CSCI 2270 – Data Structures

Instructor: Maciej Zagrodzki, Christopher Godley

Assignment 5 - Stacks and Queues

OBJECTIVES

1. **Create, add to, delete from, and work with a stack implemented as a linked list**
2. **Create, add to, delete from, and work with a queue implemented as an array**

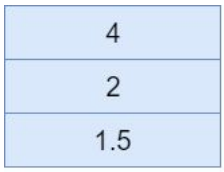
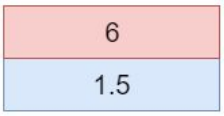
Overview

Stacks and Queues are both data structures that can be implemented using either an array or a linked list. You will gain practice with each of these in the following two mini-projects. The first is to build a simple calculator using a linked list stack and the second is to simulate a multi-process producer-consumer problem using a circular array based queue.

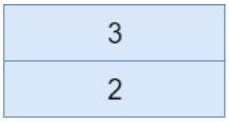
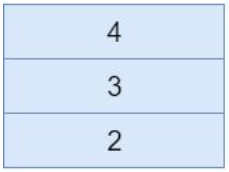
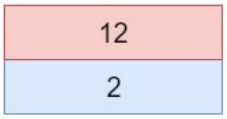
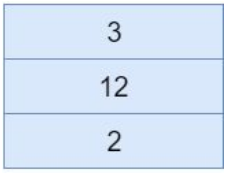
RPN calculator

During the ‘70s and ‘80s, Hewlett-Packard used the so-called Reverse Polish Notation (RPN) in all their calculators. The notation refers to the way that binary operations are processed. Rather than the standard infix notation “1 + 1”, RPN, or postfix notation, is written “1 1 +”. While it may look backward at first, it is actually less ambiguous to write operations this way and it lends itself to an elegant implementation with a stack. As numbers (operands) are entered to the calculator, they are pushed onto a stack. When an operator, e.g. “+”, is entered, the top two elements on the stack are popped, added, and the result is pushed back to the stack. Note that you can load the stack with more than two operands; for example, entering the following sequence of inputs “1.5, 2, 4, +, +” will add 4+2, push the result, 6, to the stack, then add 6+1.5, and push 7.5 to the stack.

Instruction	Stack
#> 1.5	1.5
#> 2	2 1.5

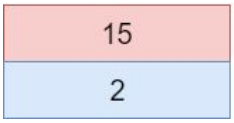
#> 4	
#> +	
#> +	

Similarly, the input sequence “2,3,4, *, 3, +, +” performs the product 4*3, then the sum 3+12, then the sum 15+2 and the resulting stack will have one item: 17.

Instruction	Stack
#> 2	
#> 3	
#> 4	
#> *	
#> 3	

CSCI 2270 – Data Structures

Instructor: Maciej Zagrodzki, Christopher Godley

#> +	
#> +	

Producer-consumer problem

In parallel computing, the producer-consumer problem describes a simple synchronization task involving two processes; a producer, which generates data, and a consumer, which processes data. The problem is that they may work at different speeds, causing one process to wait for the other which is very inefficient. We can solve this problem by creating a data buffer in the form of a queue. When the producer generates some data it pushes it to the queue. The consumer dequeues data and processes it as it comes in. You will simulate this interaction (without parallelism of course) using a circular queue.

Starter Code on Moodle

Do not modify the provided header files. You are provided with skeletons of your driver programs (*RPNCalculatorDriver.cpp, ProducerConsumerDriver.cpp*) which you need to complete and submit on Moodle. Write one corresponding C++ file for each header file that implements all of the class methods (a few of the getter functions are defined already, so you don't have to implement them). The names of these files are below:

- *Provided Starter Code:*
  - RPN Calculator files
    - RPNCalculator.hpp (DO NOT MODIFY)
    - RPNCalculatorDriver.cpp (TODO #1)
  - Producer-Consumer files
    - ProducerConsumer.hpp (DO NOT MODIFY)
    - ProducerConsumerDriver.cpp (TODO #2)
- ***IMPORTANT: Files you need to create:***
  - RPNCalculator.cpp (TODO #3)
  - ProducerConsumer.cpp (TODO #4)

CSCI 2270 – Data Structures

Instructor: Maciej Zagrodzki, Christopher Godley

RPNCalculator Class

You will build an RPN calculator that can perform addition and multiplication on floating point numbers. This class utilizes the functionality of a linked list stack, which employs the following struct (included in *RPNCalculator.hpp*)

```
struct Operand
{
    float number;
    Operand* next;
};
```

Implement the RPNCalculator class according to the following specifications

Operand* stackHead

- ➔ Points to the top of the stack (first node in the linked list). This is defined in the header file.

RPNCalculator()

- ➔ Constructor--set the **stackHead** pointer to NULL

~RPNCalculator()

- ➔ Destructor--**pop** everything off the stack and set **stackHead** to NULL.

bool isEmpty()

- ➔ Returns true if the stack is empty (i.e. **stackHead** is NULL), false otherwise

void push(float number)

- ➔ Insert a new node with **number** onto the top of the stack (beginning of linked list)

void pop()

- ➔ If the stack is empty, print *"Stack empty, cannot pop an item."* and return. Otherwise delete the top most item from the stack

Operand* peek()

- ➔ If the stack is empty, print *"Stack empty, cannot peek."* and return NULL. Otherwise, return a pointer to the top of the stack

bool compute(std::string symbol)

- ➔ Perform the arithmetic operation **symbol**, which will be either “+” or “*”, on the top 2 numbers in the stack. The return value indicates whether the operation was carried out successfully
- ➔ If **symbol** is not “+” or “*”, print *"err: invalid operation"* and return false

- ➔ Store the floats from the top two elements in the stack in local variables and **pop** them.
  - ◆ Before getting each element, make sure that the stack is not empty. If it is empty, print *"err: not enough operands"* and return false
  - ◆ If you pop the first element, and notice that the list is empty prior to getting the next element, print the error message, push the first element back to the stack (use push function), and return false
- ➔ Perform the arithmetic operation **symbol** on those two elements and push the result to the stack

**Important

- ➔ use NULL, and not nullptr.
- ➔ make sure the prompt "#> " (without a newline at the end of prompt) is printed at every input of float or an operand.
- ➔ Only '=' marks the end of expression. Till the end of expression is reached, the code is expected to: consume input, print error message if any, ignore the said erroneous input, and continue onto the next available input.

RPNCalculator main driver file

- ➔ Create a stack by instantiating an RPNCalculator object
- ➔ Prompt the user to enter the operators and operands using the print statement*, "Enter the operators and operands ('+', '*') in a postfix format"*
- ➔ Read user inputs *(Tip: use getline for all inputs)* until **"="** is entered. If the input is a number (**isNumber** function provided in starter code) then push it onto the stack, else, call the ***compute*** function on the input
- ➔ When your program reads **"="**:
  - ◆ If the stack is empty print *"No operands: Nothing to evaluate"* and return
  - ◆ If the expression is invalid print *"Invalid expression"* and return. An expression is considered valid if there is exactly one item left on the stack. *(Hint: pop the last item and check if the stack is empty)*
  - ◆ Otherwise, print the result of the expression (for floating numbers limit the precision to 4 decimals)

Below are some sample outputs

```
SAMPLE OUTPUT 1
Enter the operators and operands ('+', '*') in a postfix format
#> 2.5
#> 3
#> +
#> 2.4
```

CSCI 2270 – Data Structures

Instructor: Maciej Zagrodzki, Christopher Godley

```
#> *
#> =
13.2
```

```
SAMPLE OUTPUT 2
Enter the operators and operands ('+', '*') in a postfix format
#> 15
#> 25
#> *
#> 30
#> =
Invalid expression
```

ProducerConsumer Class

Beware of edge cases that arise from the array being circular

In this class you will build a queue using the circular array implementation. Implement the methods of **ProducerConsumer** according to the following specifications.

std::string queue[SIZE]

- ➔ A circular queue of strings in the form of an array. **SIZE** is initialized to a default value of 20 in *ProducerConsumer.hpp*

int queueFront

- ➔ Index in the array that keeps track of the front item

int queueEnd

- ➔ Index in the array that keeps track of the first available empty space (in case the queue is full, queueEnd points to queuFront).

ProducerConsumer()

- ➔ Constructor--Set **queueFront** and **queueEnd** to 0

bool isEmpty()

- ➔ Return true if the queue is empty, false otherwise

bool isFull()

- ➔ Return true if the queue is full, false otherwise

void enqueue(std::string item)

- ➔ If the queue is not full, then add the **item** to the end of the queue and modify **queueFront** and/or **queueEnd** if appropriate, else print "*Queue full, cannot add new item*"

void dequeue()

➔ Remove the first item from the queue if the queue is not empty and modify **queueFront** and/or **queueEnd** if appropriate. Otherwise print *"Queue empty, cannot dequeue an item"*

int queueSize()

➔ Return the number of items in the queue.

std::string peek()

➔ If the queue is empty then print "*Queue empty, cannot peek*" and return an empty string. Otherwise, return the first item in the queue.

ProducerConsumer main driver file

Your program will start by displaying a menu by calling the **menu()** function which is included in the provided skeleton code. The user will select an option from the menu to decide upon what the program will do, after which the menu will be displayed again. Below are the specifics of the menu

Option 1: Producer (Produce items into the queue) - This is an enqueue operation

- This option prompts the user to enter the number of items being produced using the below print statement

```
cout << "Enter the number of items to be produced:" << endl;
```

- Then prompt the user to enter each item using the below print statement

```
cout << "Item" << <ITEM_NO> << ":" << endl;
```

- Produce (***enqueue***) all the items into the queue

Option 2: Consumer (Consume items from the queue) - This is a dequeue operation

- This option prompts the user to enter the number of items being consumed using the below print statement

```
cout << "Enter the number of items to be consumed:" << endl;
```

- Consume (***dequeue***) items from the queue. If the number of items to be consumed is greater than the total number of items in the queue, then consume (***dequeue***) all the available items in the queue and notify the user with a below statement

```
cout<< "No more items to consume from queue" << endl;
```

This message should never be printed more than once.

- For each item consumed, print the following

```
cout << "Consumed: " << item << endl;"
```

where **item** is the string you are dequeuing.

Option 3: Return the queue size and exit

- This option prompts the user to return the number of items in the queue using the below print statement, before exiting the program.

```
cout << "Number of items in the queue:" <item_count>;
```

Below is a sample output

```
*---------------------------------------*
Choose an option:
1. Producer (Produce items into the queue)
2. Consumer (Consume items from the queue)
3. Return the queue size and exit
*---------------------------------------*
1
Enter the number of items to be produced:
3
Item1
Honey
Item2
Bread
Item3
Butter
*---------------------------------------*
Choose an option:
1. Producer (Produce items into the queue)
2. Consumer (Consume items from the queue)
3. Return the queue size and exit
*---------------------------------------*
2
Enter the number of items to be consumed:
2
Consumed: Honey
Consumed: Bread
*---------------------------------------*
Choose an option:
```

```
1. Producer (Produce items into the queue)
2. Consumer (Consume items from the queue)
3. Return the queue size and exit
*---------------------------------------*
3
Number of items in the queue:1
```