空尖括号对：

```
template <class T = int> class Numbers { // T默认为int
public:
    Numbers(T v = 0) : val { }
    // ...
private:
    T val;
};

Numbers<long long> a;    // 使用long long实例化类模板
Numbers<> b;             // 空<>表示希望使用默认类型(int)
```

16.1.4 成员模板

1. 一个类（无论是普通类还是类模板）可以包含本身是模板的成员函数。这些成员被称为**成员模板（member template）**。成员模板不能是虚函数。
2. 对于类模板，我们也可以为其定义成员模板。在此情况下，类和成员各自有自己的、独立的模板参数。

```
template <typename T> class Blob {
    template <typename It> Blob(It b, It e);
    // ...
}

// 与类模板的普通成员函数不同，成员模板是函数模板。
// 当我们在类模板外定义一个成员模板时，必须同时为类模板和成员模板提供模板参数列表。
// 类模板的参数列表在前，后跟成员自己的模板参数列表：
template <typename T>    // 类的类型参数
tempalte <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)) {}
```

16.1.5 控制实例化

1. 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。在新标准中，我们可以通过**显示实例化（explicit instantiation）**来避免这种开销。
2. 当编译器遇到 `extern` 模板声明时，它不会在本文件中生成实例化代码。
3. 将一个实例化代码声明为 `exter` 就表示承诺在程序其他位置有该实例化的一个非 `extern` 声明（定义）。
4. 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。
5. 在一个类模板的实例化定义中，所用类型必须能用于模板的所有成员函数。

```
// Application.cc
// 这些模板类型必须在程序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};
Blob<int> a2(a1);               // 拷贝构造函数在本文件中实例化
int i = compare(a1[0], a2[0]);  // 实例化出现在其他地方

// templateBuild.cc
// 实例化文件必须为每个在其他文件中声明为extern的类型
// 和函数提供一个（非extern）的定义
template int compare(const int&, const int&);
template class Blob<string>;    // 实例化类模板的所有成员
```

16.2 模板实参推断

1. 对于函数模板，编译器利用调用中的函数实参来确定其模板参数。从函数实参来确定模板实参的过程被称为**模板实参推断（template argument deduction）**。

16.2.1 类型转换与模板类型参数

1. 如一个函数形参的类型使用了模板类型参数，那么它采用特殊的初始化规则。
2. 只有很有限的几种类型转换会自动地应用于这些实参。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。
3. 将实参传递给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。
4. 算术转换、派生类向基类的转换以及用户定义的转换，都不能应用于函数模板。
5. 如果形参是引用，则数组不能转换为指针。
6. 一个模板类型参数可以用作多个函数形参的类型。由于只允许有限的几个类型转换，因此传递给这些形参的实参必须具有相同的类型。如果推断出的类型不匹配，则调用就是错误的。

```
// 如果希望允许函数实参进行正常的类型转换，
// 我们可以将函数模板定义为两个类型参数：
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;
}

// 可以使用不同类型的实参了：
long lng;
flexibleCompare(long, 1024);    // 正确：int(long, int)
```

7. 函数模板可以有用普通类型定义的参数，即，不涉及模板类型参数的类型。这种函数实参不进行特殊处理：它们正常转换为对应形参的类型：

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

// 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
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&
```

16.2.2 函数模板显式实参

1. 显式模板实参按由左至由右的顺序与对应的模板参数匹配。只有尾部（最右）参数的显式模板实参才可以忽略，而且前提是是它们可以从函数参数推断出来。
2. 对于模板类型参数以及显式指定了的函数实参，可以进行正常的类型转换：

```
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)
```

16.2.3 尾置返回类型与类型转换

1. 尾置返回允许我们在参数列表之后声明返回类型，它可以使用函数的参数：
2. `decltype(*beg)` 返回元素类型的引用类型。 `remove_reference::type` 脱去引用，剩下元素类型本身。

```
template <typename It>
auto fcn(It beg, It end) -> decltype(*beg){
    // 处理序列
    return *beg;     // 返回序列中第一个元素的引用
}

// 当我们不想返回元素的引用，而是想返回元素的值时：
template <typename It>
auto fcn(It beg, It end) ->
    typename remove_reference<decltype<*beg>>::type {
    // 处理序列
    return *beg;    // 返回序列中第一个元素的拷贝
}
// 注意，`type`是一个类的成员，而该类依赖于一个模板参数。
// 因此，必须在返回类型的声明中使用typename来告知编译器，type表示一个类型。
```

16.2.4 函数指针和实参推断

1. 当一个函数的参数是一个函数模板实例的地址时，程序上下文必须满足：对每个模板参数，能唯一确定其类型或值。

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

// pf1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：不知道使用哪一个compare的实例

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

16.2.5 模板实参推断和引用

1. 如果一个函数参数是指向模板参数类型的右值引用（如，T&&），则可以传递给它任意类型的实参。
2. 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：
   (1) T& &、T& &&、T&& & 都折叠成类型 T&
   (2) 类型 T&& && 折叠成 T&&
3. 右值引用通常用于两种情况：**模板转发其实参**或**模板被重载**。

16.2.6 理解 std::move

1. 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);
}

string s1("hi!"), s2;

// 传入右值
s2 = std::move(string("bye!"));
// 这个调用会实例化为：
string&& move(string&& t);

// 传入左值
s2 = std::move(s1);
// 这个调用会实例化为：
string&& move(string& t);
```

2. 从一个左值 `static_cast` 到一个右值引用是允许的
3. 通常情况下， `static_cast` 只能用于其它合法的类型转换。但是这里有一条针对右值引用的特许规则：虽然不能隐式地将一个左值转换为右值引用，但是我们可以用 `static_cast` 显式地将一个左值转换为一个右值引用。
4. 对于操作右值引用的代码来说，将一个右值引用绑定到一个左值的特性允许它们截断左值。有时候我们知道截断一个左值是安全的。
   (1) 一方面，通过允许进行这样的转换，c++语言认可了这种用法。
   (2) 另一方面，通过强制使用 `static_cast` ，c++语言试图阻止我们意外地进行这种转换。
5. 统一使用 `std::move` 使得我们在程序中查找潜在的截断左值的代码变得很容易。

16.2.7 转发

1. 某些函数需要将其一个或多个实参连同类型不变地转发给其他函数。在此情况下，我们需要保持被转发实参的所有性质，包括实参类型是否是 `const` 的以及实参是左值还是右值。
2. 通过将一个函数参数定义为一个指向模板类型参数的右值引用，我们可以保持其对应实参的所有类型信息。而使用引用参数（无论是左值还是右值）使得我们可以保持 `const` 属性，因为在引用类型中的 `const` 是底层的。

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

3. 当用于一个指向模板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。
4. 与 `std::move` 相同，对 `std::forward` 不适用using声明是一个好主意。

```
template <template F, tempalte T1, template T2>
void flip(F f, T1&& t1, T2&& t2) {
    f(std::forward<T2>(t2), std::forward<T1>(t1));
}
```

16.3 重载与模板

1. 函数模板可以被另一个函数模板或一个普通非模板函数重载。与往常一样，名字相同的函数必须具有不同数量或类型的参数。
2. 正确定义一组重载的函数模板需要对类型间的关系及模板函数允许的有限的实参类型转换有深刻的理解。
3. 当有多个重载模板对一个调用提供同样好的匹配时，应该选择**最特例化**的版本。
4. 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。
5. 在定义任何函数之前，记得声明所有重载的函数版本。这样就不必担心编译器由于未遇到你希望调用的函数而实例化一个并非你所需的版本。

16.4 可变参数模板

1. 一个**可变参数模板（variadic template）**，就是一个接受可变数目参数的模板函数和模板类。可变数目参数被称为**参数包（parameter packet）**。
2. 存在两种参数包：**模板参数包（template parameter packet）**，表示零个或多个模板参数；**函数参数包（function parameter packet）**，表示零个或多个函数参数。
3. 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。

```
// 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 <tempalte T, typename... Args>
void foo(const T& t, const Args& ... rest);
```

4. 当我们需要知道包中有多少元素时，可以使用 `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
}
```

16.4.1 编写可变参数函数模板

1. 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。
2. 对于最后一个调用，两个函数提供同样好的匹配。但是，非可变参数模板比可变参数更特例化，因此编译器选择非可变参数版本。
3. 当定义可变参数版本的 `print` 时，非可变参数版本的声明必须在作用域中。否则，可变参数版本会无限递归。

```
// 该模板用来终止递归并打印最后一个元素
// 此函数必须在可变参数版本的print定义之前声明
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
}
```

16.4.2 包扩展

1. 对于一个参数包，除了获取其大小之外，我们能对它做的唯一的事情就是 **扩展（expand）** 它。
2. 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）**。
3. 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。我们通过在模式右边放一个省略号(...)来触发扩展操作。
4. 扩展中的模式会独立地应用于包中的每个元素

```
template <typename... Args>
ostream& errorMsg(ostream& os, const Arg&... rest) {
    // print(os, debug_rep(a1), debug_rep(a2),...,debug_rep(an))
    return print(os, debug_rep(rest)...);    // 正确

    // print(os, debug_rep(a1, a2, ..., an))
    return print(os, debug_rep(rest...));    // 错误：此调用无匹配函数
}
```

16.4.3 转发参数包

1. 可变参数函数通常将它们的参数转发给其他函数。这种函数通常具有我们的 `emplace_back` 函数一样的形式：

```
// fun有零个或多个参数，每个参数都是一个模板参数类型的右值引用
template<typename... Args>
void fun(Args&&... args)     // 将Args扩展为一个右值引用的列表
{
    // work的实参既扩展Args又扩展args
    work(std::forward<Args>(args)...);
}
```

16.5 模板特例化

1. 当我们不能（或不希望）使用模板版本时，可以定义类或函数模板的一个特例化版本。
2. 一个特例化版本就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。
3. 当定义函数模板的特例化版本时，我们本质上接管了编译器的工作。即，我们为原模板的一个特殊实例提供了定义。重要的是要弄清：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。因此特例化不影响函数匹配。
4. 模板及其特例化版本应该声明在同一个头文件中。所有同名模板的声明应该放在前面，然后是这些模板的特例化版本。

```
// 打开std命名空间，以便特例化std::hash
namespace std{
template <> // 我们正在定义一个全特例化版本，模板参数为Sales_data
struct hash<Sales_data>{
    // ... ...
    size_t operator() (const Sales_data& s) const;
};

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);
}
}

// 由于hash<Sales_data>使用Sales_data的私有成员，我们必须将它声明为友元
template<class T> class std::hash;  // 友元声明需要的
class Sales_data {
    friend class std::hash<Sales_data>;
    // 其他成员定义，如前
}

// 为了让Sales_data的用户能使用hash的特例化版本，我们应该在
// Sales_data的头文件中定义该特例化版本。
```

5. 与函数模板不同，类模板的特例化不必为所有模板参数提供实参。可以只指定一部分而非所有模板参数，或是参数的一部分而非全部特性。
6. 一个类模板的**部分特例化（partical specialization）** 本身是一个模板，使用它时用户还必须为那些在特例化版本中未指定的模板参数提供实参。
7. 只能部分特例化类模板，而不能部分特例化函数模板。
8. 我们可以只特例化特定成员函数而不是特例化整个模板。

```
//只特例化Bar，不特例化整个Foo
template <typename T> struct Foo {
    void Bar() {}
    // ...
}
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
}

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();       // 使用特例化版本的FOO<int>::Bar()
```