**University of Michigan – Dearborn**

**CIS 200 – Computer Science 2**

**Lab 09**

Nahrin Sharna

[nsharna@umich.edu](mailto:nsharna@umich.edu)

03/25/2019

# Program 01:

# Source Code:

/\*

Author: Nahrin Sharna

Creation Date: 03/24/2019

Modification Date: 03/25/2019

Purpose: To create a stack and check its functionalities

\*/

#include <iostream>

#include <sstream>

#include <string>

#include <fstream>

using namespace std;

const int MAX = 5;

template <class T>

class Stack {

int top;

public:

T a[MAX]; //Maximum size of Stack

Stack() { top = -1; }

bool push(T x);

T pop();

T peek();

bool isEmpty();

bool isFull();

};

template< class T >

bool Stack<T>::isFull()

{

if ((top + 1) == MAX)

return true;

else

return false;

}

#include <iostream>

template <class T>

bool Stack<T>::push(T x) {

if (Stack<T>::isFull()) {

cout << "Stack Overflow";

return false;

}

else {

a[++top] = x;

return true;

}

}

#include <iostream>

template< class T > T Stack<T>::pop()

{

T val = -1;

if (!Stack<T>::isEmpty())

{

val = a[top];

top -= 1;

}

else

{

cout << "Stack is Empty : ";

}

return val;

}

template <class T>

T Stack<T>::peek() {

return a[top];

}

template< class T > bool Stack<T>::isEmpty()

{

if (top == -1)//is empty

return 1;

else

return 0; //is not empty

}

// This function return the sorted Stack

template <class T>

Stack<T> minMax(Stack<T> input, string mm) {

Stack<T> tmpStack;

//pop function

while (!input.isEmpty()) {

T tmp = input.peek();

input.pop();

if (mm == "MAX") {

while (!tmpStack.isEmpty() && tmpStack.peek() > tmp) {

input.push(tmpStack.peek());

tmpStack.pop();

}

}

else {

while (!tmpStack.isEmpty() && tmpStack.peek() < tmp) {

input.push(tmpStack.peek());

tmpStack.pop();

}

}

tmpStack.push(tmp);

}

return tmpStack;

}

int main() {

ifstream myfile;

myfile.open("data.dat");

string d;

char choice;

string line;

while (getline(myfile, line)) {

istringstream iss(line);

double x, y, min;

Stack<double> input;

while (iss >> x) {

input.push(x);

}

cout << "Do you want to find maximum or minimum value: ";

cin >> d;

if (d == "Minimum") {

Stack<double> tmpStack = minMax(input, "MIN");

//peek

cout << "Checking for the peek function: " << endl;

cout <<"Top is: " << tmpStack.peek() << endl;

//isFull

cout << "Checking the isFull function: " << endl;

bool r1 = tmpStack.isFull();

if (r1 == true) {

cout << "List is full." << endl;

}

else {

cout << "List is not full." << endl;

}

//isEmpty

cout << "Checking for isEmpty function: " << endl;

bool r2 = tmpStack.isEmpty();

if (r2 == true) {

cout << "The stack is empty." << endl;

}

else {

cout << "Stack is not empty." << endl;

//pop

}

//print

cout << "Printing current stack: " << endl;

for (Stack<double> dump = tmpStack; !dump.isEmpty(); dump.pop())

std::cout << dump.peek() << '\n';

//min

cout << "Printing the Minimum value: " << endl;

int i = 0;

if (tmpStack.isEmpty() == true) {

cout << "Stack is empty." << endl;

}

else {

while (!tmpStack.isEmpty()) {

i++;

y = tmpStack.peek();

if (i == 1) {

min = y;

}

tmpStack.pop();

}

cout << endl << "MIN: " << min << endl;

//checking for pop when the stack is empty

cout << "Checking for pop function when the stack is empty: " << endl;

tmpStack.pop();

}

//push

cout << "Adding new element to the empty stack using push: " << endl;

tmpStack.push(4.9);

tmpStack.push(6.4);

tmpStack.push(7.9);

tmpStack.push(9.9);

tmpStack.push(2.9);

for (Stack<double> dump = tmpStack; !dump.isEmpty(); dump.pop())

cout << dump.peek() << '\n';

//push when the list is full

cout << "Checking push when the list is full: " << endl;

tmpStack.push(3.5);

cout << endl;

//pop

cout << "Deleting top element using pop function: " << endl;

tmpStack.pop();

cout << "After using pop function, new list is: " << endl;

for (Stack<double> dump = tmpStack; !dump.isEmpty(); dump.pop())

cout << dump.peek() << '\n';

}

else if (d == "Maximum") {

Stack<double> tmpStack = minMax(input, "MAX");

//peek

cout << "Checking for the peek function: " << endl;

cout << "Top is: " << tmpStack.peek() << endl;

//isFull

cout << "Checking the isFull function: " << endl;

bool r3 = tmpStack.isFull();

if (r3 == true) {

cout << "List is full." << endl;

//checking for push when the list is full

cout << "Checking push function when the stack is full: " << endl;

tmpStack.push(9.7);

}

else {

cout << "List is not full." << endl;

//push

}

//isEmpty

cout << "Checking for isEmpty function: " << endl;

bool r4 = tmpStack.isEmpty();

if (r4 == true) {

cout << "The stack is empty." << endl;

}

else {

cout << "Stack is not empty." << endl;

}

//print

cout << "Printing current stack: " << endl;

for (Stack<double> dump = tmpStack; !dump.isEmpty(); dump.pop())

std::cout << dump.peek() << '\n';

//printing maximum element

cout << tmpStack.peek() << endl;

int i = 0;

double z = tmpStack.peek();

while (!tmpStack.isEmpty()) {

i++;

y = tmpStack.peek();

if (i == 1) {

min = y;

}

tmpStack.pop();

}

cout << endl << "MAX: " << z << endl;

//checking for pop when the stack is empty

cout << "Checking for pop function when the stack is empty: " << endl;

tmpStack.pop();

//push

tmpStack.push(4.9);

tmpStack.push(6.4);

tmpStack.push(7.9);

tmpStack.push(9.9);

tmpStack.push(2.9);

for (Stack<double> dump = tmpStack; !dump.isEmpty(); dump.pop())

cout << dump.peek() << '\n';

//push when the list is full

cout << "Checking push when the list is full: " << endl;

tmpStack.push(3.5);

cout << endl;

//pop when the list is not empty

cout << "Deleting top element using pop function: " << endl;

tmpStack.pop();

cout << "After using pop function, new list is: " << endl;

for (Stack<double> dump = tmpStack; !dump.isEmpty(); dump.pop())

cout << dump.peek() << '\n';

}

}

myfile.close();

system("pause");

return 0;

}

# Initial Test Case:

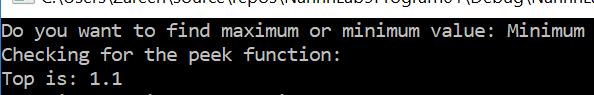
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test # | Valid / Invalid Data | Description of test | Input Value | Expected Output | Actual Output | Test Pass / Fail |
| 1 | Valid | User wants to get the minimum of the list; peek() will check the element at top of the Sorted list | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | Top is: 1.1 |  |  |
| 2 | Valid | User wants to get the minimum of the list; isFull() will check if the list list is full: has atmost 5 elements | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | The list is full |  |  |
| 3 | Valid | User wants to get the minimum of the list; isEmpty() will check if the list list is empty | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | Stack is not empty |  |  |
| 4 | Valid | Print the stack in ascending means the minimum value is on the first of the list and so on the top of the stack | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | 1.1 2.3 4.1 5.2 8.1 |  |  |
| 5 | Valid | Print the minimum value | Choice: Minimum  Sorted stack: 1.1 2.3 4.1 5.2 8.1 | MIN: 1.1 |  |  |
| 6 | Valid | Using pop function when the stack is empty | tmpStack.pop() | Stack is empty |  |  |
| 7. | Valid | Create a new stack using push() function and print it | push(4.9), push(6.4), push(7.9), push(9.9), push(2.9); | New stack:  2.9 4.9 6.4 7.9 9.9 |  |  |
| 8 | Valid | Use push function when the list is full | push(3.5) | Stack overflow |  |  |
| 9 | Valid | Delete the top element using pop() | tmpStack.pop() | New list:  4.9 6.4 7.9 9.9 |  |  |
| 10 | Valid | User wants to get the maximum value; peek() will get the value from the top of the sorted list | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | Top is: 8.1 |  |  |
| 11 | Valid | Print the stack in descending means the maximum value is on the first of the list and top of the stack | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | 8.1 5.2 4.1 2.3 1.1 |  |  |
| 12. | Valid | Print the maximum value | Choice: Maximum  Sorted stack:  8.1 5.2 4.1 2.3 1.1 | MAX: 8.1 |  |  |
| 13 | Valid | User wants to get minimum value; peek() will be used to get the top value | Choice: Minimum  File input:  4 1 13 3 2 | Top is: 1 |  |  |
| 14 | Valid | Print the stack in ascending means the minimum value is on the first of the list and so on the top of the stack | Choice: Minimum  File input:  4 1 13 3 2 | Stack:  1 2 3 4 13 |  |  |
| 15 | Valid | Print the minimum value | Choice: Minimum  Sorted stack:  1 2 3 4 13 | MIN: 1 |  |  |
| 16 | Valid | User wants to get the maximum value; peek() will get the value from the top of the sorted list | Choice: Maximum  File input:  4 1 13 3 2 | Top is: 13 |  |  |
| 17 | Valid | Print the stack in descending means the maximum value is on the first of the list and top of the stack | Choice: Maximum  File input:  4 1 13 3 2 | New stack:  13 4 3 2 1 |  |  |
| 18 | Valid | Print the maximum value | Choice: Maximum  File input:  Sorted stack:  13 4 3 2 1 | MAX: 13 |  |  |

# Final Test Case:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test # | Valid / Invalid Data | Description of test | Input Value | Expected Output | Actual Output | Test Pass / Fail |
| 1 | Valid | User wants to get the minimum of the list; peek() will check the element at top of the Sorted list | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | Top is: 1.1 | Top is: 1.1 | Pass |
| 2 | Valid | User wants to get the minimum of the list; isFull() will check if the list list is full: has atmost 5 elements | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | The list is full | The list is full | Pass |
| 3 | Valid | User wants to get the minimum of the list; isEmpty() will check if the list list is empty | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | Stack is not empty | Stack is not empty | Pass |
| 4 | Valid | Print the stack in ascending means the minimum value is on the first of the list and so on the top of the stack | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | 1.1 2.3 4.1 5.2 8.1 | 1.1 2.3 4.1 5.2  8.1 | Pass |
| 5 | Valid | Print the minimum value | Choice: Minimum  Sorted stack: 1.1 2.3 4.1 5.2 8.1 | MIN: 1.1 | MIN: 1.1 | Pass |
| 6 | Valid | Using pop function when the stack is empty | tmpStack.pop() | Stack is empty | Stack is empty | Pass |
| 7. | Valid | Create a new stack using push() function and print it | push(4.9), push(6.4), push(7.9), push(9.9), push(2.9); | New stack:  2.9 4.9 6.4 7.9 9.9 | New stack:  2.9 4.9 6.4 7.9 9.9 | Pass |
| 8 | Valid | Use push function when the list is full | push(3.5) | Stack overflow | Stack overflow | Pass |
| 9 | Valid | Delete the top element using pop() | tmpStack.pop() | New list:  4.9 6.4 7.9 9.9 | New list:  4.9 6.4 7.9 9.9 | Pass |
| 10 | Valid | User wants to get the maximum value; peek() will get the value from the top of the sorted list | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | Top is: 8.1 | Top is: 8.1 | Pass |
| 11 | Valid | Print the stack in descending means the maximum value is on the first of the list and top of the stack | Choice: Minimum  File input: 1.1 4.1 8.1 5.2 2.3 | 8.1 5.2 4.1 2.3 1.1 | 8.1 5.2 4.1 2.3 1.1 | Pass |
| 12. | Valid | Print the maximum value | Choice: Maximum  Sorted stack:  8.1 5.2 4.1 2.3 1.1 | MAX: 8.1 | MAX: 8.1 | Pass |
| 13 | Valid | User wants to get minimum value; peek() will be used to get the top value | Choice: Minimum  File input:  4 1 13 3 2 | Top is: 1 | Top is: 1 | Pass |
| 14 | Valid | Print the stack in ascending means the minimum value is on the first of the list and so on the top of the stack | Choice: Minimum  File input:  4 1 13 3 2 | Stack:  1 2 3 4 13 | Stack:  1 2 3 4 13 | Pass |
| 15 | Valid | Print the minimum value | Choice: Minimum  Sorted stack:  1 2 3 4 13 | MIN: 1 | MIN: 1 | Pass |
| 16 | Valid | User wants to get the maximum value; peek() will get the value from the top of the sorted list | Choice: Maximum  File input:  4 1 13 3 2 | Top is: 13 | Top is: 13 | Pass |
| 17 | Valid | Print the stack in descending means the maximum value is on the first of the list and top of the stack | Choice: Maximum  File input:  4 1 13 3 2 | New stack:  13 4 3 2 1 | New stack:  13 4 3 2 1 | Pass |
| 18 | Valid | Print the maximum value | Choice: Maximum  File input:  Sorted stack:  13 4 3 2 1 | MAX: 13 | MAX: 13 | Pass |
|  |  |  |  |  |  |  |

# Screenshots:

## Test Case 1:



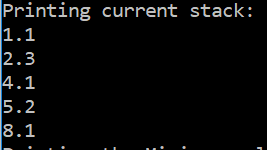
## Test Case 2:



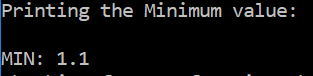
## Test Case 3:



## Test Case 4:



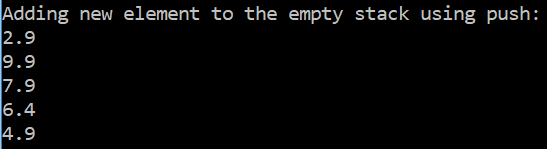
## Test Case 5:



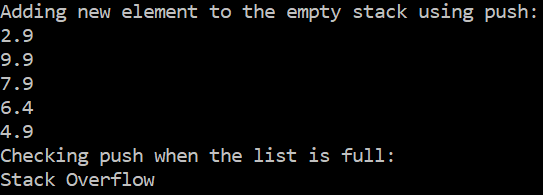
## Test Case 6:



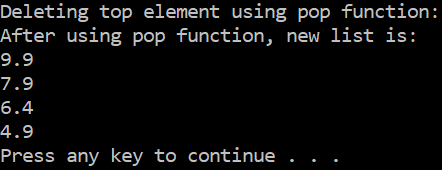
## Test Case 7:



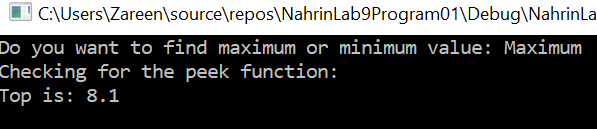
## Test Case 8:



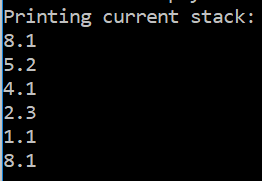
## Test Case 9:



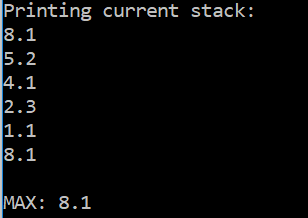
## Test Case 10:



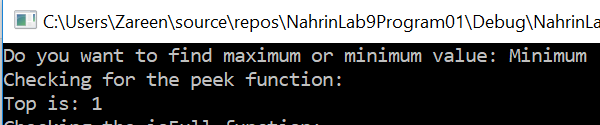
## Test Case 11:



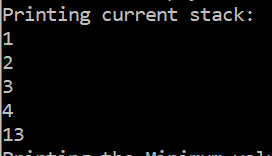
## Test Case 12:



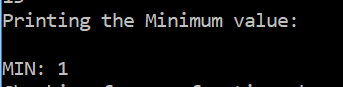
## Test Case 13:



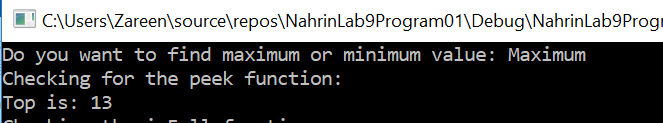
## Test Case 14:



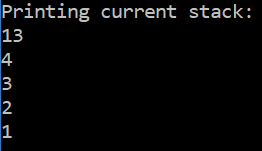
## Test Case 15:



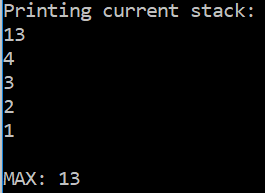
## Test Case 16:



## Test Case 17:



## Test Case 18:



# Program 02

# Source Code:

/\*

Author: Nahrin Sharna

Creation Date: 03/24/2019

Modification Date: 03/24/2019

Purpose: Testing for copy constructor of template classes

\*/

#include <iostream>

#include<string>

using namespace std;

struct Date

{

int day;

int month;

int year;

};

template <typename T> class B;

template <typename T>

class A

{

T valuea;

public:

A() {};

T getValuea()

{

return valuea;

}

void setValuea(T x)

{

valuea = x;

}

A(const A &x)

{

valuea = x.valuea;

}

friend class B<T>;

};

//template class B

template <typename T>

class B : public A<T>

{

T valueb;

public:

B() {};

T getValueb()

{

return valueb;

}

void setValueb(T x)

{

valueb = x;

}

B(const B &x)

{

valueb = x.valueb;

this->valuea = x.valuea;

}

friend ostream& operator<<(ostream& output, const Date& dt);

};

ostream& operator<<(ostream& output, const Date& dt)

{

output << dt.day << '/' << dt.month << '/' << dt.year;

return output;

}

// source file

#include <iostream>

#include<fstream>

#include<string>s

using namespace std;

int main()

{

B<float> b;

b.setValuea(1.34);

b.setValueb(3.14);

cout << "b.setValuea(1.34): " << b.getValuea() << endl

<< "b.setValueb(3.14): " << b.getValueb() << endl;

cout << endl;

//object B of type int

B<int> a;

a.setValuea(1);

a.setValueb(3);

cout << "a.setValuea(1): " << a.getValuea() << endl

<< "a.setValueb(3): " << a.getValueb() << endl;

cout << endl;

//object B of type character

B<char> y;

y.setValuea('a');

y.setValueb('c');

cout << "y.setValuea('a'): " << y.getValuea() << endl

<< "y.setValueb('c'): " << y.getValueb() << endl;

cout << endl;

//object B of type string

B<string> u;

u.setValuea("good");

u.setValueb("morning");

cout << "u.setValuea(good): " << u.getValuea() << endl;

cout << "u.setValueb(morning): " << u.getValueb() << endl;

cout << endl;

//object B of type Date

B<Date> p;

Date d1 = { 27, 10, 2014 };

Date d2 = { 2, 11, 2015 };

p.setValuea(d1);

p.setValueb(d2);

cout << "p.setValuea({27, 10, 2014}): " << p.getValuea() << endl;

cout << "p.setValueb({2, 11, 2015}): " << p.getValueb() << endl;

cout << endl;

system("Pause");

return 0;

}

# Initial Test Case:

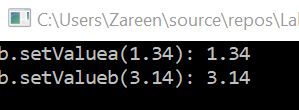
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test # | Valid / Invalid Data | Description of test | Input Value | Expected Output | Actual Output | Test Pass / Fail |
| 1 | Valid | When an instance B of template class is of float data type | valuea = 1.34  valueb = 3.14 | b.setValuea(1.34): 1.34  b.setValueb(3.14): 3.14 |  |  |
| 2 | Valid | When an instance B of template class is of int data type | valuea = 1 valueb = 3 | a.setValuea(1): 1  a.setValueb(3): 3 |  |  |
| 3 | Valid | When an instance B of template class is of character data type | valuea = ‘a’ valueb = ‘c’ | y.setValuea(‘a’): a  y.setValueb(‘b’): b |  |  |
| 4 | Valid | When an instance B of template class is of string type | valuea = “good”  valueb = “morning” | u.setValuea(good): good  u.setValueb(morning): morning |  |  |
| 5 | Valid | When an instance B of template class is of Date data type | valuea = {27,10,2015}  valueb = {2,11,2015} | p.setValuea({27,10,2015}): 27/10/2015  p.setValueb(2,11,2015): 2/11/2015 |  |  |

# Final Test Case:

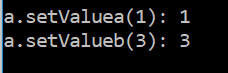
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test # | Valid / Invalid Data | Description of test | Input Value | Expected Output | Actual Output | Test Pass / Fail |
| 1 | Valid | When an instance B of template class is of float data type | valuea = 1.34  valueb = 3.14 | b.setValuea(1.34): 1.34  b.setValueb(3.14): 3.14 | b.setValuea(1.34): 1.34  b.setValueb(3.14): 3.14 | Pass |
| 2 | Valid | When an instance B of template class is of int data type | valuea = 1 valueb = 3 | a.setValuea(1): 1  a.setValueb(3): 3 | a.setValuea(1): 1  a.setValueb(3): 3 | Pass |
| 3 | Valid | When an instance B of template class is of character data type | valuea = ‘a’ valueb = ‘c’ | y.setValuea(‘a’): a  y.setValueb(‘b’): b | y.setValuea(‘a’): a  y.setValueb(‘b’): b | Pass |
| 4 | Valid | When an instance B of template class is of string type | valuea = “good”  valueb = “morning” | u.setValuea(good): good  u.setValueb(morning): morning | u.setValuea(good): good  u.setValueb(morning): morning | Pass |
| 5 | Valid | When an instance B of template class is of Date data type | valuea = {27,10,2015}  valueb = {2,11,2015} | p.setValuea({27,10,2015}): 27/10/2015  p.setValueb(2,11,2015): 2/11/2015 | p.setValuea({27,10,2015}): 27/10/2015  p.setValueb(2,11,2015): 2/11/2015 | Pass |

# Screenshots:

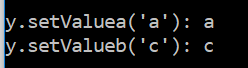
## Test Case 1:



## Test Case 2:



## Test Case 3:



## Test Case 4:



## Test Case 05:

