CS319: Scientific Computing (with C++)

CS319 Labs 4+5: Solving a puzzles with Classes and Strings

16+23 March 2021

Goal: The goals of Lab 4 (16 March) and Lab 5 (23 March) are to develop your proficiency at

(i) Programming classes and objects by further developing the `MyStack` class from Week 4. See `http://www.maths.nuigalway.ie/~niall/CS319/Week04/05MyStackConstructor.cpp`.
(ii) Manipulate strings, and
(iii) Read data from files.

You should aim to complete at least Sections 1 and 3 during Lab 4.
Submit your code on Blackboard (Labs... Lab 4+5) by 5pm, Monday 29 March. Your program file should include your name, ID number, email address, and a short description of how the code works.
Collaboration is encouraged. Include the name and email address of any person you collaborated with on the assignment, and a statement of what each of you contributed.

1. The `MyStack` Class

(i) Review the `MyStack` example from Week 4.
Modify it so that it uses a *destructor* to de-assign memory for the `contents` array.

(ii) Add a member function (method) that checks if the stack is empty.
Return type should be `bool`.

(iii) The maximum number of items we can add to the stack is controlled by `MAX_STACK`. Add a method that can be used to check if the stack is full.
Again, the return type should be `bool`.

(iv) Change the `MyStack::pop()` code so that, if you try to pop an empty stack, it gives a warning message and returns `NULL`.

(v) Modify the `MyStack::push()` code in a similar way, so that you can't put more items onto a stack than there is space for. Here (for now) you don't need a return value.

2. The `string` class

Now we'll look at the "built-in" `string` class in C++. By including the `string` header file, the programmer can declare objects of the class `string`, for storing words etc.
There are numerous methods associated with the class `string`. For example, if *`Name`* is of type `string`, then `Name.size()` returns the number of bytes used to store it (which is essentially the same as the number of `char`s stored in it). The return type is `unsigned int`.
Because of *operator overloading*, we can do the following with `string`s

- use `=` to perform an assignment.
- use `==` and `!=` to compare two strings,
- add a single character to a string using `+=`
- access the kth character of, say, a string called *`Name`* with `Name[i]`

2. The `string` class

Test your `MyStack` class, and use of strings, by writing a program that prompts the user to input a word and then checks if it is a palindrome.
(Try this in your own time) One can use a stack when converting a number form decimal to binary. This is described at

`http://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues`; see Section 1.2.1
Test your class stack by writing a programme that prompts the user for an (unsigned) integer, and then calls a function that implements the algorithm described in WikiBooks to compute a `string` that is the binary representation of the decimal integer.

3. Class templates

The assignment for Lab 4+5 is based around solving a maze problem. For that, it will be helpful to have a stack class that can store integers. But our implementation of `MyStack` is for `char`acters only.
Rather than write a whole new class, we are going to use a `template`, that is rather like the function `template` from Week 5. This idea extends directly to classes.
Insert `template <class T>` before the declaration of your class, `MyStack`.
Change the date-type of `*contents`, the return type of `pop` and argument type of `push` to `T`.
Once complete, we'll be able to define a stacks for `char`s and `int`s with

```
MyStack <char> A;
MyStack <int> A;
```

3. Class templates

However, we have some work to do yet: we also have to indicate that methods belonging to the class are also associated with the template. For example, we have to change

```
void MyStack::push(int c)
```

to

```
template <class T> void MyStack<T>::push(T c)
```

4. Solving a maze

One can use a stack to solve a maze puzzle. Our version will involve starting at a given point, and searching the maze to find an item. Our maze will be represented as a integer matrix, where

- An entry of 0 is a wall,
- 1 is a path, and
- 2 is the starting point, and
- 3 is the item we are searching for.

An example of such as maze matrix is

```
int M[6][6] = {
   { 0, 0, 0, 0, 0, 0 },
   { 0, 2, 0, 1, 1, 0 },
   { 0, 1, 1, 1, 0, 0 },
   { 0, 0, 1, 0, 3, 0 },
   { 0, 1, 1, 1, 1, 0 },
   { 0, 0, 0, 0, 0, 0 } };
```

The algorithm proceeds as follows:

4. Solving a maze

1. Define the two-dimensional `int` array M as given above, and two `int` stacks which will store the row and column index of the locations we have checked.
2. Search the array `M` to locate the entry `2` to find your starting point.
3. Push the current location (i.e., the row and column index) onto the (`<int>`) stacks.
4. Check if the value stored at the current location is a `3`; if it is, we are done. If not, set it to `2`.
5. Check the current location's four neighbours in order (e.g., clockwise from north). If any is a `1` or a `3`, pop that location onto the stack and move there next.
6. If none of the neighbours store a `1`, we are at a dead-end: set the value at that location to `0`, and pop the new location from the stack.

When the item has been located, the program should report the path between it and the starting point.

5. The maze

In your first version of the Maze programme in Section 4 the maze is hard-coded into the program.
Now we wish to store it in a file. The format of the file is as follows.

- Line 1 has two integers which are, respectively, the number of rows and columns in the maze.
- The remaining lines represent the maze.

For example, here is the contents of the file Maze6.txt from http://www.maths.nuigalway.ie/~niall/CS319/lab4 which contains the data for the maze example in Section 4

```
6 6
000000
020110
011100
001030
011110
000000
```

A simple program to load the data is given at
LoadMazeData.cpp.

5. The maze

Two aspects of it are slightly tricky:

(i) we need to dynamically allocate memory for a 2D array; or, rather; for an array of arrays.

(ii) we will read the data as `char`s and convert to `int`. This is because, whenever I code in C or C++, I find it most robust to read all data as `char`s, and then convert it. This is to handle non-printable characters. But if you can read the data directly as `int`s, then do.

6. Assignment

Write a program that solves the Maze problem. It should

1. use your `MyStack` class from Section 1, extended to stores `int`s, as in Section 3.
2. be capable of solving the maze in `Maze20.txt`, which you can download from `http://www.maths.nuigalway.ie/~niall/CS319/lab4`. (You should also test it on the data in `Maze6.txt` and `Maze10.txt`). That is, it should find a path from the starting point to the end point, and then report what that path is.

Deadline: 5pm, Monday 29 March.