Practice of Programming 8 template II

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

评分调整

- 增加一次模板的作业，其中包含两个题：
  - 函数模板和类模板
  - 10%

- 平时成绩：50% -> 40%
  - 第一阶段提交制品：15%
    - 程序设计质量：10%，考核依据为源代码和设计文档
    - 程序编写质量：5%，考核依据为源代码
  - 第二阶段提交制品：25%
    - 程序设计质量：15%，考核依据为源代码和设计文档
    - 程序编写质量：10% -> 5%，考核依据为源代码
  - 学生参与课程的程度：10% -> 5%

内容提纲

- 类模板成员
  - 类模板成员函数
  - 友元声明
  - 非类型形参的模板实参
  - 类模板中的友元声明
  - Queue和QueueItem的友元声明

定义类模板

- 可以像定义函数模板一样定义类模板

- Queue类
  - 能够支持不同类型的对象，将其定义成类模板
  - 包含的操作：
  - push，在队尾增加一项
  - pop，在队头删除一项
  - front，返回队头元素的引用
  - empty，之处队列中是否有元素

定义类模板

- Queue类

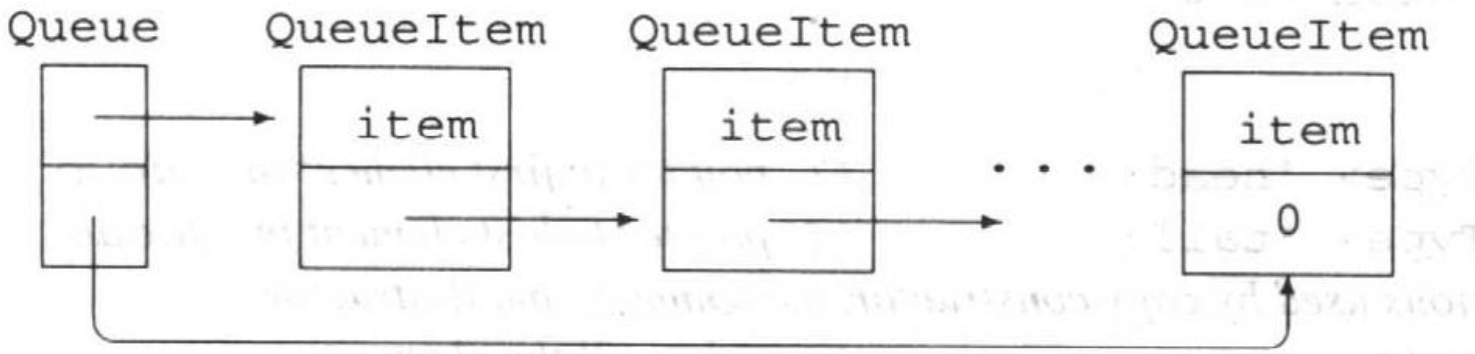
```
template <class Type> class Queue {
 public:
      Queue();
      Type &front();
      const Type &front () const;
      void push(const Type &);
      bool empty() const;
  private:
    // ....
};
```

定义类模板

- Queue类的实现
  - 使用链表来实现
  - 分成两个类
    - QueueItem类：表示Queue链表中的节点
      - 包含两个数据成员：item和next
      - item保存Queue中元素的值，它的类型随Queue的每个实例而变化
      - next是指向队列中下一个元素的指针
      - Queue中的每个元素保存在一个QueueItem对象中
    - Queue类：表示队列
      - 包含两个数据成员：head和tail
      - 都是指向QueueItem类型的指针

定义类模板

- Queue类的实现
  - 使用链表来实现



定义类模板

- QueueItem类

```
template <class Type> class QueueItem{
    // private class: no public section
    QueueItem(const Type &t):item(t),next(0){};
    Type item;       // value stored in this element
    QueueItem *next;  // pointer to next element in the Queue
};
```

- QueueItem是个类模板
  - 使用模板形参指定item成员的类型
  - 每实例化一个Queue时，也会实例化相应的QueueItem
  - QueueItem是私有类，没有公共接口

友元 friend

- 允许特定的非成员函数访问一个类的私有成员
  - 同时仍然阻止一般的访问
- 友元机制允许一个类将对其非公有成员的访问权授予指定的函数或类
  - 友元的声明以关键字friend开始，只能出现在类定义内部
- 友元声明可以出现在类中的任何地方
  - 友元不是授予友元关系的那个类的成员
  - 所以它们不受其声明出现部分的访问控制影响

友元 friend

- Screen类表示显示器屏幕
- Window_Mgr类表示窗口管理器，管理给定显示器上的一组Screen

```
class Screen {
   friend class Window_Mgr;
   .......
}
```

- Window_Mgr的成员可以直接访问Screen的私有成员

友元 friend

- 友元可以是普通的非成员函数，其他类的成员函数或整个类

- 将一个类设为友元，友元类的所有成员函数都可以访问授予友元关系的那个类的非公有成员

```
class Screen(){
    friend Window_Mgr&
        Window_Mgr::relocate(Window_Mgr::index,
                             Window_Mgr::index, Screen&);
}
```

  - 将成员函数声明为友元时，函数名必须用该函数所属的类名字加以限定

友元 friend

- 为了正确地构造类，需要注意友元声明与友元定义之间的互相依赖

- 类Window_Mgr必须先定义
  - 否则，Screen类就不能将Window_Mgr函数指定为友元
  - 只有在定义Screen之后，才能定义relocate函数
    - 友元访问了Screen的成员

- 更一般地讲，必须先定义包含成员函数的类，才能将成员函数设为友元
  - 另一方面，不必预先声明类和非成员函数来将它们设为友元

友元 friend

- 类必须将重载函数集中每一个希望设为友元的函数都声明为友元：

```
extern std::ostream& storeOn(std::ostream&, Screen&)；
extern BitMap& storeOn(BitMap&, Screen&);
class Screen {
    friend std::ostream& storeOn(std::ostream&, Screen&)
```

- 类Screen将接受一个ostream&的storeOn版本设为自己的友元，接受一个BitMap&的版本对Screen没有特殊访问权

定义类模板

- Queue类

```
template <class Type> class Queue{
 public:
    // empty Queue
    Queue():head(0),tail(0) {};
 //copy control to manage pointers to QueueItems in the Queue
    Queue(const Queue &Q):head(0),tail(0) {copy_elems(Q);};
    Queue& operator= (const Queue&);
    ~Queue(){destroy();};
```

定义类模板


```
// return element from head of Queue
// unchecked operation: front on an empty Queue is undef
Type &front() {return head->item;};
const Type &front () const {return head->item;};

void push(const Type &) ;    // add element to back of Queue
void pop();                  // remove element from head of Queue

bool empty()const {          // true if no elements in the Queue
    return head == 0;
};
```

定义类模板


```
private:
    QueueItem<Type> *head; // pointer to first element in Queue
    QueueItem<Type> *tail;     // pointer to last element in Queue

    // utility functions used by copy constructor,
    // assignment, and destructor
   void destroy();    // delete all the elements
   // copy elements from parameter
   void copy_elems(const Queue&);
};
```

类模板成员函数

- 类模板成员函数的定义：
  - 必须以关键字template开头，后接类的模板形参表
  - 必须指出它是哪个类的成员
  - 类名必须包含其模板形参

- 在类外定义的Queue类的成员函数的开头应该是：

template <class T> return-type Queue<T>:: member-name

类模板成员函数

- destroy函数：

```
template <class Type> void Queue<Type>:: destroy()
{
    while (!empty())
      pop();
 }
```

- 用名为Type的类型形参定义一个函数模板；
- 它返回void
- 它是在类模板Queue<Type>的作用域中

类模板成员函数

- pop函数：

```
template <class Type> void Queue<Type>:: pop(){
    // pop is unchecked: Popping off an empty
   // Queue is undefined
   QueueItem<Type>* p = head; //keep pointer to head
   head = head->next; //head now points to next element
   delete p; //delete old head element
}
```

- pop函数假设用户不会在空Queue上调用pop
- pop的工作是除去Queue的头元素

类模板成员函数

- push函数：

```
template <class Type> void Queue<Type>:: push(const Type& val) {
    // allocate a new QueueItem Object
    QueueItem<Type>* pt = new QueueItem<Type>(val);
    //put item onto existing queue
    if (empty())
        head = tail = pt; //the queue now has only one element
    else{
        tail->next = pt; //add new element to end of the queue
        tail = pt;
    }
```

- push函数分配新的QueueItem对象，用传递的值初始化它
- 如果Queue为空，则head和tail都应该指向这个新元素

类模板成员函数

- copy_elems函数：

```
template <class Type>
void Queue<Type>:: copy_elems(const Queue &orig){
// copy elements from orig into this Queue
// loop stops when pt == 0, which happens when we
//reach orig.tail
    for(QueueItem<Type> *pt = orig.head; pt; pt = pt->next)
      push(pt->item); //copy the element
}
```

  - copy_elems函数将orig的元素复制到这个Queue中

类模板成员函数

- 类模板的成员函数本身也是函数模板
  - 需要实例化
  - 实例化类模板成员函数时，编译器不执行模板实参推断
  - 模板形参由调用该函数的对象的类型确定

- 当调用 Queue<int> 类型对象的push成员时
  - 实例化的push函数为

    void Queue<int>::push(const int &val)

- 对象的模板实参能够确定成员函数模板形参
  - 调用类模板成员函数比调用类似函数模板更灵活

类模板成员函数

- 用模板形参定义的函数形参的实参允许进行常规转换：

```
Queue<int> qi; // instantiates class Queue<int>
short s = 42;
int i = 42;
// ok:  s converted to int and passed to push
qi.push(s); // instantiates Queue<int>::push(const int&)
qi.push(i);  // uses Queue<int>::push(const int&)
f(s);   // instantiates f(const short&)
f(i);    // instantiates f(const int&)
```

类模板成员函数

- 类模板的成员函数只有为程序所用才进行实例化
  - 如果某函数从未使用，则不会实例化该成员函数

- 定义模板类型的对象时，该定义导致实例化类模板
  - 定义对象也会实例化用于初始化该对象的构造函数
  - 以及该构造函数调用的任意成员

```
// instantiates Queue<string> class and Queue<string>::Queue()
Queue<string> qs;
qs.push("hello"); //instantiates Queue<string>::push
```

非类型形参的模板实参

- Screen类

```
template<int hi, int wid> class Screen{
  public:..
      Screen():screen(hi*wid, '#'), cursor (0),
                height(hi), width(wid) {}
      // .....
  private:
      std::string  screen;
      std::string::size_type cursor;
      std::string::size_type height, width;
};
```

类模板中的友元声明

- 在类模板中可以出现三种友元声明
  - 普通非模板类或函数的友元声明
    - 将友元关系授予明确指定的类或函数
  - 类模板或函数模板的友元声明
    - 授予对友元所有实例的访问权
  - 只授予对类模板或函数模板的特定实例的访问权的友元声明

类模板中的友元声明

- 普通友元
  - 非模板类或非模板函数可以是类模板的友元

```
template<class Type> class Bar{
  friend class FooBar;
  friend void fcn();
  //.....
}
```

  - FooBar的成员和fcn函数可以访问Bar类的任意实例的private成员和protected成员

类模板中的友元声明

- 一般模板友元关系
  - 友元可以是类模板或函数模板

```
template<class Type> class Bar{
  template <class Type> friend class Foo1;
  template <class Type> friend void templ_fcn1(const T&);
   //.....
}
```

  - Foo1的任意实例都可以访问Bar的任意实例的私有元素
  - templ_fcn1的任意实例可以访问Bar的任意实例

  - 对Bar的每个实例而言，Foo和templ_fcn1的所有实例都是友元

类模板中的友元声明

- 特定的模板友元关系
  - 类可以只授予对特定实例的访问权

```
template<class Type> class Foo2;
template<class Type> void templ_fcn2(const T&);
template<class Type> class Bar{
  friend class Foo2<char*>;
  friend void templ_fcn2<char*>(char* const &);
   //.....
}
```

  - 友元关系只扩展到Foo2的形参类型为char*的特定实例
  - 只有形参类型为char*的templ_fcn2实例是Bar类的友元

类模板中的友元声明

- 特定的模板友元关系
  - 类可以只授予对特定实例的访问权

```
template<class Type> class Foo3;
template<class Type> void templ_fcn3(const T&);
template<class Type> class Bar{
 friend class Foo3<Type>;
 friend void templ_fcn3<Type>(const T&);
  //.....
}
```

  - Bar的特定实例与使用同一模板实参的Foo3或templ_fcn3的实例之间是友元关系
  - Foo3<int>可以访问Bar<int>的私有部分，但是不能访问Bar的其他实例的私有部分

类模板中的友元声明

- 声明依赖性
  - 编译器将友元声明当做类或函数的声明对待
- 要想限制对特定实例化的友元关系，必须在可以用于友元声明之前声明类或函数

```
template <class T> class A;
template <class T> class B{
 public:
   friend class A<T>;  //ok: A is known to be a template
   friend class C;       //ok: C must be an ordinary, nontemplate class
   template <class S> friend class D; //ok: D is a template
   friend class E<T>;   //error: E wasn't declared as a template
   friend class F<int>;  //error: F wasn't declared as a template
}
```

Queue和QueueItem的友元声明

- QueueItem类应该将友元关系授予所有的Queue类实例，还是只授予特定实例？

- Queue类和QueueItem类之间应该是一对一映射

```
template <class Type> class Queue;
template <class Type> class QueueItem {
    friend class Queue<Type>;
}
```

Queue和QueueItem的友元声明

- 增加输出Queue对象的内容的能力
- 重载输出操作符，并将输出操作符设为模板

```
template <class Type>
ostream& operator<< (ostream &os,  const Queue<Type> &q){
        os << "<";
        QueueItem<Type> *p;
        for(p = q.head; p; p = p->next)
           os << p->item << " ";
        os << ">";
        return os;
}
```

Queue和QueueItem的友元声明

- 输出操作符需要成为Queue类和QueueItem类的友元

```
template <class T> std::ostream& operator<<
(std::ostream &, const Queue<Type>) ;

template <class Type> class QueueItem{
  friend class Queue<Type>;
  friend std::ostream& operator<<
        (std::ostream &, const Queue<Type>) ;
}

template <class Type> class Queue{
  friend std::ostream& operator<<
        (std::ostream &, const Queue<Type>) ;
}
```

Queue和QueueItem的友元声明

- Queue类的operator<<输出操作符依赖于item对象的operator<<实际输出每个元素：

  os << p->item << " ";

- 绑定到Queue且使用Queue输出操作符的每种类型本身也必须有输出操作符

- 没有语言机制指定或强制Queue自身定义中的依赖性

- 为没有定义输出操作符的类创建Queue对象是合法的，但是输出保存这种类型的Queue对象会发生编译时或链接时错误

Reference

- 《C++ Primer（第四版）》，Stanley B. Lippman、Barbara E. Moo、Josée LaJoie著，李师贤等人译，人民邮电出版社

Thank You!