模板,STL

Andrew Huang<bluedrum@163.com>

课程内容

- 模板(Template) 的概念
- 函数模板
- 类模板
- STL(标准模板库) 简介

模板概念

模板产生的背景

- 从软件工程角度来说,代码重用性越高越好.
- 重用机制
  - 源码级或目标文件级有:C函数库或C++的类库相当于重用代码的集合
  - 组件级共享有:DLL,ActiveX 控件等
  - 把代码拷贝一份,然后用文本编辑修改,也是重用一种,而且使用机率很高.
- C++的利用原有代码扩展新的功能的几种方法
  - 继承和组合方法实现了对象代码重用
  - 参数代类型(模板)源代码重用
- 模板机制极其类似手工编辑代码模式
  - 适合实现那一些代码算法大同小异,只是处理类型不同代码
  - 手工编辑用文本替换来完成¡±代码重用¡±
  - 模板机制相当于在代码要¡±修改¡±的位置,嵌入模板标记,在用户调用这些代码时,由编译器自动完成¡±替换¡±

使用模板的优点

- 由于使用模板代码是相当是编译时展开的. 因此运行速度远高于派生类的虚函数, 这一点类似于宏展开.
- 相于多语句宏, 模板函数或模板类的最大好处在编译时就能进行严格的类型和语法检查.
- 一个模板针对一种算法只需要一份代码. 只要一经测试, 就可以应用多种类型.
  - 模板函数可以用函数重载来等效实现. 但是函数重载意味着代码并没有重用, 只是函数共享一个名称, 各部分代码还要独立实现.
  - 而且随着参数增多, 要想完整等效模板代码, 函数重载要实现大量代码, 这在工程上无法接受这一点.

由于模板的语法出现较晚,现有的调试器无法对带模板的代码进行单步调试.所以对带模板代码很多是间接的方法进行调试.这对构造较复杂的应用程序是一个很大问题.因此模板适合编写较简单和重复的小算法,整体应用程序或者复杂算法完全用模板实现相当困难现有C++的编译器对模板的编译支持不是特别完善,有时一个小的符号错误带来几百个编译错误提示.而且真正的错误提示被淹没在巨大衍生错误之中,有的甚至没有出现.这样C++编译器的错误提示对定位几乎没有帮助,开发者可能需要对代码自行作逐行检查.这样效率相当低下

模板(template) 的定义

- **模板**利用一种完全通用的方法来设计函数或类而不必预先说明将被使用的每个对象的类型，利用模板功能可以构造相关的函数或类的系列，因此模板也可称为参数化的类型。它分为**函数模板**和**类模板**
- 模板就是把功能相似、仅数据类型不同的函数或类设计为通用的函数模板或类模板，提供给用户。
  - 如链表,每一种链表的操作几乎是完全一样的.所不同是操作的结点类型有细微的差别
- 模板就是一种参数化(parameterized)的类或函数，也就是类的形态(成员、方法、布局等)或者函数的形态(参数、返回值等)可以被参数改变
  - 这个参数可以是类型(基本类型,结构和类)

模板定义(2)

- 模板的作用远不只是用来替代宏。实际上，模板是泛化编程(Generic Programming) 的基础。所谓的泛化编程，就是对抽象的算法的编程，泛化是指可以广泛的适用于不同的数据类型。
- template<> 是模板的标志，在<> 中，是模板的参数部分。参数可以是类型，也可以是数值

二种传统方法实现的效果

- 假设要实现一个二个类型之间作判断，取中间一个最大值的算法max
- 假设用宏来实现上述功能
  - #define max(a,b) ((a)>(b)?(a):(b))
- 用宏实现的问题
  - 带来展开的问题，如 int a=3,b=3;max(a++,b--) 得到结果可能是错误的
  - 没有对类型检查，只是简单做文字展开

用重载函数实现

- 重载函数版本，这意味着开发者要对每一种类型都要实现，而且每新一个类，或结构就要加一个实现.

```
int max(int a,int b) { return ((a>b)?a:b);}

long max(long a,long b) { return ((a>b)?a:b);}

char max(char a,char b) { return ((a>b)?a:b);}

double max(double a,double b) {return ((a>b)?a:b);}

//两个类的比较,要求类 class aclass 必须重载>操作符
aclass &max(aclass &a,aclass &b) {return ((a>b)?a:b);}
...
```

函数模板

函数模板的定义

- 对于一般函数而言，函数形参的类型是固定的，当调用函数时，实参的类型要与被调函

数的形参类型保持一致，否则会出现类型不一致的错误。因此，对于功能相同而只是参数的类型不同的情况，也必须定义不同的函数来分别完成相应的功能 .

- 参见重载函数max的定义

- C++语言中提供的函数模板功能就是为解决以上问题而提出的。C++语言提供的函数模板可以定义一个对任何类型变量都可进行操作的函数，从而大大增强了函数设计的通用性。因为普通函数只能传递变量参数，而函数模板却提供了传递类型的机制。
- 在C++语言中，跟一般函数一样,使用函数模板的方法是先说明函数模板，然后实例化成相应的模板函数进行调用执行。

函数模板的定义(2)

- 函数模板的一般说明形式如下

```
template <类型形参表>
返回值类型 函数名（形参表）
{
   //函数定义体
}
```

- 在上面的定义形式中，< 参数形参表> 可以有一到若干个形参，各形参前必须加上 class 关键字，表示传递类型，当有多个形参时，各形参间用逗号分隔。从中可以看出，< 类型形参表> 中的每个形参就表示了一种数据类型。¡° 形参表¡± 中至少有一个形参的类型必须用< 类型形参表> 中的形参来定义。

max 的函数模板

- 下例中
  - template<class T> 表明这个模板只有一个类型参数class T.
  - 这个参数T 应用在参数a,b 和返回值上
  - 在编译过中，函数模板实例化后,class T 会被开发者设定的类型¡± 替换¡± 掉. 形成一个真正可以运行函数.

```
template <class T>
const T & max(const T &a, const T & b)
 {
   return ((a>b)?a:b);
}
```

函数模板实例化方法

- 函数模板要代入具体数据，整个函数才能执行. 这一过程称为实例化.
- 模板实例化的方法分为两种
  - 显式实例化
    - **模板实参必须是一个基本数据类型或用户已定义的数据类型。**
  - 隐式实例化
    - **编译系统将根据调用函数的实际参数的数据类型自动实例化函数模板，即调用函数中的实际参数既是常规意义上的函数实参，实际参数的数据类型也是函数模板的模板实参。**
- **函数模板只能采用隐式实例化**

max 的实例化

- 用数据实例化函数模板

```
#include <iostream.h>

template <class T>
T max(T a, T b)
{
	return a > b ? a : b;
}

void main()
{
	cout << "max(20, 30) = " << max(20, 30) << endl;
	cout << "max('t', 'v') = " << max('t', 'v') << endl;
	cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl;
}
```

函数模板可作普通类方法

- 类中的方法可以为函数模板，
- 在使用时直接调用就行

```
class CLinkList{
public:
    template<class T> CNode* search(const T& t)
{ ...}
    void show();
};

//实现代码
CLinkList list;
CNode * p_node;
if((p_node = List.search(12))!=NULL)
        p_node->print();
```

类模板

类模板的定义

- 类模板实际上就是函数模板的推广
- 说明类模板的一般格式为：

```
template <类型形参表>
class  类模板名
{
   private：
     私有成员定义
   protected：
     保护成员定义
   public：
     公有成员定义
};
```

类模板的定义(2)

- <类型形参表>中可以包括一到若干个形参，这些形参既可以是”类型形参”，也可以是”表达式形参”。每个类型形参前必须加class关键字，表示对类模板进行实例化时代表某种数据类型
  - 类型形参是在类模板实例化时传递数据类型用的 ,就是所谓的显式实例化
  - 表达式形参的类型是某种具体的数据类型，当对类模板进行实例化时，给这些参数提供的是具体的数据 ,就是所谓隐式实例化
- <类型形参表>中的参数有多个时，需用逗号隔开

```
//用逗号隔开多个类型
template <class arg1,int arg2,class arg3>
class myclass
{
  //类的定义体
};
```

类模板的定义(3)

- 类模板中成员函数可以放在类模板的定义体中（此时与类中的成员函数的定义方法一致）定义，也可以放在类模板的外部来定义，此时成员函数的定义格式如下：
  - 其中：类模板名即是类模板中定义的名称；
  - 类型名表即是类模板定义中的< 类型形参表> 中的形参名。
  -

```
template <类型形参表>
函数值的返回值 类模板名<类型名表>::成员函数（形参）
{ 函数体 }
```

类模板的定义(4)

- 类模板定义只是对类的描述，它本身还不是一个实实在在的类，是类模板。
- 类模板不能直接使用，必须先实例化为相应的模板类，定义模板类的对象（即实例）后，才可使用。可以用以下方式创建类模板的实例。
  - 此处的< 类型实参表> 要与该模板中的< 类型形参表> 匹配，也就是说，实例化中所用的实参必须和类模板中定义的形参具有同样的顺序和类型，否则会产生错误。
  - 注意这里引用<>，这是与函数模板最大不同

```
类模板名<类型实参表> 对象名表;
```

类模板实例

```
template <class T>
class Stack
{
public:
        Stack(int = 10);
        ~Stack()
        {
                delete [] stackPtr;
        }
        int push(const T&);
        int pop(T&);
        int isEmpty() { return top == -1; }
        int isFull() { return top == size - 1; }
private:
        int size;       //Stack 中的元素数
        int top;
        T* stackPtr;
};
```

可以存放任何对象的栈

stack 类模板实例化

```
#include <iostream>
#include  "stack.h"
using namespace std;

void main()
{
  Stack< int > intStack;
  intStack.push(10);

  Stack< char > charStack;
charStack.push('a');
}
```

模板的混和定义

- template<> 是模板的标志，在<>中，是模板的参数部分。参数可以是类型，也可以是数值

```
template<class T, T t>
class Temp{
public:
	Temp(){}
	void print() { cout << t << endl; }
private:
	T t_;
};

void main()
{
Temp<int, 10> temp; //  合法
int i = 10;
Temp<int, i> temp; //  不合法,必须为常量

const int j = 10;
Temp<int, j> temp2; //  合法
}
```

类模板实例2

```
template<class C>          //函数模板
void square(C num, C *result){
	*result = num * num;
}

template<class T>      //类模板
class Array {
public:
	int getlength() {return length;}
	T & operator[](int i) {return array[i];}
	Array(int l) {
		length = l;
		array = new T[length];
	}
	~Array() {delete [] array;}
private:
	int length;
	T *array;
};
```

类模板实例2

```
//: STEMP.CPP
//-- Simple template example
#include <iostream.h>
#include <assert.h>

template<class T>
class array {
   enum { size = 100 };
   T A[size];
public:
   T& operator[](int index) {
      assert(index >= 0 && index < size);
      return A[index];
   }
};
main() {
   array<int> ia;
   array<float> fa;
   for(int i = 0; i < 20; i++) {
      ia[i] = i * i;
      fa[i] = float(i) * 1.414;
   }
   for(i = 0; i < 20; i++)
      cout << i << ": " << ia[i]
           << ", " << fa[i] << endl;
}
```

类模板常量

- 模板中常量(模板参数可以类参数,也可以是内置类型,可有默认值)
- 模板类在实例化时类参数用类来替换,内置类型用常量来替换

```
template<class T, int size =100>
class array {
   //enum { size = 100 };
   T A[size];
public:
   T& operator[](int index) {
      assert(index >= 0 && index < size);
      return A[index];
   }
};
```

类模板和继承

- 类模板可以做基类，其派生类也是模板类

```
template<class T>
class DriArray :public array<T>{
public:
   int Add(T* e,int idx)
  {
      assert(idx >= 0 && idx < size);
          A[idx] = e;
      return 0;
   }
};
```

类模板和继承(2)

- 一个模板类可以从一个非模板类的继承出来

```
class CBase
{
        //...
};
template<class T> class TDerived:public CBase
{
public:
        T t;
        //...
};
```

语法检查

- 对模板的语法检查有一部分被延迟到使用时刻(类被定义，或者函数被调用)，而不是像普通的类或者函数在被编译器读到的时候就会进行语法检查。因此，如果一个模板没有被使用，则即使它包含了语法的错误，也会被编译器忽略
  - 在这个模板中，我假设了T这个类型是一个类，并且有一个print()方法(t.print())。我们在简介中的min模板中其实也作了同样的假设，即假设T重载了'>'操作符。
  - 因为语法检查被延迟，编译器看到这个模板的时候，并不去关心T这个类型是否有print()方法，这些假设在模板被使用的时候才被编译器检查。只要定义中给出的类型满足假设，就可以通过编译。

```
template<class T> class Temp{
public:
    Temp(const T & t): t_(t) {}
    void print() { t.print();}
private:
    T t_;
};
```

模板类静态成员

- 同一个模板类的不同实例，如果所包含静态成员的是属于不同空间.
- 这里模板A 中增加了一个静态成员，那么要注意的是，对于aint1 和adouble1 ，它们并没有一个共同的静态成员。而aint1 与aint2 有一个共同的静态成员(对adouble1

和adouble2 也一样) 。

```
template<class T> class A{ static char a_; };
A<int> aint1, aint2;
A<double> adouble1, adouble2;
```

STL 简介

什么是STL

- 标准模板库(STL)是一个C++编程库.
- 它的组件是高度参数化的,完全基于模板实现,用来实现链表,栈等.
- 使C++程序员能够进行通用的程序设计,随着模板引入C语言,有一些C语言也应用STL

STL 历史

- STL（Standard Template Library，标准模板库)是惠普实验室开发的一系列软件的统称。它是由Alexander Stepanov、Meng Lee和David R Musser在惠普实验室工作时所开发出来的。现在虽说它主要出现在C++中，但在被引入C++之前该技术就已经存在了很长的一段时间。
- 有趣的是，对于STL还有另外一种解释--STepanov & Lee，前者是指Alexander Stepanov，STL的创始人；而后者是Meng Lee，她也是使STL得以推行的功臣，第一个STL成品就是他们合作完成的。这一提法源自1995年3月，Dr.Dobb¡¯s Journal特约记者，著名技术书籍作家Al Stevens对Alexander Stepanov的一篇专访。

STL 的本质

- STL 的出发点很简单的，就是为减少各种通用数据结构重复开发量．用模板来实现链表，队列，栈等各各机制
- 从逻辑层次来看，在STL 中体现了泛型化程序设计的思想（generic programming ），引入了诸多新的名词,比如像需求（requirements ），概念（concept ），模型（model ），容器（container ），算法（algorithmn ），迭代子（iterator ）等。与OOP（object-oriented programming ）中的多态（polymorphism ）一样，泛型也是一种软件的复用技术。

STL 简介

- STL的代码从广义上讲分为三类:
  - algorithm（算法）
  - container（容器）
  - iterator（迭代器）
- 标准模板库(STL)中
  - 通用算法被实现为函数模板
  - 容器被实现为类模板
- 在C++标准中，STL被组织为下面的13个头文件：<algorithm>、<deque>、<functional>、<iterator>、<vector>、<list>、<map>、<memory>、<numeric>、<queue>、<set>、<stack>和<utility>。

STL容器类

- 容纳其他对象的类
- 包括
  - vector、list、deque、set、mutilset、map、multimap、hash set、hash multiset、hash map和hash multimap
- 其中每个类都是一个模板，能够被实例化容纳任意对象类型

STL 容器类

数据结构	描述	头文件
向量(vector)	连续存储的元素	<vector>
列表(list)	由节点组成的双向链表，每个结点包含着一个元素	<list>
双队列(deque)	连续存储的指向不同元素的指针所组成的数组	<deque>
集合(set)	由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序	<set>
多重集合(multiset)	允许存在两个次序相等的元素的集合	<set>

STL 容器类(2)

数据结构	描述	头文件
栈(stack)	后进先出的值的排列	<stack>
队列(queue)	先进先出的执的排列	<queue>
优先队列 (priority_queue)	元素的次序是由作用于所存储的值对上的某种谓词决定的的一种队列	<queue>
映射(map)	由{键，值}对组成的集合，以某种作用于键对上的谓词排列	<map>
多重映射(multimap)	允许键对有相等的次序的映射	<map>

STL 算法

- 用于操纵容器中所保存的数据

- 有的算法与容器类是分离的
  - 算法部分主要由头文件<algorithm>，<numeric>和<functional>组成。<algorithm>

是所有STL头文件中最大的一个（尽管它很好理解），它是由一大堆模版函数组成的，可以认为每个函数在很大程度上都是独立的，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并等等。<numeric>体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。<functional>中则定义了一些模板类，用以声明函数对象。

STL迭代器

- Iterator(又翻译成游标),将算法与容器分离的一种机制
- 允许程序顺序地遍历一个容器中的元素
- 有些迭代器（如istream和ostream迭代器）与容器无关
- 为什么有迭代器? 简单数据结构的一般用for 循环进行遍历.如用
  - for(p_node=p_head;p_node;p_node=p_node->next)
- 设计Iterator 是为了复杂数据结构也能用for 进行遍历

STL 的hello world

```
#include <vector> // STL 向量的头文件。这里没有".h"。
#include <iostream>
using namespace std;
char* szHW = "Hello World";

int main(int argc, char* argv[])
{
vector <char> vec; //声明一个字符向量 vector (STL 中的数组)
//为字符数组定义一个游标 iterator。
vector <char>::iterator vi;

char* cptr = szHW; // 将一个指针指向“Hello World”字符串
while (*cptr != '\0')
{ vec.push_back(*cptr); cptr++; }
// push_back 函数将数据放在向量的尾部。

// 将向量中的字符一个个地显示在控制台
for (vi=vec.begin(); vi!=vec.end(); vi++)
// 这是 STL 循环的规范化的开始——通常是 "!=" ， 而不是 "<"
// 因为"<" 在一些容器中没有定义。
// begin()返回向量起始元素的游标（iterator），end()返回向量末尾元素的游标（iterator）。
{ cout << *vi; } // 使用运算符 “*” 将数据从游标指针中提取出来。
cout << endl; // 换行

return 0;
}
```

push_back 是将数据放入 vector（向量）或 deque（双端队列）的标准函数。Insert 是一个与之类似的函数，然而它在所有容器中都可以使用，但是用法更加复杂。end()实际上是取末尾加一（取容器中末尾的前一个元素），以便让循环正确运行——它返回的指针指向最靠近

数组界限的数据。就像普通循环中的数组，比如 for (i=0; i<6; i++) {ar[i] = i;} ——ar[6]是不存在的，在循环中不会达到这个元素，所以在循环中不会出现问题。

STL 初始化

```
//程序：初始化演示
//目的：为了说明 STL 中的向量是怎样初始化的。
#include <cstring>
#include <vector>
using namespace std;
int ar[10] = { 12, 45, 234, 64, 12, 35, 63, 23, 12, 55 };
char* str = "Hello World";
int main(int argc, char* argv[])
{
vector <int> vec1(ar, ar+10);
vector <char> vec2(str, str+strlen(str));
return 0;
}
```

STL 的实例

```
//目的：理解带有数组下标和方括号的 STL 向量
#include <cstring>
#include <vector>
#include <iostream>
using namespace std;
char* szHW = "Hello World";
int main(int argc, char* argv[])
{
vector <char> vec(strlen(sHW)); //为向量分配内存空间
int i, k = 0;
char* cptr = szHW;
while (*cptr != '\0')
{ vec[k] = *cptr; cptr++; k++; }
for (i=0; i<vec.size(); i++)
{ cout << vec[i]; }
cout << endl;
return 0;
}
```

STL 排序程序

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void main(void)
{
vector<int> num; // STL 中的 vector 容器
int element;

// 从标准输入设备读入整数，
// 直到输入的是非整型数据为止
while (cin >> element)
num.push_back(element);

// STL 中的排序算法
sort(num.begin(), num.end());

// 将排序结果输出到标准输出设备
for (int i = 0; i < num.size(); i ++)
cout << num[i] << "\n";
}
```

VC++ 对STL 的支持

- 常见的输出警告:
  - warning C4786: '¡-¡-' : identifier was truncated to '255' characters in the debug information
  - 这是因为编译器在Debug 状态下编译时，把程序中所出现的标识符长度限制在了255 个字符范围内。如果超过最大长度，这些标识符就无法在调试阶段查看和计算了。而在STL 程序中大量的用到了模板函数和模板类，编译器在实例化这些内容时，展开之后所产生的标识符往往很长（没准会有一千多个字符！）
  - 你可以在文件开头加入下面这一行：#pragma warning(disable: 4786) 。它强制编译器忽略这个警告信息

使用模板的注意事项

模板编程要求

- 类模板或函数模板的实现必须**全部实现在头文件** 中，象普通的类把声明写在头文件中，把实现代码放在CPP 文件中，编译一定通不过.
  - 因为模板本质相当于编译器作文本替换，因此必须能让C++ 编译器能直接¡ ± 看到¡ ± 所有源码，直接看到的方法就是把全部实现写在一个头文件之中，这一点可以参考STL 的实现
- 如果是一个普通类方法引用了模板类，而这个普通类代码涉及到模板类的方法也必须放在头文件之中.

模板编程要求

- 只有经过实例化的函数模板或者是类模板才正常的函数或类．只有实例化后的类模板才能声明对象
  - 所以必须要是 ClassTemp<T> ObjVar 形式;
  - 以下是错误的使用类模板

```
template <class T>
class Node{
public:
  Node * p_next ;//错误的定义,
//类模板不能声明对象变量,只能是实例化后的类才能定义,因此正确定义如下
//Node<T> * p_next;
  T data;
}
```

模板编程要求

- 错误使用定义二

```
template <class T>
class Node{
public:
  Node<T> * p_next;
  Node(T d);
  T data;
}

template<class T>
Node::Node() //错误的定义
//采用类模板来实现方法,正确定义如下
//Node<T>::Node()
{ data = d;}
```

模板编程要求

- 类似于STL的vector,list等数据结构,一般情况下只能处理同构结点
  - vector<int> a; //表示所有结点全部为int
  - list<double> b;//表示所有结点全部为double
- 如果想处理异构结点,通常只能用基类指针来实现
- 如果采用对象指针作为类型参数,注意相应数据结构在销毁不会自动调用类析构函数,需要开发者自行调用delete进行销毁

```
class Base;
class D1:public Base;
class D2:public Base;
//变通处理异构结点
void main(){
   Base * p;
     vector<Base *> vec;
  p = new D1;
   vec.push_back(p);

  p = new D2;
   vec.push_back(p);

}
```

课堂练习

- 将CNode 作一个类模板类，实现不同类型的实例
  - 应该形如下列定义 CNode<int> intNode;
  - 可采用link_list.cpp 作为链表，无需重新实现
- 完善变长数组类，新增一个setLength() 方法.
  - 这个方法除了重新分配内存以外，还要把以前的数据拷贝到新的内存中.