**暨南大学本科实验报告专用纸**

课程名称 C++程序设计 成绩评定

实验项目名称 类模板编程实验 指导教师 王勇杰

实验项目编号 10 实验项目类型 编程型 实验地点 C105

学生姓名 李炜鹏 学号 2017052544

学院 智能科学与工程学院 系 计算机 专业 信息安全

实验时间2019年12月10日 午～12月13日 午 温度 ℃湿度

1. **实验目的**
2. 认识并理解类模板的概念
3. 熟悉C++中类模板的用法，包括定义类模板、实例化类模板、类模板成员函数的实例化、在类外使用类模板名、类模板和友元（特定、通用）、模板类型别名、类模板的static成员、重载类的成员函数
4. 实验要求：阅读课本第16章的内容， 试调试运行“实验十 类模板编程实验”文件夹中的源代码
5. 实验报告要求：
   1. 实验目的
   2. 实验原理
   3. 注释必要的代码
   4. 实验结果的截图
   5. 实验评估
6. **实验原理**
7. 类模板：是用来生成类的蓝图的。与函数模板不同的是，编译器不能为类模板腿短参数类型，我们必须使用尖括号里面的参数实例化类模板。
8. 类模板的使用：
   1. 定义类模板：template <typename T> class 类名
   2. 实例化类模板：类名<参数类型> 类对象
   3. 在模板作用域中引用模板类型：P585略
   4. 类模板成员函数：

定义在类体内：内联函数

定义在类体外：必须以关键字template开始，后接模板参数列表

template <typename T> ret-type 类名<T>::成员函数名(参数列表)

注：在默认情况下，一个类模板的成员函数只有当程序用到它时才进行实例化

* 1. 类模板和友元（此知识点较为重要，详见P588-590）：一对一/特定（需前置声明）、通用、自身类型
  2. 模板类型别名
  3. 类模板的static成员

1. **主要仪器设备**

**仪器：**计算机

**实验环境：**Visual Studio 2017或Dev-C++

1. **源程序**

**Blob.h**

#ifndef BLOB\_H

#define BLOB\_H

#include <iterator>

#include <string>

#include <vector>

#include <cstddef>

#include <stdexcept>

#include <utility>

#include <memory>

#include <algorithm>

#include <iostream>

#include <cstdlib>

#include <stdexcept>

//前置声明，在Blob中声明友元所需呀的

template <typename> class BlobPtr;

template <typename> class Blob; //运算符==中的桉树所需要的

template <typename T> bool operator==(const Blob<T>&, const Blob<T>&);

template <typename T> class Blob {

friend class BlobPtr<T>;

friend bool operator==<T> (const Blob<T>&, const Blob<T>&);

public:

typedef T value\_type;

typedef typename std::vector<T>::size\_type size\_type;

//构造函数

Blob();

template <typename It> Blob(It b, It e);

Blob(T\*, std::size\_t);

//返回第一个和最后一个个元素

BlobPtr<T> begin() { return BlobPtr<T>(\*this); }

BlobPtr<T> end()

{

BlobPtr<T> ret = BlobPtr<T>(\*this, data->size());

return ret;

}

//Blob中的元素数目

size\_type size() const { return data->size(); }

bool empty() const { return data->empty(); }

//增加和删除元素

void push\_back(const T &t) { data->push\_back(t); }

void pop\_back();

//元素访问

T& front();

T& back();

T& at(size\_type);

const T& back() const;

const T& front() const;

const T& at(size\_type) const;

T& operator[](size\_type i);

const T& operator[](size\_type i) const;

//Blob类型对象交换

void swap(Blob &b) { data.swap(b.data); }

private:

std::shared\_ptr<std::vector<T> > data;

void check(size\_type i, const std::string &msg) const;

};

//构造函数

template <typename T>

Blob<T>::Blob(T \*p, std::size\_t n) :data(new std::vector<T>(p, p + n)) { }

template <typename T>

Blob<T>::Blob() :data(new std::vector<T>()) { }

template <typename T>

template <typename It>

Blob<T>::Blob(It b, It e) :data(new std::vector<T>(b, e)) { }

//检查索引数据的合理性

template <typename T>

void Blob<T>::check(size\_type i, const std::string &msg) const

{

if (i >= data->size())

throw std::out\_of\_range(msg);

}

//元素访问成员函数

template <typename T>

T& Blob<T>::front()

{

check(0, "front on empty Blob");

return data->front();

}

template <typename T>

T& Blob<T>::back()

{

check(0, "back on empty Blob");

return data->back();

}

template <typename T> void Blob<T>::pop\_back()

{

check(0, "pop\_back on empty Blob");

data->pop\_back();

}

//重载其const型元素访问成员函数

template <typename T>

const T& Blob<T>::front() const

{

check(0, "front on empty Blob");

return data->front();

}

template <typename T>

const T& Blob<T>::back() const

{

check(0, "back on empty Blob");

return data->back();

}

template <typename T>

T& Blob<T>::at(size\_type i)

{

// 如果i太大，将会抛出检查，从而阻止访问不存在的元素

check(i, "subscript out of range");

return (\*data)[i]; //(\*data)是此对象指向的容器

}

//重载const型的at函数

template <typename T>

const T& Blob<T>::at(size\_type i) const

{

check(i, "subscript out of range");

return (\*data)[i];

}

template <typename T>

T& Blob<T>::operator[](size\_type i)

{

// 如果i太大，将会抛出检查，从而阻止访问不存在的元素

check(i, "subscript out of range");

return (\*data)[i];

}

template <typename T>

const T& Blob<T>::operator[](size\_type i) const

{

check(i, "subscript out of range");

return (\*data)[i];

}

//运算符重载

template <typename T>

std::ostream& operator<<(std::ostream &os, const Blob<T> a)

{

os << "< ";

for (size\_t i = 0; i < a.size(); ++i)

os << a[i] << " ";

os << " >";

return os;

}

template <typename T>

bool operator==(const Blob<T> lhs, const Blob<T> rhs)

{

if (rhs.size() != lhs.size())

return false;

for (size\_t i = 0; i < lhs.size(); ++i) {

if (lhs[i] != rhs[i])

return false;

}

return true;

}

// 如果BlobPtr访问不存在的元素时将会引发异常

template <typename T>

bool operator==(const BlobPtr<T>&, const BlobPtr<T>&);

template <typename T>

class BlobPtr : public std::iterator<std::bidirectional\_iterator\_tag, T> {

friend bool operator==<T>(const BlobPtr<T>&, const BlobPtr<T>&);

public:

BlobPtr() : curr(0) { }

BlobPtr(Blob<T> &a, size\_t sz = 0) :wptr(a.data), curr(sz) { }

T &operator[](std::size\_t i)

{

std::shared\_ptr<std::vector<T> > p =

check(i, "subscript out of range");

return (\*p)[i]; //(\*p)是此对象指向的vector

}

const T &operator[](std::size\_t i) const

{

std::shared\_ptr<std::vector<T> > p =

check(i, "subscript out of range");

return (\*p)[i]; //(\*p)是此对象指向的vector

}

T& operator\*() const

{

std::shared\_ptr<std::vector<T> > p = check(curr, "dereference past end");

return (\*p)[curr]; //(\*p)是此对象指向的vector

}

T\* operator->() const

{ // delegate the real work to the dereference operator？不懂。

return &this->operator\*();

}

// ++运算符与--运算符

BlobPtr& operator++(); // 后置++

BlobPtr& operator--();

BlobPtr operator++(int); // 前置++

BlobPtr operator--(int);

private:

// 如果检查成功，将检查返回一个shared\_ptr到vector

std::shared\_ptr<std::vector<T> > check(std::size\_t, const std::string&) const;

//定义一个weak\_ptr，意味着底层vector可能被销毁

std::weak\_ptr<std::vector<T> > wptr;

std::size\_t curr;//数组中的当前位置

};

//==运算符

template <typename T>

bool operator==(const BlobPtr<T> &lhs, const BlobPtr<T> &rhs)

{

return lhs.wptr.lock().get() == rhs.wptr.lock().get() &&

lhs.curr == rhs.curr;

}

template <typename T>

bool operator!=(const BlobPtr<T> &lhs, const BlobPtr<T> &rhs)

{

return !(lhs == rhs);

}

// check member

template <typename T>

std::shared\_ptr<std::vector<T> > BlobPtr<T>::check(std::size\_t i, const std::string &msg) const

{

std::shared\_ptr<std::vector<T> > ret =

wptr.lock(); //是否还存在vector？

if (!ret)

throw std::runtime\_error("unbound BlobPtr");

if (i >= ret->size())

throw std::out\_of\_range(msg);

return ret; //返回指向vector的智能指针shared\_ptr

}

//重载运算符成员函数

//后缀：增加/减少对象，但返回不变的值

template <typename T>

BlobPtr<T> BlobPtr<T>::operator++(int)

{

//这里不需要check；调用前缀++将进行检查

BlobPtr ret = \*this; //储存当前值

++\*this; // advance one element; prefix ++ checks the increment

return ret; // return the saved state

}

template <typename T>

BlobPtr<T> BlobPtr<T>::operator--(int)

{

// no check needed here; the call to prefix decrement will do the check

BlobPtr ret = \*this; // save the current value

--\*this; // move backward one element; prefix -- checks the decrement

return ret; // return the saved state

}

// prefix: return a reference to the incremented/decremented object

template <typename T>

BlobPtr<T>& BlobPtr<T>::operator++()

{

// if curr already points past the end of the container, can't increment it

check(curr, "increment past end of BlobPtr");

++curr; // advance the current state

return \*this;

}

template <typename T>

BlobPtr<T>& BlobPtr<T>::operator--()

{

// if curr is zero, decrementing it will yield an invalid subscript

--curr; // move the current state back one element

check(-1, "decrement past begin of BlobPtr");

return \*this;

}

#endif

**useBlob.cpp**

#include <string>

using std::string;

#include <iostream>

using std::cout; using std::endl;

#include "Blob.h"

int main()

{

Blob<string> b1; //空的Blob类

cout << "b1.size(): " << b1.size() << endl;

{ //新的作用域

string temp[] = { "a", "ab", "abc", "abcd" };

Blob<string> b2(temp, temp + sizeof(temp) / sizeof(\*temp));

b1 = b2; //b1和b2共享相同的元素

b2.push\_back("abcde");

cout << "b1.size(): " << b1.size() << endl;

cout << "b2.size(): " << b2.size() << endl;

} //b2被销毁，但是它指向的元素仍未被销毁

cout << "b1.size(): " << b1.size() << endl;

cout << "element in b1: " << endl;

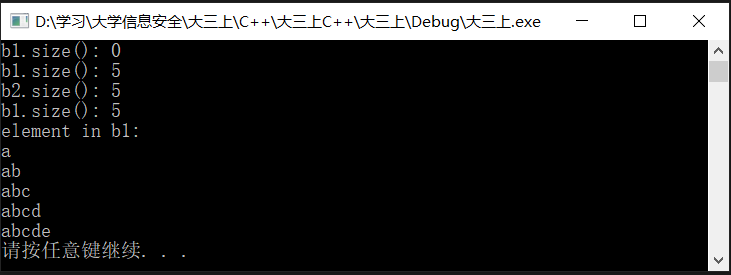
for (BlobPtr<string> p = b1.begin(); p != b1.end(); ++p)

cout << \*p << endl;

system("pause");

return 0;

}

****

参照前面实验，这里同样定义并且初始化了Blob类对象b1,b2，输出结果符合预期，本程序只是将代码中的类定义为模板类，具体实现仅仅在对应的数据成员和成员函数使用相应的模板定义而已。

1. **实验评估**

本实验主要运用了类模板的相关知识，通过本实验，我认识到了类模板的概念以及基本操作，包括定义类模板、实例化类模板、定义类模板的成员函数、使用友元、定义类模板别名等。类模板涉及C++里面的面向对象特性以及泛型编程，知识面十分广泛，操作方法也十分多样，涉及到的用法也十分的复杂，希望通过以后的深入学习继续提升。