C++ course – Exercises Set 3

Wouter Verkerke (Jan 2023)

Exercise 3.1 – Make Stack a proper class

The goal of this exercise is to turn `struct Stack` *into a proper class. In this exercise you will learn about access control and using constructors and destructors for automatic initialization and cleanup.*

- Copy the input file `Stack.cc` to your working area. This file contains the `struct Stack` implementation as shown in the course and will be your starting point for this exercise.

- First reorganize the code to improve the modularity: split the code inside file `Stack.cc` into three files: `Stack.h`, `Stack.cc`, `main.cc`:

    - Move the `struct` definition to the header file, and change it from a definition into a declaration. To do so, you should truncate the given definition of the member functions into declarations of those functions.

      It is however allowed to keep the full definitions of selected member functions in the class declaration. This is mostly done for member functions whose definition is trivial (e.g. one line of code).

      In the given struct stack the only two non-trivial member functions are `push()` and `pop()`: so these two should be truncated to a definition in `Stack.h`.

      The definition of `pop()` and `push()`, which are now eliminated from `stack.hh` should be placed in file `stack.cc`. The definitions of these functions in `stack.cc` while be outside the scope of '`struct stack`' therefore the their names should be explicitly prefixed with '`Stack::`' to signal that these are not simple functions, but rather member functions of `struct Stack`.

    - Move the test program to file `main.cc`.

    - Add proper `#include` statements in `Stack.cc` and `main.cc`. Why do you need to include `Stack.h` in both these files?

    - Compile your code and check that it still works.

- Now change `struct Stack` into **`class`** `Stack`.

  Think about which data and function members should be public and which should be private.

  Implement access control in the class using the keywords `public` and `private` according to your plan.

  Reorganize the class declaration such that all public members are on the top and all private members are on the bottom.

- Add a *constructor* that handles initialization and cleanup associated with class `Stack` in these functions.

  You can call the existing `init()` function from the constructor.
  Should `init()` still be a public function once the constructor is added?

- Next, add a new member function that allows the user of class stack to look at its contents.

  Introduce a `void inspect() const` member function that prints the current stack contents vertically in the correct orientation (i.e. the first line printed is the top of the stack).

  Print for each item both the position in the stack as well as its value.

- Finally modify your `main()` program so that it inspects the buffer after you've pushed some content it into it, and again after you've popped all content from it from it. Convince yourself that it makes sense.

- Change the main program to write 100 elements in the stack instead of 10. Do you understand what happens?

Exercise 3.2 – Add auto grow feature to Stack

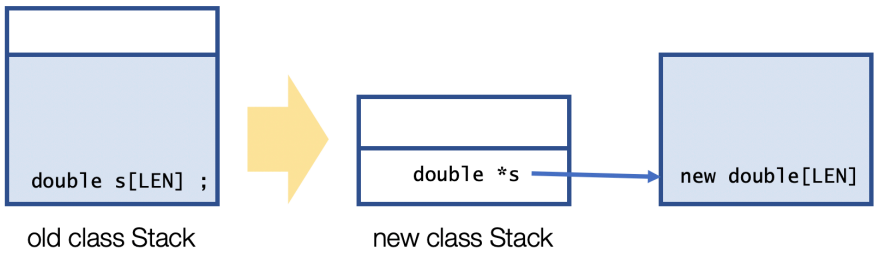
The goal for this exercise is to automatically grow the Stack internal buffer when it runs out of allocated space: The fact that our Stack can be full is an artifact of the implementation and has nothing to do with the abstract concept of a stack. You will now change your code so that it never runs out of memory again (barring physical memory limits)

- Replace the current fixed-size storage buffer of class Stack buffer with a variable buffer size that is allocated dynamically through the `new[]` operator.

  There are several steps to this process

1. Replace the array-type data member `double s[LEN]` with a pointer-type data member with the same name.

   The design goal of this step is to no longer store the buffer data of class stack inside the memory of the class itself, but rather in a separate piece of memory that can be dynamically allocated (and if needed be replaced). Schematically:



   Do you need to change any of the code in the member functions of class Stack to reflect the change `double s[LEN]' → double *s` ?

2. You need to explicitly manage the creation and deletion of the 'external' buffer memory in such a way that it is synchronous with the lifetime of the `Stack` object itself: Add code to the constructor that allocates the external buffer, and stores the address of the allocated member in the pointer data member `s`. You can initially allocate the external buffers with a fixed size of 80 in the constructor (you will change this later).

   Introduce a destructor member function to class Stack. In this function add code that deletes the memory that data member s points to.

3. Given that every object of class stack can now (potentially) have a different buffer size, you will need to change you code to keep track of the amount of buffer memory that was allocated: add a new data member to class stack that memorizes how much memory was allocated (which is different from the amount of memory actually used, which is already tracked), and change the code that checks if the stack is full to use this information of stored size of the buffer, instead of assumed fixed length.

- Run the `main()` program to verify that the new Stack class works.

- Next, augment the functionality of class stack: Let the user of `class Stack` to specify what the size of the buffer should be. To do so, modify the constructor of `Stack` so that it takes an *optional* integer argument with the desired initial stack size

- Finally, we aim at the ultimate design goal: a buffer that can be resized on the fly, so that it can never run out of storage.

1. Implement a new member function `grow(int delta)` function whose functionality will be to increase the existing buffer size by an amount `delta`. The idea is that we can use this function later to enlarge our buffer on the fly if we are about to run out of space.

2. The function `grow()` should allocate a new buffer using the `new[]` operator that is larger than the existing buffer by an amount delta.

3. The function should copy the content of the existing buffer to the new (larger) buffer

4. Then, it should delete the old buffer, and change pointer-typed data member `s` to point to the new buffer instead of the old buffer.

- Convince yourself that the `grow()` as designed above does not introduce a memory leak. *Who deletes the memory allocated inside grow function?*

- Now you are ready to modify the `push()` method so that it will exploit `grow()` to expand the buffer size when the buffer is full.

  Insert code at the beginning of push that checks if the buffer is full, and if so, calls the `grow()` method to expand the buffer by some amount `delta`.

  Think about what is a reasonable value for the growth increment `delta`.

  Explain what the advantages and disadvantages of a small and a large increment size are.

- Run the main program again and verify that the buffer is expanded on the fly to accommodate large input by calling the `inspect()` function you wrote earlier.

- Is it still necessary for the user of a stack to specify an initial size?
  Apart from necessity, why would it be it *useful* for the user to be able to specify an initial size?

Exercise 3.3 – Add auto grow feature to Stack (optional)

The goal of this exercise is to make `Stack` *a properly behaved class under all circumstances: A user of* `Stack` *should be able to make a copy of it without unintended side effects.*

Verifying the behavior

- Go to the `main()` function of your test program and modify it such that it fills the stack with 10 elements.
- Next, after the `Stack s` has been filled, make a copy of it using the copy constructor which you call `sclone`.

The intended effect of copying a `Stack` is that both the `Stack` structure and its contents are copied and two completely independent instances of class `Stack` exist: `s` and `sclone`. This means that you should be able to manipulate one of them without affecting the other.

- As a first step towards verifying this behavior, check that `s` and `sclone` have identical contents (use the `inspect()` function). Does it look OK?
- Now we will explicitly test the independence of `s` and `sclone`: Empty `s` by calling `pop()` repeatedly. Once `s` is empty, `inspect()` both `s` and `sclone` again. Does it look OK?
- As a final test, refill `s` again with 5 elements of value `100*i` instead of value `i*i`. Inspect both `s` and `sclone` again. Do you understand what happens?

Fixing the behavior

- Declare and implement a copy constructor for class `Stack`. In the copy constructor, take care to explicitly copy the value of all data members, except for the buffer pointer `s`. Why should buffer `s` be treated differently in the copy constructor?
- Allocate a sufficiently large buffer `s` in the copy constructor and copy the contents of the other `Stack` into the new buffer. Then run the `main()` program again with all the tests that you inserted for this exercise. Does it behave correctly now?

From a 'clean programming' perspective it is desirable to avoid duplication of code as much as possible.

- Do you see duplication of code between your copy constructor and other member functions of `Stack`? Rewrite the copy constructor so that it uses the `init()` function.