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。