|  |  |  |
| --- | --- | --- |
| American University of Sharjah  College of Engineering  Department of Computer Science & Engineering  P. O. Box 26666, Sharjah, UAE |  | **Instructors:** Dr. Michel Pasquier  **Lab Instructor:** Praveena Kolli  **Office:** EB2-126  **Phone**: 971-6-5152352  **e-mail**: pkolli@aus.edu  **Semester**: Spring 2021 |

**CMP 305 L – Data Structures and Algorithms**

**Lab. Assignment 6 – Stack and Queue**

***Objectives:***

* To understand stacks and to develop a stack-based application
* To understand queues and to develop a queue-based application

***Note:***

***Lab:*** Exercises 1 and 2 (10 marks)

***Bonus*:** Exercise 3 (1 mark)

Stack and Queue libraries

Stack: <http://www.cplusplus.com/reference/stack/stack/?kw=stack>

Queue: <http://www.cplusplus.com/reference/queue/queue/?kw=queue>

***Exercise 1:***

Write a program to check for balancing symbols in the C++ language using stack. Consider (), [], {} symbols only. Your program should prompt the user for a file name that consist of C++ codes.

void main()

{

int ary[10];

for( int i=0;i<=10;i++)

ary[i] = i;

}

Make sure that your program works for all cases. Your program should process three (test1,txt, test2.txt, test3.txt) given text files

Write an appropriate output message that resembles C++ compiler error messages, e.g Error at line 2, [ is not closed in the program.

**Code:**

#include<iostream>

#include<fstream>

#include<stack>

#include<string>

using namespace std;

int main() {

stack<char> mystack;

string path="C:/Users/rohan/Desktop/AUS Year 2/Sem 4/Data Structures and Algorithms/Labs/CMP305\_Repo/Lab6/";

string filename;

cout << "Enter the file to check:" << endl;

cin >> filename;

ifstream input;

input.open(path+filename);

if (input.fail()) {

cout << "Could not find file!" << endl;

exit(1);

}

string line;

int linenum = 1;

while (!input.eof()) {

getline(input,line);

for (char ch : line) {

if (ch == '(' || ch == '{' || ch == '[') {

mystack.push(ch);

}

else if (ch == ')') {

if (mystack.empty())

cout << "Extra ) found at line " << linenum;

if (mystack.top() == '(')

mystack.pop();

else

cout << "Error at line " << linenum << ", incorrect ) found. " << endl;

}

else if (ch == '}') {

if (mystack.empty())

cout << "Extra } found at line " << linenum;

if (mystack.top() == '{')

mystack.pop();

else

cout << "Error at line " << linenum << ", incorrect } found. "<< endl;

}

else if (ch == ']') {

if (mystack.empty())

cout << "Extra ] found at line " << linenum;

else if (mystack.top() == '[')

mystack.pop();

else

cout << "Error at line " << linenum << ", incorrect ] found. "<< endl;

}

}

linenum++;

}

if (!mystack.empty()) {

cout << " Reached end of file. The " << mystack.top() << " was not closed" << endl;

}

else {

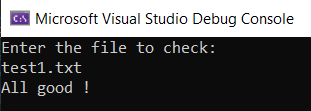
cout << "All good !" << endl;

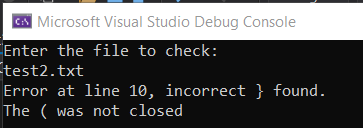
}

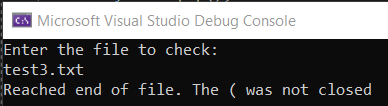
return 0;

}

**Screenshots:**







***Exercise 2:***

You are developing a queue system for Immigration counters at the airport. There are two counters open at this hour to serve travelers. You may assume that both counters have more or less the same number of families.

Each traveler/set of travelers is called an instance of the Family class.

class Family

{

private:

int noOfKids;

int noOfElders ;

int id;

public:

//provide required functions

//Constructors, setter and getters

//overloaded operators such as ==,=,!= .

};

Each Counter is a queue of Family objects.

Sample for counters:

|  |  |
| --- | --- |
| Counter 1 | Counter 2 |
| Family(100,2,3)  ID=100, elders=2, kids=3  Family(200,2)  ID=200, elders=2, kids=0  Family(400,1)  ID=400, elders=1, kids=0 | Family(300,2,1)  ID=300, elders=2, kids=1  Family(500,1,3)  ID=500, elders=1, kids=3  Family(405,1,2)  ID=400, elders=1, kids=2 |

Implement the following two different scenarios,

1. A new counter is opened and you want to move families with more than 3 kids to new counter. Write a function that returns a new queue with corresponding families.
2. A new counter is opened to process the travelers fast. You function should return a new queue after adjusting equal numbers of families in all three queues.

Newly opened counter should be filled by choosing alternate families from existing queues. For example, if existing two counters have 10 families each. New counter should have families in this order- counter1’s 8th family, counter2’s 8th family, counter1’s 9th family, counter2’s 9th family, counter1’s 10th family and counter2’s 10th family.

Note: Moving the families to other counters should be done from the cutoff point of the queue.

**Note**: Family class is provided. You may continue working on the provided skeleton of main function.

**Code:**

#include <iostream>

#include <queue>

using namespace std;

class Family

{

public:

Family(int ide=0, int eld=0, int kid=0) {

noOfElders = eld;

noOfKids = kid;

id = ide;

}

Family(const Family &a) {

noOfKids = a.noOfKids;

noOfElders = a.noOfElders;

id = a.id;

}

Family operator=(Family const &a) {

this->id = a.id;

this->noOfElders = a.noOfElders;

this->noOfKids = a.noOfKids;

return \*this;

}

int getnoOfkids() const {

return noOfKids;

}

int getnoOfElders() const {

return noOfElders;

}

int getid() const {

return id;

}

void setnoOfKids(int x) {

noOfKids = x;

}

void setnoOfElders(int x) {

noOfElders = x;

}

void setid(int x) {

id = x;

}

friend ostream & operator<<(ostream & out, const Family & a)

{

out << "The id of the travelers are: " << a.id << endl;

out << "The number of elders are: " << a.noOfElders << endl;

out << "The number of kids are: " << a.noOfKids << endl;

return out;

}

friend istream &operator >> (istream &in, Family &a) {

in >> a.id;

in >> a.noOfElders;

in >> a.noOfKids;

return in;

}

private:

int id, noOfElders, noOfKids;

};

queue<Family> KidsQueue(queue<Family> &a, queue<Family> &b)

{

queue<Family> tmpA = a, tmpB = b, c;

a = b = c;

while (!tmpA.empty() && !tmpB.empty())

{

if(!tmpA.empty())

{

Family fObj = tmpA.front();

if(fObj.getnoOfkids() >= 3)

c.push(fObj);

else

a.push(fObj);

tmpA.pop();

}

if(!tmpB.empty())

{

Family fObj = tmpB.front();

if(fObj.getnoOfkids() >= 3)

c.push(fObj);

else

b.push(fObj);

tmpB.pop();

}

}

return c;

}

queue <Family> ReOrganize(queue<Family>& a, queue<Family>& b)

{

int cutoff = ((a.size() + b.size()) / 3) / 2;

queue<Family> tmpA = a, tmpB = b, c;

a = b = c;

while (!tmpA.empty() && !tmpB.empty())

{

if(!tmpA.empty())

{

Family fObj = tmpA.front();

if(tmpA.size() <= cutoff)

c.push(fObj);

else

a.push(fObj);

tmpA.pop();

}

if(!tmpB.empty())

{

Family fObj = tmpB.front();

if(tmpB.size() <= cutoff)

c.push(fObj);

else

b.push(fObj);

tmpB.pop();

}

}

return c;

}

//print Queue

ostream &operator<<(ostream &out, const queue<Family> &q) {

queue<Family> tmp = q;

while (!tmp.empty()) {

cout << tmp.front() << endl;

tmp.pop();

}

return out;

}

int main() {

queue <Family> counter1;

queue <Family> counter2;

queue <Family> counter3;

counter1.push(Family(100, 2, 3));

counter1.push(Family(200, 2));

counter1.push(Family(400, 1));

counter1.push(Family(402, 1,4));

counter1.push(Family(789, 2));

counter2.push(Family(300, 2, 1));

counter2.push(Family(500, 1, 3));

counter2.push(Family(405, 3, 2));

counter2.push(Family(309, 1, 3));

counter2.push(Family(567, 2));

cout << "Kids Queue\n\n";

counter3 = KidsQueue(counter1, counter2);

cout << "Counter 3 has: " << endl << counter3;

cout << "Counter 1 after modification has: \n\n" << counter1;

cout << "Counter 2 after modification has: \n\n" << endl << counter2;

cout << "ReOrganize Queue\n\n";

counter3 = ReOrganize(counter1, counter2);

cout << "Counter 1 has: " << endl << counter1;

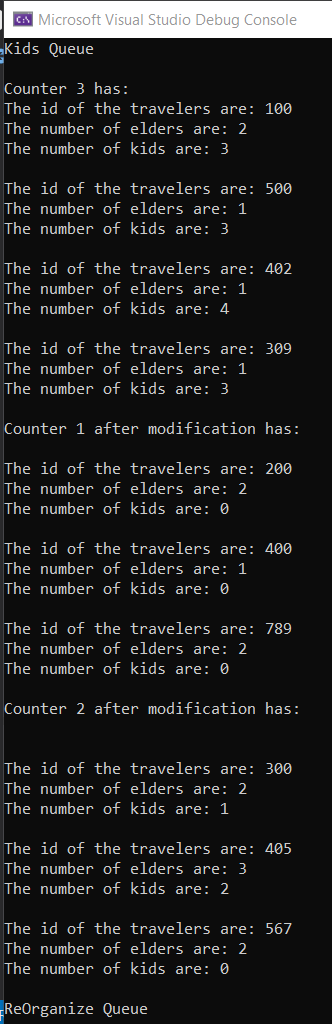
cout << "Counter 2 after modification has: \n\n" << counter2;

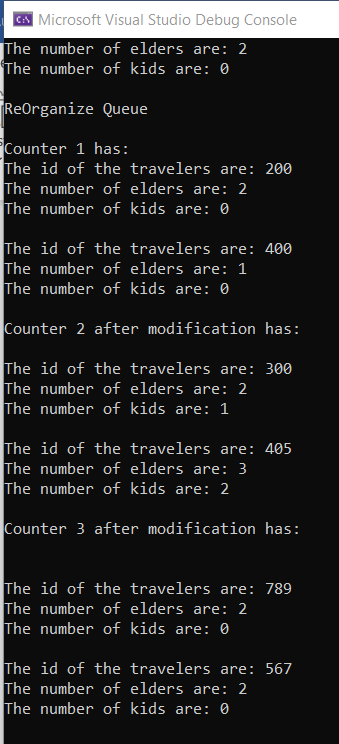
cout << "Counter 3 after modification has: \n\n" << endl << counter3;

return 0;

}

**SCREENSHOTS:**





**Bonus:**

***Exercise 3:***

One of the Stack data structure usages is to evaluate the values of postfix mathematical expressions. Postfix notation is a parenthesis-free way of writing arithmetic expressions, where one places the operator symbol after the operator's two operands. For example, the addition of 3 to 2 is written 3 2 +, and the multiplication of the result by 4 is written as 3 2 + 4 \*. Remarkably, parentheses are never needed. An example like

((3 + 2) \* 4) / (5 - 1)

is written

3 2 + 4 \* 5 1 - /

and gets manually computed as follows:

3 2 + 4 \* 5 1 - /

=> 5 4 \* 5 1 - /

=> 20 5 1 - /

=> 20 4 /

=> 5

We see that an operator evaluates with the two operands that immediately precede it. This explains why the division operator is written last in the original expression, because the division is performed only after all the other sub expressions are evaluated. To automate the computation of a postfix expression, one may use a stack data structure as follows:

Implement an application program that reads a postfix expression from the user and uses a stack to compute its value and returns it to the user. The expression will be assumed to contain integers and the following binary arithmetic operators, namely, + (addition), - (subtraction), \* (multiplication) and / (division).

**Code:**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

#include <stack>

#include <string>

#include <cstring>

using namespace std;

bool isInteger(char \* s)

{

if((s[0] -'0') >= 0 && (s[0] -'0') <= 9)

return true;

else

return false;

}

int main() {

int op1, op2, result;

stack<int> stack;

char pstfix[80];

cout << "Enter post fic notation : ";

cin.getline(pstfix,80);

char \* pch;

cout << "Splitting string "<< pstfix << " into tokens and computing your answer... "<<endl;

pch = strtok (pstfix ," ");

while (pch != NULL)

{

if (isInteger(pch)) {

stack.push((int)\*pch - 48);

}

else {

int a, b;

a = stack.top();

stack.pop();

b = stack.top();

stack.pop();

switch (\*pch) {

case '+':

stack.push(b + a);

break;

case '-':

stack.push(b - a);

break;

case '\*':

stack.push(b \* a);

break;

case '/':

stack.push(b / a);

break;

}

}

pch = strtok (NULL, " ");

}

cout << "The expression evaluates to: "<< stack.top();

return 0;

}

**Screenshot:**

