華中科技大學

第13章 模板与内存回收

许向阳

xuxy@hust.edu.cn

大纲

13.0 模板的概念

13.1 变量模板及其实例

13.2 函数模板

13.3 函数模板实例化

13.4 类模板

13.5 类模板的实例化及特化

13.6 内存回收实例

13.0 模板的概念


```
int     add(int   a, int   b)
   {   int  x;    x=a+b;   return  x; }

double  add(double a, double b)
   {   double x;  x=a+b;   return  x; }

short   add(short    a, short    b)
    {  short  x;  x=a+b;   return  x; }
```

参数化多态性：

- 将一段程序所处理的对象类型进行参数化
- 使一段程序代码可以用于处理多种不同类型的对象

13.0 模板的概念

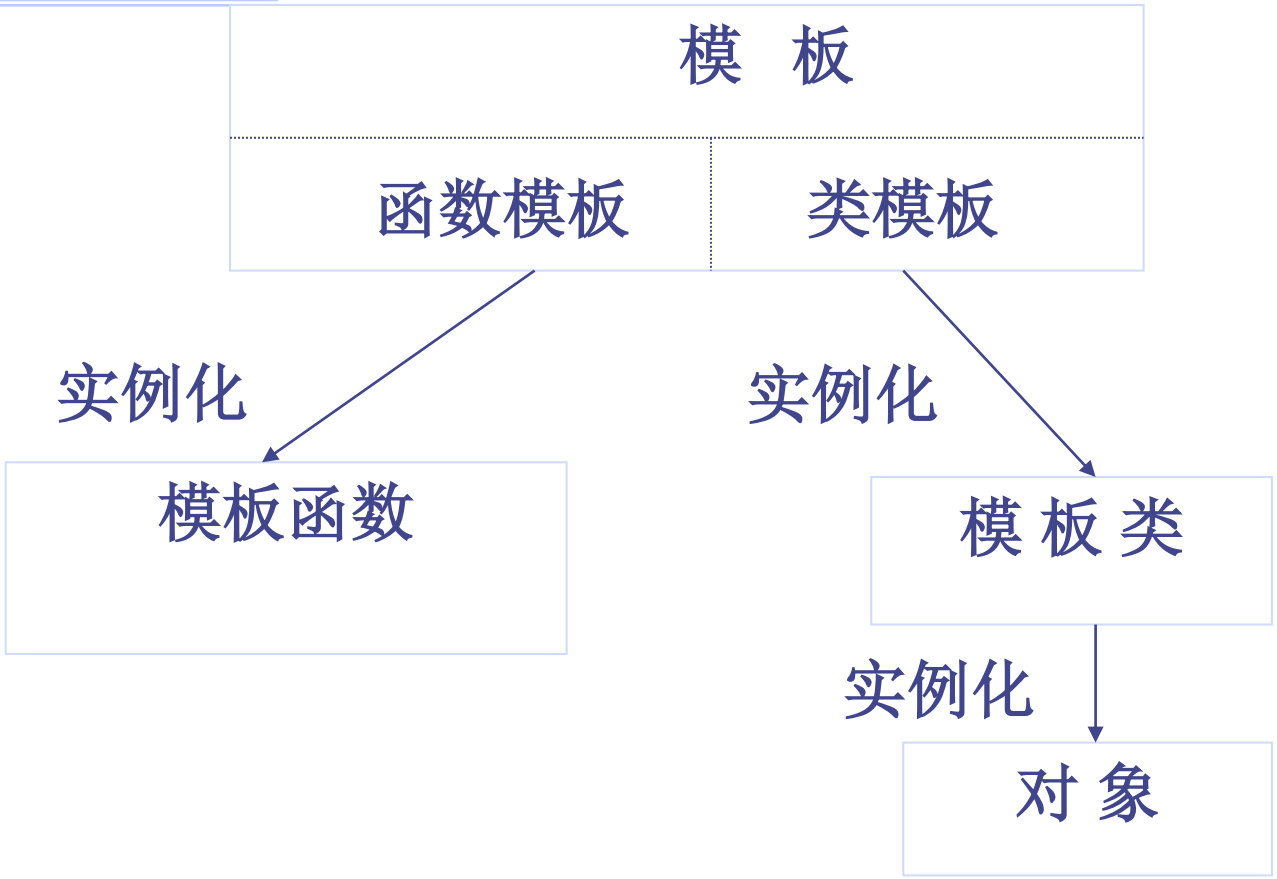
- 模板是对具有相同特性的函数或类的再抽象
- 模板是一种参数化的多态性工具；

采用模板编程，可以为各种逻辑功能相同而数据类型不同的程序提供一种代码共享的机制。

13.0 模板的概念

- 模板并非通常意义上可直接使用的函数或类
- 模板是对一族函数或类的描述，是参数化的函数和类。
- 模板不是以数据为参数，而是以它所使用的数据类型为参数
- 模板是一种使用无类型参数来产生一族函数或类的机制，它由通用代码构成。

13.0 模板的概念



模板实例化示意图

13.2 函数模板


```
int  add(int a, int b) {
      int   x;
      x= a+b;
      return  x;
}
```

```
double  add(double a, double b) {
      double   x;
      x= a+b;
      return  x;
}
```

T 换成 int

T 换成 double

```
T  add(T   a, T   b) {
      T   x;
      x= a+b;
      return  x;
}
```

13.2 函数模板


```
template  <模板形参表>
返回类型   函数名  (参数表)
{
        函数体
}
```

```
template <class T>
T  add(T   a, T   b) {
      T   x;
      x= a+b;
      return  x;
}
```

调用时：

```
int        x, y, z;
double u,v,w;
z =  add(x,  y);
w = add(u, v);
```

模板函数名：add

参数化的类型：T

13.2 函数模板

在机器代码中，有add<int> 和add<double>两个函数体

```
; 17   : z=add(x,y);

        mov     eax, DWORD PTR _y$[ebp]
        push    eax
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        call    ??$add@H@@YAHHH@Z
                        ; add<int>
```

13.2 函数模板


```
; 21   : w=add(u,v);
        sub     esp, 8
        movsd xmm0, QWORD PTR _v$[ebp]
        movsd QWORD PTR [esp], xmm0
        sub     esp, 8
        movsd xmm0, QWORD PTR _u$[ebp]
        movsd QWORD PTR [esp], xmm0
        call    ??$add@N@@YANNN@Z
                        ;  add<double>

        call    ??$add@H@@YAHHH@Z
                        ; add<int>
```

两个函数的名称不相同

13.2 函数模板


```
w=add(u,v);      // 根据实参，自动确定参数类型
                 // 生成模板函数
    call         add<double>
```

```
w= add<int>(u, v);   // 指定参数类型，
                     // 实参按参数类型进行转换
```

```
cvttsd2si   eax,mmword ptr [v]
push        eax
cvttsd2si   ecx,mmword ptr [u]
push        ecx
call        add<int>
```

13.2 函数模板

- <模板形参表>可以包含一个或多个用逗号分开的参数
- 每一项均有关键字class或 typename引导一个标识符
- 此标识符为模板参数，表示一种数据类型
- 类型可以是基本数据类型，也可以是类类型
- 参数表中至少有一个参数说明
- 参数在函数体中至少应用一次

13.2 函数模板

- 函数模板只是一种说明，并不是一个具体函数
- 它是一组函数的模板，在定义中使用了参数化类型
- 编译系统对函数模板不产生任何执行代码
- 只有在遇到具体函数调用的时候，根据调用处的具体参数类型，在参数实例化以后才产生相应的代码，此代码称为模板函数。

13.2 函数模板

- 函数模板中可以使用类型形参和非类型形参；
- 可以单独定义类的函数成员为函数模板；
- 函数模板不能在非成员函数的内部声明；
- 根据函数模板生成的模板实例函数也和函数模板的作用域相同。

13.2 函数模板

- 函数模板中可以使用类型形参和非类型形参；
- 可以单独定义类的函数成员为函数模板；
- 函数模板不能在非成员函数的内部声明；
- 根据函数模板生成的模板实例函数也和函数模板的作用域相同；
- 在调用函数时可隐式自动生成模板实例函数；
- 可强制显式生成模板实例函数

  template 返回类型 函数名<类型实参>(形参列表);

13.2 函数模板

- 函数模板中可以使用类型形参和非类型形参；
- 可以单独定义类的函数成员为函数模板；
- 函数模板不能在非成员函数的内部声明；
- 根据函数模板生成的模板实例函数也和函数模板的作用域相同；
- 在调用函数时可隐式自动生成模板实例函数；
- 可强制显式生成模板实例函数

  template 返回类型 函数名<类型实参>(形参列表);

13.2 函数模板

- 在调用函数时可隐式自动生成模板实例函数;
- 可强制显式生成模板实例函数

template 返回类型 函数名<类型实参>(形参列表);

```
template <class T>
T  func_add(T  a, T  b)
{        T   x;
         x = a+b;
         return x;
}
template  int  func_add<int>(int, int);   // 强制显式
z=func_add(x, y);   // 隐式自动
```

类中的函数模板

```
#include<typeinfo>
class ANY   {   //   可存储任何简单类型值的类ANY
    void * p;
    const char * t;
public:
    template <typename T> ANY(T x) {

        p = new T(x);
        t = typeid(T).name();
    }
    ~ANY() noexcept { if (p) { delete p;  p = nullptr; } }
}a(20);       // t 为 int
ANY    b(3.5);     // t 为 double
```

任意个类型形参的函数模板

```
int  println() {
    cout << endl;          return 0;
}
template < class H,class ...T>   // 用…表示任意个类型形参
int  println(H h, T ...t) {    //递归下降展开函数的参数表
    cout << h << " * ";
    return 1 + println(t...);     //递归下降调用
}
int n= println(1, '2', 3.3, "expand");   // n=4
```

用递归定义的方法可展开并处理类型形参。
生成实例函数时，可能因递归生成多个实例函数。

模板实例函数的隐藏

```
template <typename T>
T max(T a, T b)
{
    return a>b?a:b;
}
template < >    //此行可省
const char *max(const char *x, const char *y)
{
    return strcmp(x, y)>0?x:y;
}
// 特化实例函数：特化函数将被优先调用
//                    可用于隐藏模板实例函数
const char  *p, *q;
 p = max(p, q);
```

模板实例函数的隐藏

- 优先使用实参与形参完全匹配的特化函数；
- 使用实参与形参能够匹配的模板实例函数；
- 若模板函数调用时，明确了实参类型，对调用实参进行强制类型转换。

```
const char  *p, *q;      p = max(p, q);   // 使用特化函数
int x,  y, z;                z=max(x, y);     // 实参与形参匹配
double u,v,w;          w=max<int>(u,v); //实参强制类型转换
z=max(x, 'a');  // 模板参数不明确, 一个为int ,另一个为char
z = max('x', 'a');
```

13.4 类模板

- 定义一个堆栈类 STACK
- 可以将数据元素压栈
- 可以从栈中弹出数据元素

- 数据元素是整数类型时 STACK_INT
- 数据元素是double类型时 STACK_DOUBLE
- 数据元素是某一类型时 STACK_?

13.4 类模板

- 类模板是参数化的类，即用于实现数据类型参数化的类；
- 应用类模板可以使类中的数据成员、成员函数的参数及成员函数的返回值能根据模板参数匹配情况取任意数据类型；
- 类型既可以是C++预定义的数据类型，也可以是用户自定义的数据类型。

13.4 类模板


```
template  <模板形参表>
class 类名
{
    <类体说明>
};
```

- <模板形参表>中包含一个或多个用逗号分开的参数项，每一参数至少应在类的说明中出现一次；
- 参数项可以包含基本数据类型，也可以包含类类型；
- 若为类类型，使用前缀class。
- 形参类型用于说明数据成员和成员函数的类型。

类模板的申明

```
template <class T>
class VECTOR
{
  private:
          T  *data;
          int  size;
  public:
          VECTOR(int);
          ~VECTOR();
          T  &  operator[](int);
        VECTOR &  operator=(const VECTOR & a);
};
```

类模板的中函数的定义

```
template <class T>
VECTOR<T>::VECTOR(int n)
{       data = new T[size=n];  }

template<class T>
VECTOR<T>::~VECTOR()
{       delete []data;  }

template<class T>
T & VECTOR<T>::operator[](int i)
{       return data[i];  }
```

将函数的实现 也放在头文件中

类模板的中函数的定义

VECTOR & operator=(const VECTOR & a);

```
template <class T>
VECTOR<T> &  VECTOR<T>:: operator=(const VECTOR & a);
{
    ……
}
```

模板类对象及其使用

```
void main()
{       VECTOR<int>    LI(20);
        VECTOR<double>   LD(30);
        LI[0] = 10;
        LI[1] = 20;
        int z=LI[0] + LI[1];
        cout << z << endl;              // 显示 30
}
```

- 自动生成模板类

  模板<模板参数表> 对象名1, 对象名2,…;

- 类模板强制实例化

  template VECTOR<int>;

13.4 类模板

T & VECTOR<T>::operator[](int i)

```
VECTOR<int>    LI(20);
LI[0] = 10;
        push       0
        lea         ecx,[LI]
        call        VECTOR<int>::operator[]
        mov        dword ptr [eax],0Ah
LI[1] = 20;
int z=LI[0] + LI[1];
        push       0
       lea         ecx,[LI]
       call        VECTOR<int>::operator[]
       mov          esi,eax    ;  esi是LI中 data[0] 的地址
```

13.4 类模板

缺省的类型参数

```
template <class T=int>
class VECTOR
{
  private:
        T  *data;
        int  size;
  public:
        VECTOR(int);
        ~VECTOR();
        T  &  operator[](int);
};
```

```
VECTOR<int>    LI(20);
VECTOR<>       LI(20);
```

13.4 类模板

- 类模板自身并不产生代码
- 它指定类的一个家族
- 当引用时，才产生代码，生成模板类

13.4 类模板

类模板与继承

① 类模板可以从类模板派生

基类模板 **VECTOR<T>**

```
template <class T>
class VECTOR
{
  private:
        T *data;
        int size;
  public:
        VECTOR(int);
        ~VECTOR();
        T & operator[](int);
        int getsize() {return size;}
};
```

13.4 类模板

类模板与继承

① 类模板可以从类模板派生

```
template <class T>
class STACK : public VECTOR<T>
{
private:
        int  top;
public:
        int  full() { return  top==getsize();}
        int  empty() {return top==0;}
        int  push(T t);
        int  pop(T &t);
        STACK(int s):VECTOR<T>(s) { top=0;}
        ~STACK() { }
};
```

基类模板 **VECTOR<T>**
派生类模板 **STAC<T>**

13.4 类模板

类模板与继承

① 类模板可以从类模板派生

```
template <class T>
int STACK<T>::push(T t)
{
        if (full()) return 0;
        (*this)[top++]=t;
        return 1;
}
```

13.4 类模板

类模板与继承

① 类模板可以从类模板派生

```
void main()
{
        STACK<int>   SI(20);
        STACK<double>   SD(30);

        SI.push(10);
        SI.push(20);
        SI.push(30);
}
```

② 非模板类可以从模板类派生

```
template <class T>
class base
{
……
};

class derive:public base<int>
{
……
};
```

derive 不是类模板，不含参数，其基类是实例化的模板类

类模板与继承

- 在派生中，作为非模板类的基类，必须是类模板实例化后的模板类
- 在定义派生类前不需要模板声明语句：
  template<class T>

定义对象：
derive obj1(…),…

13.4 类模板

类模板与继承

③类模板可以从非模板类派生

② 非模板类可以从类模板派生

① 类模板可以从类模板派生

总结

模板的概念

函数模板的定义、使用

- 自动生成模板实例函数
- 函数模板强制实例化
- 函数模板实例特化
- 任意个形参的函数模板
- 成员函数模板

类模板的定义、使用

总结

类模板的定义、使用

自动生成类模板的模板实例

类模板强制实例化

类模板实例特化、类模板的部分特化

省略参数的类模板、 多类型参数模板

派生类模板

变量模板

变量模板使用类型形参定义变量的类型，

可根据类型实参生成变量模板的实例变量。

在函数模板、类模板中已有变量模板。

```
template <class T>
T  add(T   a, T   b) {
      T   x;
      x= a+b;
      return  x;
}
```

```
template <class T>
class VECTOR {
   private:
         T  *data;
         int  size;
};
```

变量模板


```
template<typename  T>   // template<classname  T>
 T   xxx;
                                  // 显式 定义实例变量
xxx<double> = 11.22;    // double xxx=11.22;
cout<< xxx<double> *2 << endl;    // 22.44

xxx<int> =56;        // int xxx=56;
xxx<short>;         // short xxx;
```

变量模板


```
template<typename  T>
constexpr T pi = T(3.14159265358979323851);

template<class T>
T  area(T  r)  {
    printf("%p\n", &pi<T>);
        //生成模板函数实例时，也生成pi<T>的模板实例变量
        // %p, 以十六进制显示 实例化变量的地址
    return pi<T> * r * r;
}

int a1 = area<int>(3);     //  a1 = 27
double a2 = area<double>(3);  //  a2 = 28.2743
```

变量模板

- 变量模板不能在函数内部声明；
- 显式或隐式生成的实例变量和变量模板的作用域相同；
- 因此，变量模板生成的模板实例变量只能为全局变量或者模块静态变量；
- 模板的参数列表除了可以使用类型形参外，还可以使用非类型的形参；
- 变量模板实例化时，非类型形参需要传递常量作为实参。
- 非类型形参可以定义默认值，若变量模板实例化时未给出实参，则使用其默认值实例化变量模板。

变量模板


```
template<class T, int x=3>
static  T  girth = T(3.14159265358979323 85L*2*x);
    //定义变量模板girth，其类型形参为T，非类型形参 x

template float girth<float>;
     //生成static girth<float>，作用域与变量模板相同

cout<<  girth<double, 4> <<endl;       // 25.1327
```