Solution7

191220008 陈南曈

概念题

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?

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。

析构函数执行时先调用派生类的析构函数，其次才调用基类的析构函数。如果析构函数不是虚函数，而程序执行时又要通过基类的指针去销毁派生类的动态对象，那么用delete销毁对象时，只调用了基类的析构函数，未调用派生类的析构函数。这样会造成销毁对象不完全，造成内存泄漏。

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。

- 对象的静态类型：就是它在程序中被声明时所采用的类型（或理解为类型指针或引用的字面类型），在编译期确定；
- 对象的动态类型：是指“目前所指对象的类型”（或理解为类型指针或引用的实际类型），在运行期确定；
- 静态绑定：又名前期绑定，绑定的是静态类型，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：又名后期绑定（late binding），绑定的是动态类型，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；

一般的，virtual函数是动态绑定，non-virtual函数是静态绑定，缺省参数值也是静态绑定。

在继承体系中所有虚函数使用的是动态绑定，其他的全部是静态绑定。

编程题

1、请阅读下面的代码，写出程序的运行结果。

分析如下：

```
int main() {
    A* a = new A(); // 创建A类对象。调用A类默认构造函数
    A* b = new B(); // 创建B类对象，调用B类默认构造函数
    func1(*a); // 用A类对象对A类形参初始化，调用A类拷贝构造函数
    func2(*a); // 传入A类对象的引用，无需调用拷贝构造函数
    func1(*b); // 用B类对象对A类形参初始化，调用A类拷贝构造函数
```

```
    func2(*b); // 传入B类对象的引用，无需调用拷贝构造函数，虚函数g()动态绑定，函数f()静态绑定
    *a = *b; // 对象的默认赋值操作，函数f和g不改变
    func1(*a); // 用A类对象对A类形参初始化，调用A类拷贝构造函数
    func2(*a); // 传入A类对象的引用，无需调用拷贝构造函数
    delete a; // 调用A类析构函数
    delete b; // 先调用派生类B析构函数，再调用基类A的析构函数
    return 0;
}
```

结果如下：

```
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、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 <iostream>
using namespace std;

struct Node
{
    int value;
    Node* next;
};

class Queue
{
protected:
    Node* head;
    Node* tail;
```

```
public:
    Queue() { head = NULL; tail = NULL; }
    ~Queue() { while (head) { Node* p = head; head = head->next; delete p; } }
    virtual bool enqueue(int num) = 0; //入列
    virtual bool dequeue(int& num) = 0; //出列
};

class Queue1 : public Queue // 先进先出
{
public:
    bool enqueue(int num); //入列
    bool dequeue(int& num); //出列
};

bool Queue1::enqueue(int num) //入列
{
    Node* p = new Node;
    p->value = num;
    p->next = NULL;
    if (head == NULL)
    {
        head = p;
        tail = p;
    }
    else
    {
        tail->next = p;
        tail = p;
    }
    return true;
}

bool Queue1::dequeue(int& num) //出列
{
    if (head == NULL)
        return false;
    Node* p = head;
    num = p->value;
    head = head->next;
    delete p;
    return true;
}

class Queue2 : public Queue // 最小元素先出
{
public:
    bool enqueue(int num); //入列
    bool dequeue(int& num); //出列
};

bool Queue2::enqueue(int num) //入列
{
    Node* p = new Node;
    p->value = num;
    p->next = NULL;
    if (head == NULL)
    {
        head = p;
```

```
        tail = p;
    }
    else
    {
        Node* cur = head;
        Node* pre = NULL;
        while (cur != NULL && cur->value < num)
        {
            pre = cur;
            cur = cur->next;
        }
        p->next = cur;
        if (pre != NULL)
            pre->next = p;
        else
            head = p;
        if (p->next == NULL)
            tail = p;
    }
    return true;
}

bool Queue2::dequeue(int& num) //出列
{
    if (head == NULL)
        return false;
    Node* p = head;
    num = p->value;
    head = head->next;
    delete p;
    return true;
}

class Queue3 : public Queue // 最大元素先出
{
public:
    bool enqueue(int num); //入列
    bool dequeue(int& num); //出列
};

bool Queue3::enqueue(int num) //入列
{
    Node* p = new Node;
    p->value = num;
    p->next = NULL;
    if (head == NULL)
    {
        head = p;
        tail = p;
    }
    else
    {
        Node* cur = head;
        Node* pre = NULL;
        while (cur != NULL && cur->value > num)
        {
            pre = cur;
            cur = cur->next;
```

```
        }
        p->next = cur;
        if (pre != NULL)
            pre->next = p;
        else
            head = p;
    }
    return true;
}

bool Queue3::dequeue(int& num) //出列
{
    if (head == NULL)
        return false;
    Node* p = head;
    num = p->value;
    head = head->next;
    delete p;
    return true;
}
```