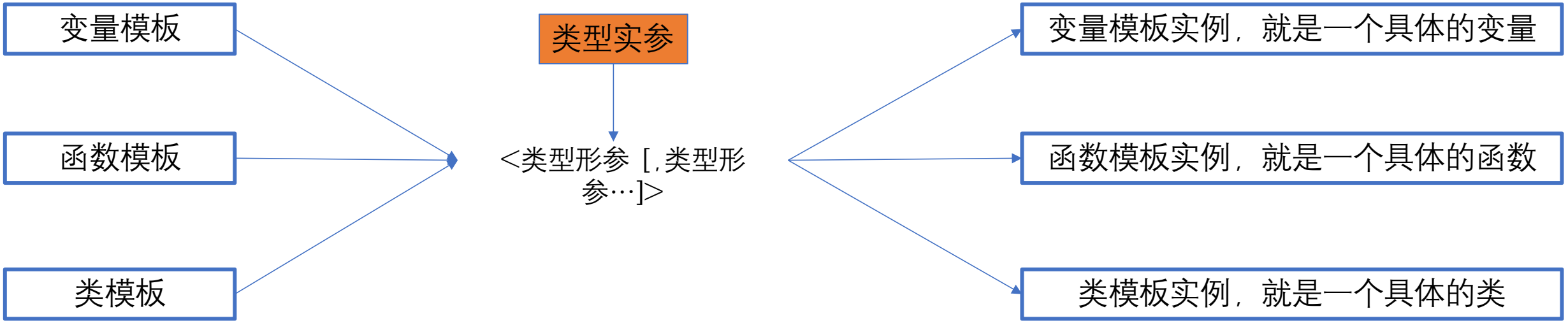
C++程序设计精要教程

华中科技大学

第13章 模板与内存回收

◆13.1 变量模板及其实例

- C++提供了3种类型的模板，即变量模板、函数模板和类模板。
- 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**。
- 生成实例变量的途径有三种：一种是从变量模板隐式地或显式地生成模板实例变量；另一种是通过函数模板（见13.2节）和类模板（见13.4节）生成。
- 在定义变量模板、函数模板和类模板时，**类型形参**的名称可以使用关键字class或者typename定义，即可以使用“template<class T>”或者“template<typename T>”。
- 生成模板实例变量时，将使用实际类型名代替T。实际类型名可以是内置类型名、类名或类模板实例（类模板实例就是具体的class，是类型）



编译器是根据模板实例进行编译和类型检查，因此要求模板函数、和模板类的具体实现必须要放在模板所在的头文件里。

第13章 模板与内存回收

【例13.1】定义变量模板并生成模板实例变量。

```
#include<stdio.h>
template<typename  T>
constexpr T pi = T(3.14159265358979323851);          //定义变量模板pi，其类型形参为T

template<class T> T area(T r) {          //定义函数模板area，其类型形参为T
   printf(“%p\n”, &pi<T>);                        //生成模板函数实例时附带生成变量模板pi<T>的变量模板实例
   return pi<T> * r * r;
}
```

类型实参

VS2019不支持显式生成变量模板实例，下面二行编译报错：fatal error C1001: 内部编译器错误。注意：pi<float>就是一个变量

```
template const float pi<float>;          //显式生成变量实例pi<float>，不能使用constexpr
//等价于“const float pi<float>= float (3.14159265358979323851);”
template const double pi<double>; //生成模板实例变量pi<double>，实参类型为double
//等价于“const double pi<double >= double (3.14159265358979323851);”
void main(void)
```

第13章 模板与内存回收

直接使用变量模板例可以

```
{
   const float &d1 =pi<float>;                    //引用变量模板生成的模板实例变量pi< float>
   const double &d2=pi<double>;                   //引用变量模板生成的模板实例变量pi<double>
   const long double &d3=pi<long double>; //生成并引用模板实例变量pi<long double>
   float a1= area<float>(3);                      // area<float>为函数模板area<T>的实例函数
   double a2 = area<double>(3);
   long double a3 = area<long double>(3);
   int a4 = area<int>(3);                         //调用area<int>时，隐式生成变量模板实例pi<int>
}
```

第13章 模板与内存回收

◆13.1 变量模板及其实例

- 变量模板不能在函数内部声明。
- 显式或隐式实例化生成的变量模板实例和变量模板的作用域相同。因此，变量模板生成的变量模板实例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。
- 在变量模板实例化时，非类型形参需要传递常量作为实参。
- 非类型形参也可以定义默认值，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。

第13章 模板与内存回收

【例13.2】使用非类型形参定义变量模板，并生成变量模板的模板实例变量。

```
#include<stdio.h>
template<typename  T>                          //定义变量模板pi，其类型形参为T
constexpr  T  pi = T(3.14159265358979323851);

template<class T> T area(T r) {                //定义函数模板area，其类型形参为T
   return pi<T> * r * r;
}
template<class T, int x=3>                     //定义变量模板girth，其类型形参为T
static  T  girth = T(3.14159265358979323851*2*x);

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

非类型形参，可以给默认值

第13章 模板与内存回收

```
void main(void)
{
   const double &f=pi<double>;                          //引用在main()外生成的全局变量pi<double>
   double &g=girth<double,4>; //引用在main()外生成的static girth <double>，非类型实参传递常量
   double &h=girth<double, sizeof(printf("abc"))>;//引用在main()外生成的static girth <double>
   double &k=girth<double>;                  //引用在main()外用默认值生成的static girth <double>
   printf(“%lf\n”, girth<double, 4>);  //生成的模板实例变量不是main函数的局部自动变量
   printf("%lf\n", area<double>(4));
}
```

第13章 模板与内存回收

◆13.2 函数模板

- 函数模板是使用**类型形参**定义的函数框架，可根据**类型实参**生成函数模板的**模板实例函数**。
- 模板（变量模板、函数模板、类模板）的声明或定义只能在全局范围、命名空间或类范围内进行。
- 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**。
- 在函数模板时，可以使用类型形参和非类型形参。
- 模板实例化时非类型形参需要传递常量作为实参。
- **可以单独定义类的函数成员为函数模板。**

第13章 模板与内存回收

【例13.3】定义用于变量交换和类型转换的两个函数模板

```
template <class T, int m=0>      //class可用typename代替
void swap(T& x, T& y=m)
{
   T  temp = x;
   x = y;
   y = temp;
}

template <class D, class S>                    //class可用typename来代替定义形参D、S
D convert(D& x, const S& y)                    //模板形参D、S必须在函数参数表中出现
{
   return x = y;                        //将y转换成类型D后赋给x
}
```

非类型形参，可以给默认值

可以有多个类型参数

第13章 模板与内存回收

```
struct A {
    double i, j, k;
public:
    A(double x, double y, double z) :i(x), j(y), k(z) { };
};
void main(void){
    long x = 123, y = 456;
    char a = 'A', b = 'B';
    A  c(1, 2, 3), d(4, 5, 6);
    swap<long, 0>(x, y);     //显式给出类型实参，必须用常量传给非类型实参m，函数模板实例化得到实例函数
    swap(x, y);       //自动生成实例函数void swap(long &x, long &y)，通过实参推断类型形参T，隐式实例化函数模板
    swap(a, b);       //自动生成实例函数void swap(char &x, char &y)，隐式给出类型实参char，m=0
    swap(c, d);       //自动生成实例函数void swap(A &x, A &y)
    convert(a, y);    //自动生成实例函数char convert (char &x, const long &y)
}
```

第13章 模板与内存回收

【例13.4】单独定义函数成员为模板

```
#include<typeinfo>
class ANY {                                  //定义一个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) {//单独定义构造函数模板
      p = new T(x);
      t = typeid(T).name();
   }
   void* P() { return p; }
   const char* T() { return t; }             //此T为函数成员的名称，不是模板的类型形参
   ~ANY() noexcept { if (p) { delete p;  p = nullptr; } }
}a(20);                                      //自动从构造函数模板生成构造函数ANY::ANY(int) ,通过实参类型推断
void main(void){
   double* q(nullptr);          //等价于“double* q=nullptr;”
   if (a.T() == typeid(double).name()) q = (double*)a.P(); //a.P()返回void *
}
```

第13章 模板与内存回收

◆13.2 函数模板（课堂略）

- 函数模板的类型形参允许参数个数可变，用省略参数“…”表示任意个类型形参
- 用递归定义的方法可展开并处理这些类型形参。
- 生成实例函数时，可能因递归生成多个实例函数，这些实例函数之间递归调用。

第13章 模板与内存回收

【例13.5】定义任意个类型形参的函数模板。

```
#include <iostream>
using namespace std;
template<class …Args>                    //类型形参个数可变的函数模板声明
int  println(Args …args);

template<class …Args>                    //递归下降展开的停止条件：println()的参数表为空
int  println( ) {
   cout << endl;
   return 0;
}

template < class H, class…T>             //递归下降展开println()的参数表
int  println(H h, T…t) {
   cout << h<<”  *”  ;
   return 1+println(t…);                 //递归下降调用
}
int main( ) {
   int n= println(1, ‘2’, 3.3, “expand”); //根据实参，递归的推断类型实参并展开，生成实例函数（递归调用）
   return n;                             //返回n=4
}
```

第13章 模板与内存回收

◆13.3 函数模板实例化

- 在调用函数时可隐式自动生成模板实例函数（根据实参推断）。
- 也可使用“template 返回类型 函数名<类型实参>(形参列表)”，显式强制函数模板按类型实参显式生成模板实例函数。
- 有时候生成的模板实例函数不能满足要求，可定义特化的模板实例函数隐藏自动生成的模板实例函数。
- 在特化定义模板的实例函数时，一定要给出特化函数的完整定义。
- 一般的比较为浅比较，当涉及字符串运算时，应通过特化实现深比较。

第13章 模板与内存回收

显式强制函数模板按类型实参显式生成模板实例函数

```
template <class T>          //class可用typename代替
 T sum(const T& x, const T& y)
 {
    return x + y;
 }

 //显式实例化函数模板，得到实例函数sum<int>
 template int sum<int>(const int & x, const int &y);
 int i = sum<int>(1,2); //调用函数sum<int>
```

第13章 模板与内存回收

【例13.8】模板实例函数的生成以及隐藏。

```
#define _CRT_SECURE_NO_WARNINGS      //防止strcmp出现指针使用安全警告
#include <string.h>                  //定义函数模板max()
template <typename T>
T max(T a, T b)
{
   return a>b?a:b;
}
template < >                         //此行可省， template < >表示特化实例函数：特化函数将被优先调用
const char *max(const char *x, const char *y)  //特化函数：用于隐藏模板实例函数
{
   return strcmp(x, y)>0?x:y;                  //进行字符串内容比较
}
```

第13章 模板与内存回收

```
int u, v;
int greed(int x, int y=max(u, v))  //产生默认值时生成模板实例函数int max(int, int)
{
   return x*y;
}
void main(void)
{
   const char *p="ABCD";                    //字符串常量"ABCD"的默认类型为const char*
   const char *q="EFGH";
   p=max(p, q);                        //调用特化定义的实例函数，进行字符串内容比较

}
```

第13章 模板与内存回收

◆13.4 类模板

- 类模板也称为类属类或类型参数化的类，用于为相似的类定义一种通用模式。
- 编译程序根据类型实参生成相应的类模板实例类
- 类模板既可包含类型参数，也可包括非类型参数。
- 类型参数可以包含1个以上乃至任意个类型形参。
- 非类型形参在实例化是必须使用常量做为实参。

第13章 模板与内存回收

【例13.10】定义向量类的类模板。

```
template <class T, int v=20>      //类模板的模板参数列表有非类型形参v，默认值为20
class VECTOR
{
   T  *data;
   int  size;
public:
   VECTOR(int n = v+5);                    //由v构成表达式v+5，将其作为构造函数成员形参的默认值
   ~VECTOR( ) noexcept;
   T &operator[ ](int);
};

template <class T, int v>              //在类体外函数实现时，加template<class T, int v>, v不能给缺省值
VECTOR <T, v>::VECTOR(int n) //须用VECTOR <T, v>作为类名
{   data = new T[size = n]; }
```

第13章 模板与内存回收

```
template <class T, int v>  //函数实现时，加template<class T, int v>, v不能给缺省值
VECTOR <T, v>::~VECTOR( ) noexcept      //须用VECTOR <T, v>作为类名
{
   if (data) delete data;
   data = nullptr;
   size = 0;
}

template <class T, int v>
T &VECTOR <T, v>::operator[ ](int i)                    //须用VECTOR <T, v>作为类名
{
   return data[i];
}
```

第13章 模板与内存回收

```
void main(void)
{
   VECTOR<int> LI(10);                    //定义包含10个元素的整型向量LI
   VECTOR<short> LS;                      //定义包含25个元素的短整型向量LS
   VECTOR<long>          LL(30);          //定义包含30个元素的长整型向量LL
   VECTOR<char*> LC(40);                  //定义包含40个元素的字符型向量LC
   VECTOR<double, 10> LD(40);             //非类型形参必须使用常量作为实参
   VECTOR<int> *p = &LI;                  //定义指向整型向量的指针p

   //以下q指向40个由VECTOR<int>元素构成的数组，其中
   //每个VECTOR<int>元素又是包含25个int元素的向量
   VECTOR<VECTOR<int>>*q=new VECTOR<VECTOR<int>>[40];
   delete q;
}
```

第13章 模板与内存回收

◆13.4 类模板

- 可以用类模板定义基类和派生类。
- 在实例化派生类时，如果基类是用类模板定义的，也会同时实例化基类。这里的实例化指模板类实例化，生成实例类
- 派生类函数在调用基类的函数时，最好使用“基类<类型参数>::”限定基类函数成员的名称，以帮助编译程序识别函数成员所属的基类。
- 对于实例化的模板类，如果其名字太长，可以使用typedef重新命名定义。

第13章 模板与内存回收

【例13.11】类模板的派生用法。

```
template <class T>                                    //定义基类的类模板
class VECTOR
{
   T* data;
   int  size;
public:
   int getsize( ) { return size; };
   VECTOR(int n) { data = new T[size = n]; }; //在类体里实现函数不用再加模板形参
   ~VECTOR( ) noexcept { if(data) {delete[ ]data; data=nullptr; size=0;}};
   T& operator[ ](int i) { return data[i]; };
};
```

第13章 模板与内存回收

【例13.11】类模板的派生用法。

```
template <class T>                          //定义基类的类模板
class VECTOR
{
   T* data;
   int  size;
public:
   int getsize( ) { return size; };
   VECTOR(int n) { data = new T[size = n]; };
   ~VECTOR( ) noexcept { if(data) {delete[ ]data; data=nullptr; size=0;}};
   T& operator[ ](int i) { return data[i]; };
};
template <class T>                          //定义派生类的类模板
class STACK : public VECTOR<T> { //派生类类型形参T作为实参，实例化VECTOR<T>
   int top;
```

第13章 模板与内存回收

调用基类的函数时，最好使用“基类<类型参数>::”限定基类函数成员的名称

```
public:
   int full( ) { return top==VECTOR<T>::getsize( ); }
   int null( ) { return top==0; }
   int push(T t);
   int pop(T& t);
   STACK(int s): VECTOR<T>(s) { top = 0; };
   ~STACK( ) noexcept { };
};
template <class T>
int STACK<T>::push(T  t)
{
   if (full( )) return 0;
   (*this)[top++] = t;  //调用operator[]
   return 1;
 }
```

父类继承的operator[]

第13章 模板与内存回收

```
template <class T>
int STACK<T>::pop(T& t)
{
   if (null( )) return 0;
   t = (*this)[--top];
   return 1;
}
void main(void)
{
   typedef  STACK<double>  DOUBLESTACK; //对实例化的类重新命名定义
   STACK <int>    LI(20);   //类模板实例化得到实例类STACK<int>（及其父类VECTOR<int>）
   STACK <long>  LL(30);
   DOUBLESTACK            LD(40);  //等价于 “STACK<double> LD(40);”
}
```

第13章 模板与内存回收

【例13.12】类模板中的多个类型形参的顺序变化对类模板的定义没有影响。

```
template<class T1, class T2> struct A {                    //定义类模板A
   void f1( );
   void f2( );
};
template<class T2, class T1> //正确：A<T2, T1>同类型形参一致
void A<T2, T1>::f1( ){ }

template<class T1, class T2> //正确：A<T1, T2>同类型形参一致
void A<T1, T2>::f2( ){ }

//template<class T2, class T1> //错误：A<T1, T2>与类型形参不同
void A<T1, T2>::f2( ){}
template<class...Types> struct B {
   void f3( );
   void f4( );
};
template<class ...Types> void B<Types ... >::f3( ){ }   //正确：Types表示类型形参列表
template<class ...Types> void B<Types ... >::f4( ){ }   //正确：Types表示类型形参列表
//template<class ...Types> void B<Types>::f4( ){ }      //错误：必须用“Types ...”的形式
void main(void) { }
```

第13章 模板与内存回收

◆13.4 类模板

- 当一个类模板有多个类型形参时，在类中只要使用同样的类型形参顺序，都不会影响他们是同一个类型形参模式。
- 当使用“…”表示类型形参时，表示类型形参有任意个类型参数。可通过递归定义展开类模板。【例13.13】

```
//通过auto定义变量模板n，元素全部初始化为0
template <class T> auto n = new VECTOR<T>[10]{};

//通过auto定义变量模板p
template <class T> static auto p = new VECTOR<T>[10];

//定义Lambda表达式模板
template <class T> auto q = [ ](T& x)->T& { return x; };
```

创建大小为10的数组，每个元素类型为VECTOR<T>

第13章 模板与内存回收

```
template <class T>                                    //定义类模板
   class VECTOR
   {
      T* data; int  size;
   public:
      int getsize( ) { return size; };
      VECTOR(int n = 10) { data = new T[size = n]; }; //在类体里实现函数不用再加模板形参
      ~VECTOR( ) noexcept { if(data) {delete[ ]data; data=nullptr; size=0;}};
      T& operator[ ](int i) { return data[i]; };
   };
template <typename T> auto m = new T[10];
//显式将变量模板实例化，产生变量m<int>。可以注释下面一行，直接使用m<int>(隐式实例化),特别是使用auto时
template int * m<int>;

//通过auto定义变量模板，元素全部初始化为0
template <class T> auto n = new VECTOR<T>[10]{ };

//显式将变量模板实例化，产生变量n<int>。可以注释下一行，直接使用n<int>(隐式实例化),特别是使用auto时
template VECTOR<int>*  n<int>;

//定义Lambda表达式模板，可以直接（实际上只能）隐式实例化Lambda表达式模板，如q<int>(i)
template <class T> auto q = [ ](T& x)->T& { return x; };
```

定义变量模板

第13章 模板与内存回收

#include <cxxabi.h> //gcc

```
const char *unreadable_type_name_m = typeid(m<int>).name(); //隐式实例化变量模板，得到变量m<int>
char *type_name_m = abi::__cxa_demangle(unreadable_type_name_m, nullptr, nullptr, nullptr);

cout << m<int> << "," << typeid(m<int> ).name() << endl; //gcc下输出：0xf12200, Pi
cout << m<int> << "," << type_name_m << endl; //gcc下输出：0xf12200, int *
```

输出指针的值还有内部的符号Pi

abi::__cxa_demangle 将低级符号名解码(demangle)成用户级名字，现在输出int *了

```
const char *unreadable_type_name_n = typeid( n<int> ).name();
char *type_name_n = abi::__cxa_demangle(unreadable_type_name_n, nullptr, nullptr, nullptr);

cout << n<int> << "," << typeid(n<int>).name() << endl; //gcc下输0xdf2238,PN15class_template36VECTORIiEE
cout << n<int> << "," << type_name_n << endl; //gcc下输出：0xdf2238,class_template3::VECTOR<int>*

int i = 3;
int j = q<int>(i); //前面显式初始化Lambda表达式模板，得到Lambda表达式实例q<int>，并调用
cout << “j = ” << j << endl; //输出3
const char *unreadable_type_name_q = typeid(q<int>).name();
char *type_name_q = abi::__cxa_demangle(unreadable_type_name_q, nullptr, nullptr, nullptr);
cout << typeid(q<int>).name() << endl; //gcc下输出：1N15class_template31qIiEUlRiE_E
cout << type_name_q << endl; //gcc下输出：1class_template3::q<int>::{lambda(int&)#1}
```

第13章 模板与内存回收

◆13.5 类模板的实例化及特化

- 可采用“template 类名<类型实参列表>”的形式显示直接实例化类模板。参见例13.14中的：template A <int>;。
- 也可在变量、函数参数、返回类型等定义时实例化类模板。
- 实例化生成的实例类同类模板的作用域相同。使用非类型形参的模板必须用常量作为非类型形参的实参。
- 当实例化生成的类实例、函数成员实例不合适时，可以自定义（特化）类、函数成员隐藏编译自动生成的类实例或函数成员。

第13章 模板与内存回收

【例13.16】定义特化的字符指针向量类来隐藏通过类模板自动生成的字符指针向量类。

```
#include <iostream>
using namespace std;
template <class T>
class VECTOR                                        //定义主类模板
{
   T*  data;
   int  size;
public:
   VECTOR(int n): data(new T[n]), size(data?n:0){ };
   virtual ~VECTOR( ) noexcept {                    //实例类的析构函数将成为虚函数
      if(data){ delete data; data=nullptr; size=0; }
      cout<<"DES O\n";
   };
   T& operator[ ](int i) { return data[i]; };
};
```

第13章 模板与内存回收

```
template < >                              //定义特化的字符指针向量类
class VECTOR <char*>
{
   char** data;
   int  size;
public:
   VECTOR(int);                           //特化后其所属类名为VECTOR <char*>
   ~VECTOR( ) noexcept;                   //特化后其所属类名为VECTOR <char*>，不是虚函数
   virtual char*& operator[ ](int i) { return data[i]; };     //特化后为虚函数
};
VECTOR <char*>::VECTOR(int n)  //使用特化后的类名VECTOR <char*>
{
   data = new char* [size = n];
   for (int i = 0; i < n; i++) data[i] = 0;
}
```

如果类模板定义了虚函数，由编译器自动实例化产生的实例类的对应函数是虚函数。但如果自己定义特化类，可以修改为非虚函数。
特化就是自己定义实例类，不是由编译器自动产生。

第13章 模板与内存回收

```
VECTOR <char*>::~VECTOR( ) noexcept //使用特化后的类名VECTOR <char*>
{
   if(data==nullptr) return;
   for(int i=0; i<size; i++)   delete data[i];
   delete data;       data=nullptr;
   cout << "DES C\n";
}
class A : public VECTOR<int> { //VECTOR<int>是类模板的实例类，由编译器自动产生，A继承VECTOR<int>
public:
   A(int n): VECTOR<int>(n) { }; //调用父类VECTOR<int>的构造函数
   ~A( ){ cout << "DES A\n"; }          //自动成为虚函数：因为基类析构函数是虚函数
};
class B : public VECTOR<char*> {//VECTOR<char *>是自定义的特化实例类，B继承VECTOR<char *>
public:
   B(int n): VECTOR<char*>(n) { };
   ~B( ) noexcept { cout << "DES B\n"; }//特化的VECTOR<char *>析构函数不是虚函数，故~B也不是
};
```

第13章 模板与内存回收

```
void main(void)
{
    VECTOR <int>        LI(10);              //自动生成的实例类VECTOR <int>
    VECTOR <char*>      LC(10);     //优先使用特化的实例类VECTOR<char*>
    VECTOR<int>* p = new VECTOR<int>(3);
    delete p;
    p = new A(3);           //VECTOR<int>是A的父类，父类指针指向子类对象
    delete p;
    VECTOR<char*> *q = new VECTOR<char*>(3);
    delete q;
    q = new B(3);           //VECTOR<char *>是B的父类，父类指针指向子类对象
    delete q;
}
```

除了可以特化整个类外，还可以仅特化部分函数成员，见例13.18中的：

```
template < >  VECTOR <char*>::~VECTOR( ) noexcept{  }
```

第13章 模板与内存回收

◆13.5 类模板的实例化及类型推导

- 在定义类模板时，可以使用类模板的**类型形参**进行类型推导。
- 实例类可以作为基类
- 实例函数可以成为类的友元函数

例13.19 定义模板类实例作为基类，模板函数实例作为友元
定义Lambda模板和自动类型推导

```
template<class T>
  class A{   //定义主类模板
    T i;
  public:
    A(T x){ i = x;}
    virtual operator T(){ //重载强制类型转换()
      auto x = [this]()->T {return this->i;}; //定义Lmabda表达式模板，必须显式捕获this，返回this->i
      return x(); //返回调用Lambda得到的值
//      return i; //上面代码只是演示Lambda表达式。直接返回i也可。
    }
  };

  template<class T>
  void output( T &x){ cout << x.k;} //定义函数模板


  //定义派生类B时，自动生成实例类A<int>作为B的父类
  class B:public A<int>{
    int k;
    friend void output<B>(B &); //生成模板函数实例并将其作为B的友元
  public:
    B(int x):A<int>(x) { k = x;} //必须显式调用父类构造
    operator int(){ return k + A<int>::operator int();} //自动成为虚函数
  };
```

```
void test(){
    A<int> a(4); //自动从模板类生成实例类A<int>
    auto b = a; //b的类型推导为A<int>
    auto c = a.operator int(); //c的类型： int
    auto d = A<double>(5); //d的类型为A<double>
    //e为double，自动调用 A<double>::operator double()
    auto e = 1 + A<double>(5);
    B f(6); output(f);
  }
```

第13章 模板与内存回收

◆13．5　类模板的实例化及特化

- 类模板或类模板的实例均可以定义实例成员指针。见如下例13.20。
- 实例化类模板时，类模板中的成员指针类型随之实例化。
- 当使用类模板的实例化类，作为类模板实例化的实参时，会出现嵌套的实例化现象。原本没有问题的类型形参T，用实参int实例化new T[10]时没有问题；但在嵌套实例化时，若用A<int>实例化new T[10]时，则会要求类模板A定义定义无参构造函数A<int>::A()。
- 若类模板中使用非类型形参，实例化时使用表达式很可能出现“>”，导致编译误认为模板参数列表已经结束，此时可用“（）”如List<int, (3>2)> L1(8);，见例子13.21。
- 类模板常用在STL标准类库等需要泛型定义的场合。见如下例13.20；注意其中类型转换static_cast等的用法。

第13章 模板与内存回收

【例13.20】实例类的实例成员指针和静态成员指针的用法。

```
template <class T, int n=10>
struct A {
   static T t;
   T  u;
   T* v;
   T  A::* w;                  //实例成员指针指向A的T类型成员
   T  A::* A::* x;             //实例成员指针指向A的T A::*类型成员
   T  A::** y;                 //普通指针y指向T A::*类型成员
   T* A::* z;                  //实例成员指针指向A的T* 类型成员
   A(T k=0, int h=n);                          //因A()被调用，故必须定义A()，等价于调用A(0, n)
   ~A() { delete[ ]v; }
};
template <class T, int n>  T A<T, n>::t = 0;       //类模板静态成员的初始化
```

第13章 模板与内存回收

```
template <class T, int n>
A<T, n>::A(T k, int h)                    //不得再次为h指定默认值，因为其类模板已指定
{
   u = k;
   v = new T[h];                          //初始化数组对象，必须调用无参构造函数T()
   w = &A::u;
   x = &A::w;
   y = &w;
   z = &A::v;
    v = &A::t; // &A::t：取静态成员t的地址，因此v是普通指针
}
template  struct  A<double>;              //从类模板生成实例类A<double,10>
```

第13章 模板与内存回收

```
void main(void)
{
   A<int> a(5);                                  //等价于“A<int,10> a(5);”
   int u = 10, * v = &u;
   int A<int>::* w = &A<int>::u;                 //等价于“int A<int, 10>::* w;”
   int A<int>::*  A<int>::* x = &A<int>::w;
   int A<int>::** y = &w;
   int* A<int>::* z = &A<int>::v;
   v = &A<int>::t;
   v = &a.u;
   y = &a.w;
   //使用类模板的实例化类A<int>，作为类模板A(外面一层A）实例化的类型实参时，出现嵌套的实例化现象
   A<A<int>> b(a);     //等价于“A<A<int,10>, 10> b(a);”，构造b时调用  A<int>::A()构造a

   A<int>  A<A<int>>::*c= &A<A<int>>::u;
   a = b.*c;
   A<int> A<A<int>>::* A<A<int>>::*d = &A<A<int>>::w;
}
```

第13章 模板与内存回收

【例13.22】定义用两个栈模拟一个队列的类模板

```
#include <iostream>
using namespace std;
template <typename T>                          //定义主类模板
class STACK {
   T* const  elems;                            //申请内存，用于存放栈的元素
   const int   max;                            //栈能存放的最大元素个数
   int   pos;                                  //栈实际已有元素个数，栈空时pos=0;
public:
   STACK(int m=0);                             //等价于定义了STACK(),防嵌套实例化出问题
   STACK(const STACK& s);                      //用栈s初始化p指向的栈
   STACK(STACK&& s)noexcept;                   //用栈s初始化p指向的栈
   virtual T operator [ ] (int x)const;        //返回x指向的栈的元素
   virtual STACK& operator<<(T e);             //将e入栈，并返回p
   virtual STACK& operator>>(T& e);            //出栈到e，并返回p
   virtual ~STACK( )noexcept;                  //销毁p指向的栈
   ……
};
```

第13章 模板与内存回收

【例13.22】定义用两个栈模拟一个队列的类模板

```
template <typename T>
STACK<T>::STACK(STACK&& s) noexcept: elems(s.elems), max(s.max), pos(s.pos){
   const_cast<T*&>(s.elems) = nullptr;   // 等价于*(T**)&(s.elems) = nullptr;
   const_cast<int&>(s.max) = s.pos=0;
}
template <typename T>
class QUEUE: public STACK<T> {
   STACK<T>  s2;                                          //队列首尾指针
 public:
   QUEUE(int m=0);                                        //初始化队列：最多m个元素
   QUEUE(const QUEUE& s);                                 //用队列s复制初始化队列
   QUEUE(QUEUE&& s) noexcept;                             //移动构造
   virtual QUEUE& operator=(QUEUE&& s) noexcept;          //移动赋值
   ~QUEUE( )noexcept;                                     //销毁队列
   ……
};
```

第13章 模板与内存回收

【例13.20】定义用两个栈模拟一个队列的类模板

```
template <typename T>
//以下初始化一定要用move，否则QUEUE是移动赋值而其下层是深拷贝赋值
QUEUE<T>::QUEUE(QUEUE&& s) noexcept: STACK<T>(move(s)), s2(move(s.s2)) { }
template <typename T>
QUEUE<T>& QUEUE<T>::operator=(QUEUE<T>&& s) noexcept {
   //以下赋值一定用static_cast，否则QUEUE是移动赋值而其下层是深拷贝赋值
   *(STACK<T>*)this = static_cast<STACK<T>&&>(s);
   //等价于STACK<T>::operator=(static_cast<STACK<T>&&>(s));
   //或等价于STACK<T>::operator=(std::move(s));
   s2 = static_cast<STACK<T>&&>(s.s2);
   //等价于“s2=std::move(s.s2);”，可用“std::move”代替“static_cast<STACK<T>&&>”
   return *this;
}
```

第13章 模板与内存回收

◆13.5 类模板的实例化及特化

- 为解决内存泄漏问题，可像Java那样定义一个始祖基类Object。
- 所有其他类都从Object继承，比如Name。参见例13.21。
- 定义一个Type类模板，用于管理类Name的对象引用计数，若对象被引用次数为0，则可析构该对象。Type类模板的构造函数使用Name*作为参数，一遍所有Name的对象都是通过new产生的。Type类模板的赋值运算符重载函数负责对象的引用计数。
- 当要使用Name产生对象时，可用Name作为类模板Type的类型实参，产生实例化类，然后使用该实例化类。