Assignment 1

一、题目描述

概念题：

1、从数据和过程的角度，简述抽象与封装的区别。

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现形式。而数据封装把数据极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（接口）来完成。

过程抽象用一个名字来代表一段完成一定功能的代码，代码的使用者只需要知道代码的的名字以及相应的功能，而不需要知道对应程序的代码是如何实现的。过程封装把命名代码的具体实现隐藏起来，使用者只能通过代码名字来访问相应的代码。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。

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。

面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了；面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

比如GUI编程，传统的面向过程的语言在编写和设计GUI应用时十分吃力，而面向对象可以将界面元素抽象为控件，比如按钮、输入框、滚动条等，再使用这些模型创造实例就非常简单。如果使用面向过程的C语言，那么指针、结构体、构造、析构都需要自己实现，容易出错，开发成本大大增高。

大量需要迭代的程序追求扩展的程序，如APP、游戏开发、内核，面向对象编程可以大大提高代码的易维护性、可扩展性、高复用性，多快好省，好拆分，稳定，模块化等等，比如Linux内核的驱动、驱动设备、总线、调度系统等等。

言之有理即可。

编程题：

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 <stdio.h>
#include <cstdlib>

class Queue {
 private:
  struct Node {
    int _data;
```

```
    Node *_next;
  };
  Node *_front, *_rear;
 public:
  void enQueue(int i) {
    Node *n = (Node *) malloc(sizeof(Node));
    n->_data = i;
    n->_next = nullptr;
    if (_rear) {
      _rear->_next = n;
      _rear = n;
    } else {
      _front = n;
      _rear = n;
    }
  }
  void deQueue(int &i) {
    if (_front == nullptr) {
      fprintf(stderr, "Queue empty\n");
      exit(-1);
    }
    i = _front->_data;
    Node * n = _front;
    _front = _front->_next;
    if (_front == nullptr)
      _rear = nullptr;
    free(n);
  }
  Queue() {
    _front = nullptr;
    _rear = nullptr;
  }
};
```

数组实现：

```
#include <stdio.h>
#include <cstdlib>
class Queue {
 private:
  int *_buffer;
  int _rear_index;
  int _front_index;
  int _capacity;
  const int max_capacity = 101;
  const int min_capacity = 11;
 public:
  Queue() {
    _buffer = (int *) malloc(min_capacity * sizeof(int));
```

```
14      _capacity = min_capacity;
15      _rear_index = 0;
16      _front_index = 0;
17    }
18
19    void enQueue(int i) {
20      if ((_rear_index + 1) % _capacity == _front_index) {
21        // may need to resize
22        if (_capacity < max_capacity) {
23          int new_capacity = _capacity + 20 < max_capacity ? _capacity + 20 :
max_capacity;
24          int *buffer = (int *) malloc(new_capacity * sizeof(int));
25          int index = 0;
26          while (_front_index != _rear_index) {
27            buffer[index++] = _buffer[_front_index];
28            _front_index = (_front_index + 1) % _capacity;
29          }
30          _front_index = 0;
31          _rear_index = _capacity - 1;
32          free(_buffer);
33
34          _buffer = buffer;
35          _capacity = new_capacity;
36        } else {
37          fprintf(stderr, "Queue overflow\n");
38          exit(-1);
39        }
40      }
41      _buffer[_rear_index] = i;
42      _rear_index = (_rear_index + 1) % _capacity;
43    }
44
45    void deQueue(int &i) {
46      if (_rear_index == _front_index) {
47        fprintf(stderr, "Queue empty\n");
48        exit(-1);
49      }
50      i = _buffer[_front_index];
51      _front_index = (_front_index + 1) % _capacity;
52    }
53    void printAll() {
54      int i = _front_index;
55      printf("Queue: ");
56      while (i != _rear_index) {
57        printf("%d ", _buffer[i]);
58        i = (i + 1) % _capacity;
59      }
60      printf("\n");
61    }
```

```
62 };
```