Solution1

191220008 陈南曈

概念题

1、从数据和过程的角度，简述抽象与封装的区别。

抽象：抽象是指该程序实体的外部可观察到的行为，不考虑该程序实体的内部是如何实现的。处理大而复杂的问题的重要手段是抽象，强调事物本质的东西（控制复杂度）。抽象包括过程抽象和数据抽象。

封装：封装是指把该程序实体内部的具体实现细节对使用者隐藏起来，只对外提供一个接口（信息保护）。封装包括过程封装和数据封装。

过程抽象：用一个名字来代表一段完成一定功能的程序代码，代码的使用者只需要知道代码的名字以及相应的功能，而不需要知道对应的程序代码是如何实现的。

过程封装：把命名代码的具体实现隐藏起来（对使用者不可见，或不可直接访问），使用者只能通过代码名字来使用相应的代码。命名代码所需要的数据是通过参数（或全局变量）来获得，计算结果通过返回机制（或全局变量）返回。

数据抽象：只描述对数据能实施哪些操作以及这些操作之间的关系，数据的使用者不需要知道数据的具体表示形式。

数据封装：把数据及其操作作为一个整体来进行实现，其中，数据的具体表示被隐藏起来（使用者不可见，或不可直接访问），对数据的访问（使用）只能通过提供的操作（对外接口）来完成。

区别：

封装考虑内部实现，抽象考虑的是外部行为

封装是屏蔽细节，抽象是提取共性

抽象是一种思维方式，而封装则是一种基于抽象性的操作方法。通过抽象所得到数据信息及其功能，以封装的技术将其重新聚合，形成一个新的聚合体，也就是类。

2、简述面向过程与面向对象程序设计的区别；列举两个更适合面向对象的场景，并说明理由。

	面向过程程序设计	面向对象程序设计
抽象性	以功能为中心，强调过程（功能）抽象，但数据与操作分离，二者联系松散。	以数据为中心，强调数据抽象，操作依附于数据，二者联系紧密。
封装性	实现了操作的封装，但数据是公开的，数据缺乏保护。	实现了数据的封装，加强了数据的保护。
模块划分	按子程序划分模块，模块边界模糊。	按对象类划分模块，模块边界清晰。
复用性	子程序往往针对某个程序而设计，这使得程序难以复用。	对象类往往具有通用性，再加上继承机制，使得程序容易复用。
可维护性	功能易变，程序维护困难。	对象类相对稳定，有利于程序维护。
问题求解	基于子程序的解题方式与问题空间缺乏对应。	基于对象交互的解题方式与问题空间有很好的对应。

场景一：当需要进行大量类似的操作，每次操作大致功能相同，但细节上有区别时。

理由：可以利用面向对象的继承机制，更易于达到这样的功能。

场景二：当需要不断对一些底层实现进行更改，但对外接口不变时。

理由：面向对象有易于维护的特点，当更改实现方式时，不会对使用者产生影响。

编程题

1、仿照课堂所讲栈类Stack的实现，利用链表和数组分别实现队列类Queue。

链表实现

```
#include <iostream>

using namespace std;

const int QUEUE_SIZE = 100;

class Queue
{
public: //对外的接口（外部可使用的内容）
```

```
    Queue(); // 构造函数
    void enQueue(int i); // 入队列
    void deQueue(int& i); // 出队列
    void printAll(); // 打印队列内所有元素
private: //隐藏的内容，外部不可使用
    int count;
    struct Node
    {
        int content;
        Node* next;
    }*head, *tail;
};

Queue::Queue()
{
    count = 0;
    head = NULL;
    tail = head;
}

void Queue::enQueue(int i)
{
    Node* p = new Node;
    if (count == QUEUE_SIZE - 1)
    {
        cout << "Queue is overflow.\n";
        exit(-1);
    }
    else
    {

        p->content = i;
        p->next = NULL;
        if(tail==NULL)
        {
            head = tail = p;
        }
        else
        {
            tail->next = p;
            tail = p;
        }
        count++;
        return;
    }
}
void Queue::deQueue(int& i)
{
    if (count == 0)
    {
        cout << "Queue is empty.\n";
        exit(-1);
    }
    else
    {
        Node* p = head;
        head = head->next;
        i = p->content;
```

```
        delete p;
        count--;
        return;
    }
}

void Queue::printAll()
{
    Node* p = head;
    while (p != NULL)
    {
        cout << p->content << endl;
        p = p->next;
    }
}
int main()
{
    Queue queue;
    int i = 0;
    printf("i=%d\n", i);
    queue.enQueue(1);
    queue.enQueue(2);
    queue.deQueue(i);
    queue.enQueue(3);
    printf("i=%d\n", i);
    queue.printAll();
    return 0;
}
```

数组实现

```
#include <iostream>

using namespace std;

const int QUEUE_SIZE = 100;

class Queue
{
public: // 对外的接口（外部可使用的内容）
    Queue(); // 构造函数
    void enQueue(int i); // 入队列
    void deQueue(int& i); // 出队列
    void printAll(); // 打印队列内所有元素
private: // 隐藏的内容，外部不可使用
    int size;
    int head;
    int tail;
    int *buffer;
    int count;
};

Queue::Queue()
{
    size = 10;//队列容量
```

```
    head = 0;
    tail = -1;
    buffer = new int[10];
    count = 0;//元素个数
}

void Queue::enQueue(int i)
{
    if (tail != -1 && (tail + 1) % QUEUE_SIZE == head && count != 0)
    {
        cout << "Queue is overflow.\n";
        exit(-1);
    }
    else
    {
        if (tail != -1 && (tail + 1) % size == head && count != 0)
        {
            int* newbuffer = new int[size + 10];//每次扩容10个元素
            for (int i = 0; i < size; i++)
                newbuffer[i] = buffer[(head + i) % size];
            int* temp = buffer;
            buffer = newbuffer;
            delete[]temp;//释放原队列空间
            head = 0;
            tail = size - 1;
            size = size + 10;
        }
        tail = (tail + 1) % size;
        buffer[tail] = i;
        count++;
        return;
    }
}
void Queue::deQueue(int& i)
{
    if ((tail + 1) % QUEUE_SIZE == head && count == 0)
    {
        cout << "Queue is empty.\n";
        exit(-1);
    }
    else
    {
        i = buffer[head];
        head = (head + 1) % size;
        count--;
        return;
    }
}

void Queue::printAll()
{
    for (int i = 0 ; i < count; i++)
        cout << buffer[(head + i) % size] << endl;
}
int main()
{
    Queue queue;
    int i = 0;
```

```
    printf("i=%d\n", i);
    queue.enQueue(1);
    queue.enQueue(2);
    queue.deQueue(i);
    queue.enQueue(3);
    printf("i=%d\n", i);
    queue.printAll();
    return 0;
}
```