三、类模板

如果一个类的成员变量的类型、成员函数返回值、参数表、局部变量的类型、成员类型(嵌套类)甚至基类的类型(选爹)，用参数表示，那么这个类就是一个类模板。

附注：只要有类模版参数，不使用也是类模版。

1.定义

template<typename 类型形参1，typename 类型形参2, ...>

class 类模板名 [: 继承表] {

成员变量

成员函数

成员类类型

};

在成员变量、成员函数、成员类型，以及继承表中都可以引用类型形参。

例如：

template<typename M, typename R, typename A，tyepname V, typename T, typename B>

class MyClass:public B{

M m\_mem;

R fun(A arg){ ... V var ... }

typedef T\* pointer;

};

--------------------------

2.使用

类模板名<类型实参1,类型实参2,...> 对象(构造实参表);

类模板名<类型实参1,类型实参2,...>& 引用 = 对象;

类模板名<类型实参1,类型实参2,...>\* 指针 = &对象;

|<------------------------------>|

类

用类模板创建对象实际上包含了两个实例化过程：

**处理时间： 编译期 运行期**

**处理者： 编译器 处理器**

**处理过程： 类模板 ==实例化==> 类 ==实例化==> 对象**

任何时候一个类模板都不具备类型语义,只有的当该类模板结合了类型实参，即被实例化为具体类以后才具有类型语义。

举例：

A.类模板不能放在"::"运算符前面；

B.类模板不能作为sizeof/typeid运算符的运算数；

----------------------

1. 类型参数
2. 类模板的类型参数不能隐式推断，必须显式指明。
3. 类模板中，只有那些被调用的成员函数才会被实例化，才会二次编译。某些类型虽然并没有提供类模板所要求的全部功能，但照样可以实例化该类模板，只要不直接或间接调用那些依赖于未提供功能的成员函数即可(这一点，与函数模板不同)。
4. 作为类模板的设计者，应该尽量减少对类型实参的要求，以降低类模板使用者的工作量。

注解：在不影响功能的前提下，类模板中需要支持的操作符数量越少越好，即类模板的通用性。

参见：ctmpl.cpp

---------------------------

4.静态成员变量

类模板的静态成员变量，既不是一个对象一份，也不是一个模板一份，而是在该类模板的每个实例化类中各有一份，且为该实例化类的所有对象所共享。

参见：static.cpp

-------------------------

5.递归实例化(空间递归)

(1)用一个类模板的实例化类型实例化该类模板自身的实例化方式被称为类模板的递归实例化。

(2)通常用这种方法可以很方便地构建那些在空间上具有递归特性的数据结构，如：多维数组、二叉树等。

注意：模板参数表的右尖括号中间至少要保留一个空格，以防止编译器将其误解为右移运算符。

参见：array.cpp

---------------------------

扩展用法：

template<typename T> class Array; // 数组

template<typename T> class List; // 链表

Array<Array<int> > a; // 二维数组

List<List<int> > b; // 二维链表

Array<List<int> > c; // 链表数组

List<Array<int> > d; // 数组链表

template<typename T> class BiTree; // 二叉树

BiTree<List<Array<int> > > e; // 数组链表二叉树

-------------------------------------

6.特(例)化(可以理解为重载)

如果一个类模板的通用实现针对某些特殊的类型实参，不能做出正确的处理，或者处理虽然可行但性能不佳，那么就可以针对该类型实参给出特殊的实现，以弥补通用版本的不足，这就叫类模板的特(例)化。

1)全类(模板)特化

template<>

class 类模板名<类型实参1, 类型实参2, ...>{ ... };

2)成员特化

template<>

返回类型 类模板名<类型实参1,类型实参2, ...>::成员函数名(调用实参表){ ... }

注意：类模板的特化只是一种权宜之计，不应成为损失通用设计的借口。

参见：spec.cpp

-------------------------------

7.局部特化(偏特化)

1)只针对通用版本中的部分类型参数所做的特化被称为类模板的局部特化或偏特化。

2)类模板的局部特化只能以全类方式进行，不能只针对成员函数做特化。

3)虽然没有强调类型参数取某种特定类型，但是强调类型参数之间的某种特殊关系或特殊属性，这种情况也属于局部特化。

参见：part.cpp

-----------------------------

8.智能指针(auto\_ptr和smart\_ptr)

1)利用栈对象析构过程销毁堆对象；

2)利用类模板做到与类型无关(包括基本类型)；

3)利用操作符重载模拟普通指针的行为；

4)利用转移拷贝避免double free异常(smart\_ptr中用引用计数解决)；

5)利用局部特化管理动态分配的对象数组(smart\_ptr中可以)。

参见：auto.cpp