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

- C++提供了3种类型的模板，即变量模板、函数模板、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两种：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、类名或类模板实例代替T。

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(3.1415926L);    //定义变量模板pi，其类型形参为T
等价: const T pi = T(3.1415926L);
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>);  //打印 pi<T>的模板实例变量的地址
   return pi<T> * r * r;
}
//显式生成模板的匿名实例变量pi<float>，即 float (3.1415926L)
template const float pi<float>;
template constexpr float pi<float>;   //error, 实例化不能使用constexpr
template float pi<float>;             //error, 与模板类型 const T 不匹配
//显式生成模板的匿名实例变量pi<double>，即 double (3.1415926L)
template const double pi<double>;
```

第13章　模板与内存回收

```
void main(void)
{
   const float &d1 = pi<float>;                    //引用已经生成的模板实例变量 pi<float>
   printf("%p\n", &d1);
   const double &d2 = pi<double>;                  //引用已经生成的模板实例变量 pi<double>
   printf("%p\n", &d2);
   const long double &d3 = pi<long double>;  //生成新的模板实例变量 pi<long double>
   printf("%p\n", &d3);
   float a1 = area<float>(3);                      //在函数模板的实例中显示d1的地址
   double a2 = area<double>(3);                    //在函数模板的实例中显示d2的地址
   long double a3 = area<long double>(3); //在函数模板的实例中显示d3的地址
   int a4 = area<int>(3);                          //在函数模板的实例中隐式生成模板实例变量 pi<int>
}
```

第13章　模板与内存回收

◆13.1 变量模板

- 变量模板不能在函数内部声明。
- 显式或隐式实例化生成的模板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类型形参外，还可以有非类型的形参,非类型形参可以定义默认值。实例化变量模板时，非类型形参需要传递常量作为实参。
- 可使用“template 类型 模板名<类型实参>”，显式生成匿名的实例变量。

第13章　模板与内存回收

```
template<class T, int x=0>
T g = T(10 + x);

//生成匿名实例变量 g<float, 0>
template float g<float>;

//将匿名实例变量 g<float, 0>的值拷贝给g1
float  g1 = g<float>;
float  &g2 = g<float>;
const float &g3 = g<float>;
const float &g4 = g<float, 0>;
const float &g5 = g<float, 4>;
const float &g6 = g<float,
sizeof(printf("A"))>;  //去掉sizeof ?
```

```
int main(void)
{   //将匿名实例变量 g<float, 0>的值拷贝给a1
    float a1 = g<float>;
    float &a2 = g<float>;
    const float &a3 = g<float>;
    const float &a4 = g<float, 0>;
    const float &a5 = g<float, 4>;
    //&g2=&g3=&g4=&a2=&a3=&a4,
    //&g5=&g6=&a5,
    //&g1!=&g2, &a1!=&a2
    //g5=g6=a5=14, other variables = 10
    a2 = 1;   //g2=g3=g4=a2=a3=a4=1
    //a3 = 100;  //error
}
```

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。
- 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。
- 在函数模板中，可以使用类型形参和非类型形参。
- 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
}
```

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);  //必须用常量传给非类型实参m,  ⇔ ::swap(x, y)
    std::swap(x, y);        //自动生成实例函数void swap(long &x, long &y)
    std::swap(a, b);        //自动生成实例函数void swap(char &x, char &y)
    std::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, 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)

void main(void) {
    double *q(nullptr);
    if (a.T() == typeid(double).name()) q = (double *)a.P();
}
```

第13章　模板与内存回收

◆13.2 函数模板

- 函数模板的类型形参允许参数个数可变 。“…”表示任意个类型形参，并且各形参的类型可以不同。
- 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。
- VS2019暂时不支持用 export 导出函数模板。

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 H, class...T>
int  println(H h, T...t);
int  println();
int  println() { 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");
    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。
- 可使用 “template 返回类型 函数名<类型实参>(形参列表)”，显式将函数模板实例化。
- 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。模板函数与具体函数同名。
- 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

【例13.7】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，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，也可称为模板类或类模板实例。
- 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13.9】定义向量类的类模板。

```
//类模板的模板参数列表有非类型形参v，默认值为20
template <class T, int v=20>
class VECTOR
{
   T  *data;
   int  size;
public:
   VECTOR(int n = v+5);
   ~VECTOR( ) noexcept;
   T &operator[ ](int);
};

template <class T, int v>
VECTOR<T, v>::VECTOR(int n)   //必须用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>
VECTOR<T, v>::~VECTOR( ) noexcept
{
    if (data) delete [ ] data;
    data = nullptr;
    size = 0;
}
template <class T, int v>
T &VECTOR<T, v>::operator[ ](int i)
{
    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<int>  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 char * 向量LC
   VECTOR<double,10> LD(40);       //非类型形参必须使用常量作为实参, 40个元素
   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，会同时实例化基类。
- 派生类函数在调用基类的函数时，最好使用“基类<类型参数>::”访问基类的成员，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13.10】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

```
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;
    return 1;
 }
```

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);
   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13.11】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 {
   void f1( );
   void f2( );
};
template<class T2, class T1> void A<T2, T1>::f1( ) { }   //正确：A<T2, T1>同类型形参一致
template<class T1, class T2> void A<T1, T2>::f2( ) { }   //正确：A<T1, T2>同类型形参一致
//template<class T2, class T1>void A<T1, T2>::f2( ) { }  //错误：A<T1, T2>与类型形参不同
template<class...Types> struct B {   //可变类型形参
   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，元素全部初始化为0
template <class T> auto n = new VECTOR<T>[10] { };
template <class T> static auto p = new VECTOR<T>[10];   //通过auto定义变量模板
template <class T> auto q = [ ](T &x)->T & { return x; };    //定义Lambda表达式模板
```

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 T<类型实参列表>”的形式显式实例化类模板T。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。
- 实例化生成的类同类模板的作用域相同。实例化包含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13.15】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
{
   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;
}
```

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> {
public:
   A(int n): VECTOR<int>(n) { };
   ~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 *> {
public:
   B(int n): VECTOR<char *>(n) { };
   ~B( ) noexcept { cout << "DES B\n"; }  //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);
   delete p;
   VECTOR<char *> *q = new VECTOR<char *>(3);
   delete q;
   q = new B(3);
   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13.16中的:
//template < >  VECTOR <char *>::~VECTOR( ) noexcept { }
```

第13章　模板与内存回收

◆13.5 类模板的实例化及特化

- 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。
- 在用实例化类型产生对象作为初始化表达式时也可以进行类型推导。

```
template <class T>    //例3.17：定义主类模板
class A {
   T  i;
public:
   A(T x) { i = x; }
   virtual operator T( ) {  //重载运算符用于强制类型转换
      auto x = [this]( )->T { return i; };   //不能去掉this
      return x();
   }
};
```

```
A<int> a(4);  //自动从类模板A中实例化A<int>类
auto b = a;     //b的类型为A<int>
auto c = a.operator int( );  //c的类型为int
auto d = A<double>(5);     //d的类型为A<double>
```

第13章　模板与内存回收

◆13.5 类模板的实例化及特化

- 类模板中可以定义实例成员指针。见如下例13.18。
- 实例化类模板时，类模板中的成员指针类型随之实例化。
- 使用类模板的实例化类作为类模板实例化的形参时，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()。例13.18
- 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);。
- 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13.18】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;
    T  A::*A::* x;
    T  A::**y;
    T  *A::*z;
    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;          //&w 与 &A::w ?
   z = &A::v;
   v = &A::t;
}
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;
   A<A<int>> b(a);  //等价: A<A<int,10>, 10> b(a)，构造b时调用 A<int>::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13.20】定义用两个栈模拟一个队列的类模板

```
#include <iostream>
using namespace std;
template <typename T>
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，深拷贝
   STACK(STACK &&s) noexcept;                //用栈s初始化p指向的栈，移动构造
   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13.20】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) = 0;
   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;
   ……
};
```

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

```
//以下初始化一定要用std::move, 否则QUEUE是移动赋值而其下层是深拷贝赋值
template <typename T>
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，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。