華中科技大學

第13章　模　　板

许向阳

xuxy@hust.edu.cn

泛型程序设计思想

大纲

13.1 模板的概念

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 智能指针及其实现原理

13.7 STL——Standard Template Library

课堂目标

理解：类模板是对一组具有公共性质的类的抽象；

模板 ——通用代码；　泛型程序设计的思想

实现原理——编译器起到抄写/替换的作用

- 学会定义类模板，使用模板类编写程序；
- 掌握智能指针的实现关键技术：

  指针封装成对象、不同类型的指针抽象为模板
- 学会使用智能指针编写程序；
- STL 中的容器、迭代器、算法、空间配置器、仿函数各自的作用，以及相互协作的关系；
- 使用 STL 编写程序

13.1 模板的概念

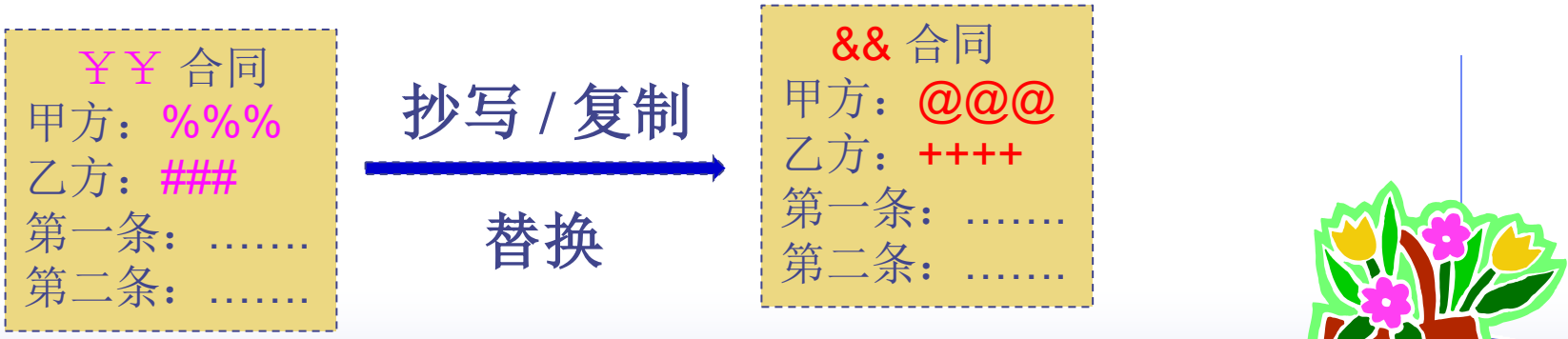
- 日常生活中，经常使用模板

  保险合同

  租房协议

  PPT模板

  报告模板



13.1 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; }

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

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

参数化多态性：

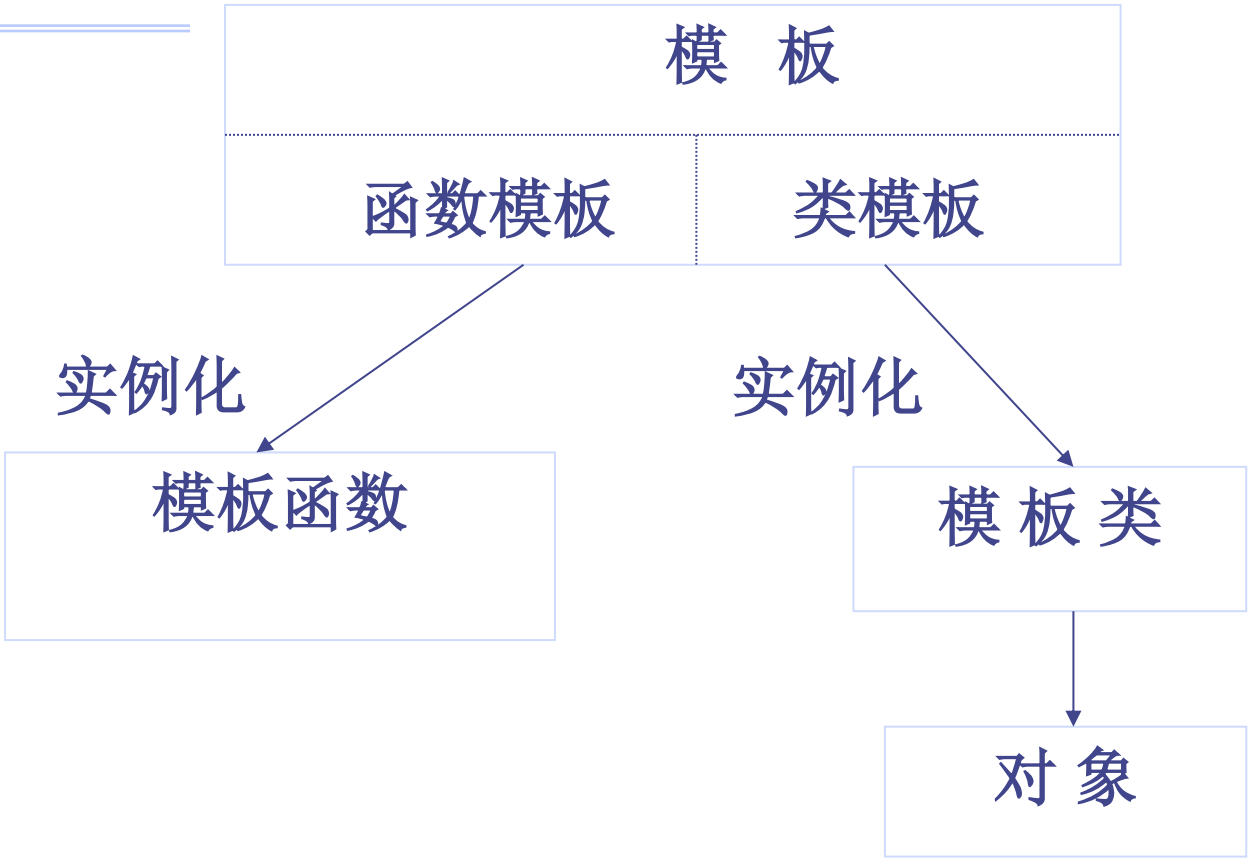
- 将函数所处理的对象类型进行参数化
- 使一个函数可以处理多种不同类型的对象

13.1 模板的概念


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

- 模板并非通常意义上可直接使用的函数或类，编译器无法对模板生成OBJ文件或可执行代码；
- 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
- 模板是一种参数化的多态性工具，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。

13.1 模板的概念



实例化：用具体的数据类型代替参数化数据类型

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

```
template <class T>
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;
}
```

多参数模板 template <class T1， typename T2, …>

调用时：

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

MAT  a1,  a2, a3;
a3 = add(a1, a2);
```

函数模板名：add

参数化的类型：T

13.2 函数模板

在机器代码中，有add<int> 、add<double>、add<MAT>
三个不同的函数体

```
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


```
w=add(u,v);     // u,v,w double类型
……
call     ??$add@N@@YANNN@Z      ;add<double>
```

```
z=add(x,y);     // x,y,z int类型
……
call     ??$add@H@@YAHHH@Z     ; add<int>
```

```
a3=add(a1,a2);     // a1,a2,a3 MAT类型
……
call  ??$add@VMAT@@@@YA?AVMAT@@V0@0@Z; add<MAT>
```

在机器语言程序中，三个函数的名称不相同！

C++的换名机制，实现静态多态的武器！

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

- 函数模板只是一种说明，并不是一个具体函数
- 编译系统对函数模板不产生任何执行代码
- 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

- 模板实例函数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（不调用也生成实例函数）

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
z=func_add<int>(x, y);   // 强制调用指定类型的函数
```

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;   // p 指向的类型名
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。

13.2 函数模板

模板实例函数的隐藏
——特化实例函数

通用规则 VS 特殊情况

```
template <typename T>
T max(T a, T b)
{
    return a>b?a:b;
}
template < >   //此行可省，特化实例函数
const char *max(const char *x, const char *y)
{
    return strcmp(x, y)>0?x:y;
}
```

特化函数被优先调用，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；
- 其次使用实参与形参能够匹配的模板实例函数；
- 明确了实参类型时，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、函数成员的参数及函数成员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。

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);
};
```

13.4 类模板


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

简单的类型名替代、类型替代

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

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

13.4 类模板


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

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

- 类中的成员函数名不变；构造函数、析构函数等的名称没有变化；
- 函数参数无变化；
- 函数的返回类型 有变化。

13.4 类模板

类模板的中函数的定义　　将函数的实现 也放在头文件中

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

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

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

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>;

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

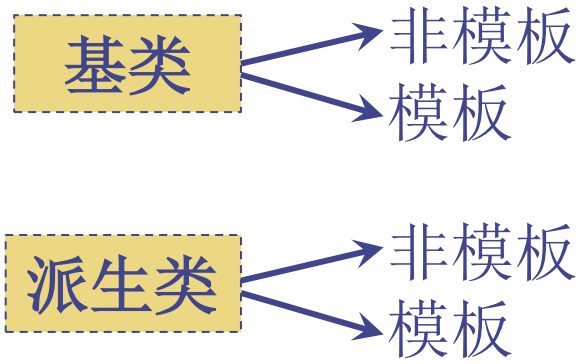
- 类模板自身并不产生代码
- 它指定类的一个家族
- 当引用时，才产生代码，生成模板类

类模板与继承

① 类模板可以从类模板派生

② 非模板类可以从模板类派生

③ 类模板可以从非模板类派生



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;}
};
```

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;
}
```

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);
}
```

13.4 类模板

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。

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

① 类模板可以从类模板派生

② 非模板类可以从类模板派生

③类模板可以从非模板类派生

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(3.14159265358979323385L);

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
```

应用示例

在 **mat.h** 中，申明一个类模板

```
template <typename T>
class MAT {
        int r, c;      // 矩阵的行r和列c大小
        T*  e;         // 指向所有整型矩阵元素的指针
  public:
        MAT(int r, int c);
        MAT(const MAT& a);
        MAT(MAT&& a)noexcept;
        virtual ~MAT()noexcept;
        virtual T* operator[ ](int r);
        virtual MAT operator+(const MAT& a)const;
        ……
};
```

应用示例

在 **mat.hpp** 中，定义类模板的成员函数

```
#include “mat.h”
template <typename T>
MAT<T>::MAT(int r, int c) : r(r), c(c), e(new T[r * c]) { }

template <typename T>
MAT<T>::MAT(const MAT& a):r(a.r),c(a.c),e(new T[a.r*a.c])
{   memcpy(e, a.e, sizeof(T) * r * c);  }

template <typename T>
MAT<T> MAT<T>:: operator+(const MAT& a)const
{    int i, j;       MAT temp(a.r, a.c);
     for (i = 0;i < r;i++)
        for (j = 0;j < c; j++)
             temp[i][j] = ((MAT &)*this)[i][j] + ((MAT &)a)[i][j];
   return temp;
}
```

应用示例

在 **test.cpp** 中，定义模板类对象

```
#include “mat.hpp”

void test( )
{
    MAT<int>   a(1, 2);

    MAT<int>   b=a;

   a[0][0] = 1;

}
```

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

将模板的实现放在头文件中，

只有在实例化时，编译器才生成对应的代码