模板与泛型编程

面向对象编程（OOP）和泛型编程都能处理在编写程序时不知道类型的情况。不同之处在于：OOP能处理类型在程序运行之前都未知的情况；而在泛型编程中，在编译时就能获知类型了。

模板是C++中泛型编程的基础。一个模板就是创建类或者函数的蓝图。当使用泛型类或者泛型函数时，我们提供足够的信息，将蓝图转换为特定的类或函数。这种转换发生在编译时。

# 定义模板

## 函数模板

### 实例化函数模板

当我们调用一个函数模板时，编译器通常用函数实参来为我们推断模板实参。假如有一个模板函数

template <typename T>

int compare(const T& v1, const T& v2)

{…}

当我们调用compare时，编译器使用实参的类型来确定绑定到模板参数T的类型。例如，在下面的调用中：

cout << compare(1, 0) << endl;

编译器会推断出模板实参为int，并将它绑定到模板参数T。

### 非类型模板参数

除了类型参数，还可以在模板中定义非类型参数。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。

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。这个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。

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引用。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。我们不能用一个普通（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。

### inline或constexpr的函数模板

inline或constexpr说明符放在模板参数列表之后，返回类型之前。

### 编写类型无关的代码

模板程序应该尽量减少对实参类型的要求。

### 模板编译

函数模板和类模板成员函数的定义通常放在头文件中。

通常，当我们调用一个函数时，编译器只需要掌握函数的声明。当我们使用一个类类型的对象时，类定义必须是可用的，但成员函数的定义不必已经出现。因此我们将类定义和函数声明放在头文件中，而普通函数和类的成员函数的定义放在源文件中。

模板则不同：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，与非模板代码不同，模板的头文件通常既包括声明也包括定义。

## 类模板

与函数模板不同，编译器不能为类模板推断模板参数类型。

### 类模板成员函数的实例化

默认情况下，一个类模板的成员函数只有当程序用到它时才进行实例化。如果一个成员函数没有被使用，则它不会被实例化。

### 在类代码内简化模板类名的使用

当我们使用一个类模板类型时必须提供模板实参，但这一规则有一例外。在类模板自己的作用域中，可以直接使用模板名而不提供实参：

template <typename T> class BlobPtr

{

BlobPtr& operator++();

}

### 类模板与友元

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。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。

为了引用类模板或函数模板的一个特定实例，必须首先声明模板自身。一个模板声明包含模板参数列表：

// 前置声明，在Blob中声明友元所需要的

template <typename T> class BlobPtr;

template <typename T> 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用相同类型实例化的BlobPtr和相等运算符

friend class BlobPtr<T>;

friend bool operator==(const Blob<T>&, const Blob<T>&);

}

一个类也可以将另一个模板的每个实例都声明为自己的友元，或者限定特定的实例为友元：

// 前置声明，在将模板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的每个实例都将相同实例化的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类模板本身不同的模板参数。

在新标准中，可以将模板类型参数声明为友元：

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

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; // authors是一个pair<string, string>

一个模板类型别名是一族类的别名。

## 模板参数

### 使用类的类型成员

在普通代码中，编译器掌握类的定义。因此，它知道通过作用域运算符访问的名字是类型还是static成员。但对于模板代码，假设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，它不会知道mem是一个类型成员还是一个static数据成员，直到实例化时才会直到。例如编译器遇到语句：T::size\_type \* p;它需要知道我们是正在定义一个名为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++语言假定通过作用域运算符访问的名字不是类型。我们通过使用关键字typename显示告诉编译器该名字是一个类型。

### 默认模板实参

新标准中，可以为函数和类模板提供默认实参。更早的C++标准只允许为类模板提供默认实参。

## 成员模板

一个类（普通类或模板类）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。

### 普通类的成员模板

与普通函数模板相同；

### 类模板的成员模板

类和成员各自有自己的、独立的模板参数。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>

template <typename It>

Blob<T>::Blob(It b, It e) {…}

## 控制实例化

当两个或多个独立编译的源文件使用了相同的模板，并提供了相同的模板参数时，每个文件中都会有该模板的一个实例。

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，可以通过显示实例化避免这种开销。显示实例化的形式如下：

extern template declaration; // 实例化声明

template declaration; // 实例化定义

declaration是一个类或函数声明，其中所有模板参数已被替换为模板实参。例如：

extern template class Blob<string>;

template int compare(const int&, const int&);

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。将一个实例化声明为extern就表示承诺在程序的其他位置有该实例化的一个非extern声明(定义)。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。

由于编译器在使用一个模板时自动对其实例化，因此extern声明必须出现在任何使用此实例化版本的代码之前。

对每个实例化声明，在程序的某个位置必须有其显示的实例化定义。

### 实例化定义会实例化所有成员

一个类模板的实例化定义会实例化该模板的所有成员，包括内联的成员函数。因此，我们用来显示实例化一个类模板的类型，必须能用于该模板的所有成员。

# 模板实参推断

从函数实参来确定模板实参的过程被称为模板实参推断。

## 类型转换与模板类型参数

将实参传递给带模板类型的函数形参时，编译器通常不对实参进行类型转换，而是生成一个新的模板实例。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。

* const转换：可以将一个非const对象的引用或指针传递给一个const对象的引用或指针形参。
* 数组或函数指针转换：如果函数形参不是引用类型，则可以对数组或函数类型的实参应用正常的指针转换。一个数组实参可以转换为一个指向其首元素的指针。类似的，一个函数实参可以转换为一个该函数类型的指针。

其他类型转换，如算术转换、派生类到基类的转换、用户定义的转换，都不能应用于函数模板。

### 正常类型转换应用于普通函数实参

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。