Week 5: Queue ADT

CSCI 2100 Data Structures
Fall 2017

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
Queue

- A queue is a data structure that stores data in such a way that the last piece of data stored, is the last one retrieved
  - also called First-In, First-Out (FIFO)
- Only access to the stack is the first and last element
  - consider people standing in line
    - they get service in the order that they arrive
Queue

- **Enque**
  - operation to place a new item at the tail of the queue

- **Dequeue**
  - operation to remove the next item from the head of the queue
Queue

enqueue(F)

item = dequeue()
item = X
Queue Specification

- **Definitions**: (provided by the user)
  - *MAX_ITEMS*: Max number of items that might be on the queue
  - *ItemType*: Data type of the items on the queue

- **Operations**
  - Boolean *IsEmpty*
  - Boolean *IsFull*
  - Enqueue (*ItemType newItem*)
  - Dequeue (*ItemType& item*)
Implementing a Queue

● At least two common methods for implementing a queue
  ▪ array
  ▪ linked list

● Which method to use depends on the application
  ▪ advantages? disadvantages?
Regular Linear Array

- In a standard linear array
  - Index 0 is the first element
  - array.length is the last element
- All objects removed would come from element 0
- All objects added would go one past the last currently occupied slot
- To implement this, when an object is removed, all elements must be shifted down by one spot
Regular Linear Array

```
| A | B | F | S | D | K | M |
```

```
start
```

```
end
```

```
dequeue()
```

```
B | F | S | D | K | M |
```

```
start
```

```
dequeue()
```

```
B | F | S | D | K | M |
```

```
start
```

```
shift
```

```
B | F | S | D | K | M |
```
Circular Array

- Need to keep track of the index that holds the first item
- Need to keep track of the index that holds the last item
- The “wrap-around” is accomplished through the use of the mod operator (%)
  
  \[
  \text{index} = (\text{end} + 1) \mod \text{array.length}
  \]
  
  - assume \text{array.length} = 5 and \text{end} = 4
  - then: \text{index} = (4 + 1) \mod 5 = 0
Circular Array

start

enqueue(F)

end = (end + 1) % 8

start = (start + 1) % 8

decqueue()

start

end

CSCI 2100
```cpp
#include <iostream>
using namespace std;
#define MAX_SIZE 10 //maximum size of the array that will store Queue.

class Queue
{
private:
    int A[MAX_SIZE];
    int front, rear;

public:
    // Constructor - set front and rear as -1.
    Queue() {
        front = -1;
        rear = -1;
    }

    // To check whether Queue is empty or not
    bool IsEmpty() {
        return (front == -1 && rear == -1);
    }

    // To check whether Queue is full or not
    bool IsFull() {
        return (rear + 1) % MAX_SIZE == front ? true : false;
    }

    // Inserts an element in queue at rear end
    void Enqueue(int x) {
        cout << "Enqueuing " << x << endl;
        if (IsFull()) throw runtime_error("Error: Queue is Full");
        if (IsEmpty()) {
            front = rear = 0;
        } else {
            rear = (rear + 1) % MAX_SIZE;
        }
        A[rear] = x;
    }

    // Removes an element in Queue from front end.
    void Dequeue() {
        cout << "Dequeueing " << endl;
        if (IsEmpty()) throw runtime_error("Error: Queue is Empty");
        if (front == rear) {
            rear = front = -1;
        } else {
            front = (front + 1) % MAX_SIZE;
        }
    }

    // Printing the elements in queue from front to rear.
    void Print()
    {
        // Finding number of elements in queue
        int count = (rear + MAX_SIZE - front) % MAX_SIZE + 1;
        cout << "Queue : ";
        for(int i = 0; i < count; i++) {
            int index = (front + i) % MAX_SIZE;
            cout << A[index] << " ";
        }
        cout << endl;
    }

    int main()
    {
        Queue Q; // creating an instance of Queue.
        Q.Enqueue(2); Q.Print();
        Q.Enqueue(4); Q.Print();
        Q.Enqueue(6); Q.Print();
        Q.Enqueue(8); Q.Print();
    }
};
```

Managing Templated Classes

- The conventions are different for templated classes in a multi-file project. The main issue is that templated code is not pre-compiled into independent object-code, because the underlying machine code depends intricately on the actual data type supplied as the template parameter.

- Therefore, templated code is compiled as needed when instantiated from other contexts.

- As a result, we use a slightly different convention for embedding the source code within separate files.

- The .tcc file is explicitly included from within the header file.
Pointers, Dynamic Memory, and the “new” Operator

- The operator new dynamically allocates the correct amount of storage for an object of a given type from the free store and returns a pointer to this object.

- This memory is also sometimes called heap memory.

```cpp
char* buffer = new char[500];  // allocate a buffer of 500 chars
buffer[3] = 'a';              // elements are still accessed using []
delete [] buffer;            // delete the buffer
```

- Memory leak: Having inaccessible objects in dynamic memory is called a memory leak. If an object is allocated with new, it should eventually be deallocated with delete.
class Vect {  
public:
    Vect(int n);  // constructor, given size
    ~Vect();     // destructor
    // ... other public members omitted
private:
    int* data;    // an array holding the vector
    int size;     // number of array entries
};

Vect::Vect(int n) {  // constructor
    size = n;
    data = new int[n];  // allocate array
}

Vect::~Vect() {  // destructor
    delete [] data;  // free the allocated array
}
Copy constructor and assignment operator

Notice that this does not copy the contents of the array; rather it copies the pointer to the array’s initial element. This default action is sometimes called a shallow copy.

Remember

Every class that allocates its own objects using `new` should:
- Define a **destructor** to free any allocated objects.
- Define a **copy constructor**, which allocates its own new member storage and copies the contents of member variables.
- Define an **assignment operator**, which deallocates old storage, allocates new storage, and copies all member variables.
Copy constructor and assignment operator

```cpp
Vect::Vect(const Vect& a) {
    size = a.size;
    data = new int[size];
    for (int i = 0; i < size; i++) {
        data[i] = a.data[i];
    }
}

Vect& Vect::operator=(const Vect& a) {
    if (this != &a) {
        delete [] data;
        size = a.size;
        data = new int[size];
        for (int i=0; i < size; i++) {
            data[i] = a.data[i];
        }
    }
    return *this;
}
```
The assignment operator

- Internally, we use the keyword \textit{this} to identify the class instance being operated upon.

- The \textit{this} pointer stores the address of the class instance, to enable pointer access of the members to the member functions of the class.

- From a technical perspective, \textit{this} is a pointer type in C++.

- Because we want to return the class object rather than a pointer to a class object, we dereference \texttt{*this} in the return statement.
Copy constructor vs assignment operator in C++

```cpp
#include <iostream>
using namespace std;

class Test {
public:
    Test() {}
    Test(const Test &t) {
        cout << "Copy constructor called " << endl;
    }
    Test& operator = (const Test &t) {
        cout << "Assignment operator called " << endl;
    }
};

int main() {
    Test t1;
    Test t2 = t1;
    Test t3(t2);
    Test t4, t5;
    t5 = t4;
    return 0;
}
```
Copy Constructor and Destructor

```cpp
39 // Copy Constructor
40 template <typename Object>
41 ArrayStack<Object>::ArrayStack(const ArrayStack& other)
42    : S(new Object[other.capacity]), capacity(other.capacity), t(other.t)
43 {
44    for (int j=0; j <= t; j++)
45        S[j] = other.S[j];
46 }
47
48 // Destructor
49 template <typename Object>
50 ArrayStack<Object>::~ArrayStack() {
51    delete[] S;
52 }
53
```
The assignment operator

```cpp
54 // Assignment Operator
55 template <typename Object>
56 ArrayStack<Object>& ArrayStack<Object>::operator=(const ArrayStack& other) {
57   if (this != &other) {
58     capacity = other.capacity;
59     t = other.t;
60     delete[] S;
61     S = new Object[capacity];
62     for (int j=0; j <= t; j++)
63       S[j] = other.S[j];
64   }
65   return *this;
66 }
```
Complete homework 3

Complete homework 3
Queue

- ArrayQueue implantation in the class.
Pointers are powerful features of C++ that differentiates it from other programming languages like Java and Python.

Pointers are used in C++ program to access the memory and manipulate the address.
$ git fetch origin

$ git checkout -b week5 origin/week5

If you want to back to hw3 remote, then switch the branch as

$ git branch // list your branches

$ git checkout hw3 // switch to hw3
Example: Address in C++

- The 0x in the beginning represents the address is in hexadecimal form.
- Notice that first address differs from second by 4-bytes and second address differs from third by 4-bytes.
- This is because the size of integer (variable of type int) is 4 bytes in 64-bit system.
Pointers Variables

- C++ gives you the power to manipulate the data in the computer's memory directly. You can assign and de-assign any space in the memory as you wish. This is done using Pointer variables.

- Pointers variables are variables that points to a specific address in the memory pointed by another variable

- Watch Video

- [http://cslibrary.stanford.edu/104/](http://cslibrary.stanford.edu/104/)

- Note: There is no single, official, word for the concept of a pointee — pointee is just the word used in the explanations.
How to declare pointer?

- The statement above defines a pointer variable `p`. It holds the memory address.
- The asterisk is a dereference operator which means pointer to.
- Here, pointer `p` is a pointer to int, i.e., it is pointing to an integer value in the memory address.

```c
int *p;

OR,
int* p;
```
Reference operator (&) and Dereference operator (*)

- Reference operator (&) as discussed above gives the address of a variable.

- To get the value stored in the memory address, we use the dereference operator (*).

- For example: If a number variable is stored in the memory address 0x123, and it contains a value 5.

- The reference (&) operator gives the value 0x123, while the dereference (*) operator gives the value 5.

- Note: The (*) sign used in the declaration of C++ pointer is not the dereference pointer. It is just a similar notation that creates a pointer.
#include <iostream>
using namespace std;

int main()
{
    int *pc, c;

    c = 5;
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;

    pc = &c;    // Pointer pc holds the memory address of variable c
    cout << "Address that pointer pc holds (pc): " << pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

    c = 11;    // The content inside memory address &c is changed from 5 to 11.
    cout << "Address pointer pc holds (pc): " << pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

    *pc = 2;
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;

    return 0;
}
Example Pointers.cpp: C++ Pointers

- When `c = 5`; the value 5 is stored in the address of variable `c` - 0x7fff5fbff8c.
- When `pc = &c`; the pointer `pc` holds the address of `c` - 0x7fff5fbff8c, and the expression (dereference operator) `*pc` outputs the value stored in that address, 5.
- When `c = 11`; since the address pointer `pc` holds is the same as `c` - 0x7fff5fbff8c, change in the value of `c` is also reflected when the expression `*pc` is executed, which now outputs 11.
- When `*pc = 2`; it changes the content of the address stored by `pc` - 0x7fff5fbff8c. This is changed from 11 to 2. So, when we print the value of `c`, the value is 2 as well.
int c, *pc;

pc=c;  /* Wrong! pc is address whereas, c is not an address. */

*pc=&c; /* Wrong! *pc is the value pointed by address whereas, \&c is an address. */

pc=&c; /* Correct! pc is an address and, \&pc is also an address. */

*pc=c; /* Correct! *pc is the value pointed by address and, c is also a value. */
#include <iostream>
using namespace std;

int main()
{
    float arr[5];

cout << "Displaying address using arrays: " << endl;
for (int i = 0; i < 5; ++i) {
    cout << "&arr[" << i << "] = " << &arr[i] << endl;
}

// ptr = &arr[0]
float* ptr;
ptr = arr;

cout << "Displaying address using pointers: " << endl;
for (int i = 0; i < 5; ++i) {
    cout << "ptr + " << i << " = " << ptr + i << endl;
}

for (int i = 0; i < 5; ++i) {
    arr[i] = i;
}

cout << "Displaying value using pointers: " << endl;
for (int i = 0; i < 5; ++i) {
    cout << "arr[" << i << "] = " << "*(p + " << i << "] = " << *(ptr + i) << endl;
}

return 0;
}
Pointers and Functions

- Check the previous Swap Lectures
  - Passing by reference without pointers
  - Passing by reference with pointers
Memory Management

- **Arrays** can be used to store multiple homogenous data but there are serious drawbacks of using arrays.

- You should allocate the memory of an array when you declare it but most of the time, the exact memory needed cannot be determined until runtime.

- The best thing to do in this situation is to declare an array with maximum possible memory required (declare array with maximum possible size expected). The downside to this is unused memory is wasted and cannot be used by any other programs.

- To avoid wastage of memory, you can dynamically allocate memory required during runtime using `new` and `delete` operator in C++.
```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
    int num;
    cout << "Enter total number of students: ";
    cin >> num;
    float* ptr;

    // memory allocation of num number of floats
    ptr = new float[num];

    cout << "Enter GPA of students." << endl;
    for (int i = 0; i < num; ++i)
    {
        cout << "Student" << i + 1 << ": ";
        cin >> *(ptr + i);
    }

    cout << "Displaying GPA of students." << endl;
    for (int i = 0; i < num; ++i) {
        cout << "Student" << i + 1 << ": " << *(ptr + i) << endl;
    }

    // ptr memory is released
    delete [] ptr;

    return 0;
}
```
Example: DynamicArrayClass.cpp

```cpp
#include <iostream>
using namespace std;

class Test {
    private:
        int num;
        float *ptr;
    public:
        Test() {
            cout << "Enter total number of students: ";
            cin >> num;

            ptr = new float[num];

            cout << "Enter GPA of students." << endl;
            for (int i = 0; i < num; ++i) {
                cout << "Student" << i + 1 << ": ";
                cin >> *(ptr + i);
            }
        }
        ~Test() {
            delete[] ptr;
        }

        void Display() {
            cout << "\nDisplaying GPA of students." << endl;
            for (int i = 0; i < num; ++i) {
                cout << "Student" << i + 1 << ": " << *(ptr + i) << endl;
            }
        }
};

int main() {
    Test s;
    s.Display();
    return 0;
}
```