Assignment 7

一. 概念题

1.1 C++中虚函数的作用是什么？为什么C++中析构函数往往是虚函数?

解：虚函数有两个作用：指定消息采用动态绑定、指出基类中可以被派生类重定义的成员函数；为了防止内存泄露，使得在析构派生类对象时，总是调用派生类的析构函数。

1.2 简述C++中静态绑定和动态绑定的概念，并说明动态绑定发生的情况.

解：静态绑定：绑定静态类型，所对应的函数或属性依赖于对象的静态类型，发生在编译期；动态绑定：绑定动态类型，所对应的函数或属性依赖于对象的动态类型，发生在运行期；动态绑定发生情况：通过基类的引用或指针调用虚函数。

二. 编程题

2.1 请阅读下面的代码，写出程序的运行结果.

```
#include <iostream>
using namespace std;
class A {
  public:
    A() { cout << "default construct A" << endl; }
    A(const A& a) { cout << "copy construct A" << endl; }
    virtual ~A() { cout << "destruct A" << endl; }
    void f () { cout << "A::f" << endl; }
    virtual void g() { cout << "A::g" << endl; }
};
class B : public A {
  public:
    B() { cout << "default construct B" << endl; }
    B(const B& b) { cout << "copy construct B" << endl; }
    ~B() { cout << "destruct B" << endl; }
    void f() { cout << "B::f" << endl; }
    void g() { cout << "B::g" << endl; }
};
void func1(A a) {
    a.f();
    a.g();
}
void func2(A &a) {
    a.f();
    a.g();
}
int main() {
    A *a = new A();
    A *b = new B();
```

```
    func1(*a); func2(*a);
    func1(*b); func2(*b);
    *a = *b;
    func1(*a); func2(*a);
    delete a; delete b;
    return 0;
}

default construct A
default construct A
default construct B
copy construct A
A::f
A::g
destruct A
A::f
A::g
copy construct A
A::f
A::g
destruct A
A::f
B::g
copy construct A
A::f
A::g
destruct A
A::f
A::g
destruct A
destruct B
destruct A
```

2.2 要求基于抽象类Queue实现三种形式的队列，其中Queue1按照先进先出的原则，Queue2选择最小的元素出列，Queue3选择最大的元素出列.

```
#include <functional>
class Queue
{
  public:
    virtual bool enqueue(int num) = 0;
    virtual bool dequeue(int &num) = 0;
};
class Queue1 : public Queue
{
  private:
    struct Node
    {
        int value;
        Node *next;
        explicit Node(int v, Node *n = nullptr) : value(v), next(n) {}
```

```
    };
    Node *sentinel;
    Node *last;

  public:
    Queue1()
    {
        sentinel = new Node(0);
        last = sentinel;
    }
    bool enqueue(int num) override
    {
        last->next = new Node(num);
        last = last->next;
        return true;
    }
    bool dequeue(int &num) override
    {
        if (!sentinel->next)
        {
            return false;
        }
        num = sentinel->next->value;
        Node *lastSentinel = sentinel;
        sentinel = sentinel->next;
        delete lastSentinel;
        return true;
    }
};
template <class Compare>
class PQ : public Queue
{
  private:
    static const int INIT_CAP = 16;
    Compare compare;
    int *items;
    int size;
    int cap;
    static int parent(int n)
    {
        return n / 2;
    }
    static int left(int n)
    {
        return n * 2;
    }
    static int right(int n)
    {
        return n * 2 + 1;
    }
    void swap(int a, int b)
    {
        int temp = items[a];
        items[a] = items[b];
        items[b] = temp;
    }
    void swim(int n)
    {
```

```
        if (n == 1)
        {
            return;
        }
        if (compare(items[n], items[parent(n)]))
        {
            swap(parent(n), n);
            swim(parent(n));
        }
    }
    void sink(int n)
    {
        if (left(n) > size)
        {
            return;
        }
        else if (right(n) > size)
        {
            if (compare(items[left(n)], items[n]))
            {
                swap(n, left(n));
            }
        }
        else
        {
            int bestIndex = compare(items[left(n)], items[right(n)]) ? left(n) :
right(n);
            if (compare(items[bestIndex], items[n]))
            {
                swap(n, bestIndex);
                sink(bestIndex);
            }
        }
    }
    void resize(int newCap)
    {
        if (size > newCap)
        {
            return;
        }
        int *oldItem = items;
        items = new int[newCap + 1];
        cap = newCap;
        for (int i = 1; i <= size; ++i)
        {
            items[i] = oldItem[i];
        }
        delete[] oldItem;
    }

  public:
    PQ()
    {
        items = new int[INIT_CAP + 1];
        size = 0;
        cap = INIT_CAP;
    }
    bool enqueue(int num) override
```

```
    {
        items[++size] = num;
        swim(size);
        if (size == cap)
        {
            resize(cap * 2);
        }
        return true;
    }
    bool dequeue(int &num) override
    {
        if (size == 0)
        {
            return false;
        }
        num = items[1];
        items[1] = items[size--];
        sink(1);
        if (cap > INIT_CAP && size <= cap / 2)
        {
            resize(cap / 2);
        }
        return true;
    }
};
typedef PQ<std::less<int>> Queue2;
typedef PQ<std::greater<int>> Queue3;
```