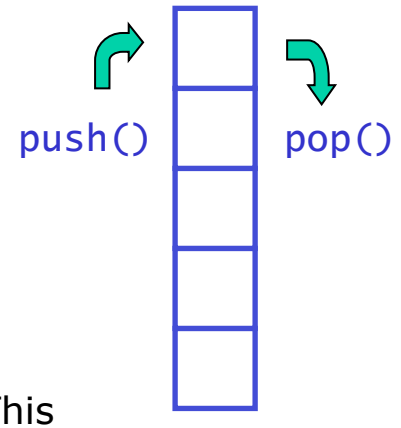
Exercise 3.1 – Make `Stack` a proper class



- Goal: 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.
- Approach
  a) Copy the file `ex3.1/input/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.
  b) First reorganize the code to improve the modularity: split the code in `Stack.cc` into three files: `Stack.h`, `Stack.cc`, `main.cc`: Move the class declaration to the header file, but keep the implementation of Stack functions `push()` and `pop()` in `Stack.cc` (i.e. outside the class definition). Move the test program to file `main.cc`. Add proper `#include` statements in `Stack.cc` and `main.cc`. Compile your code and check that it still works.
  c) 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. Organize the class declaration such that all public members are on the top and all private members are on the bottom.

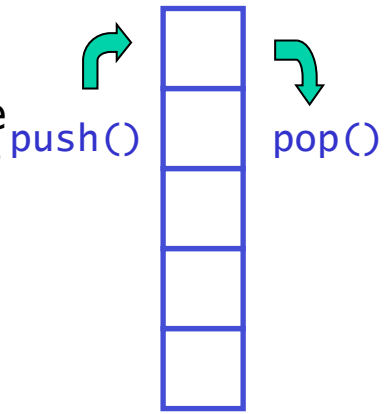
Exercise 3.1 – Make `Stack` a proper class

d) Add a *constructor* and *destructor* and handle initialization and cleanup in these functions. You can call the existing `init()` function from the constructor. Should `init()` still be public?

e) Next we add a new member function that allows the user of class stack to look at its contents. Add an `inspect()` member function that prints the current stack contents vertically in the correct orientation (i.e. top of the stack on top). Print for each item both the position in the stack as well as its value.

f) Finally modify your `main()` program so that it inspects the buffer after you've written into it, and after you've read from it. Convince yourself that it makes sense.

g) 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

- Goal for this exercise: automatically grow the `Stack` internal buffer when it runs out of memory
  - 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. We will now change our code so that it never runs out of memory again (barring physical memory limits)
- Implementation plan in steps

  a) Reimplement `Stack` buffer with a variable buffer size through the use of dynamic memory allocation for its internal buffer. To do this you will need to modify
     - the constructor, destructor,
     - the type of the data member `s` and change the way the length of the buffer `s` is stored (why?).

     Allow the user to specify the size of the buffer as an argument to the `Stack` constructor.

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

Exercise 3.2 – Add auto grow feature to Stack

c) Implement a `grow(int delta)` function that resizes the buffer `s` by 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. The function `grow()` should

- Allocate a new buffer that is larger by amount `delta`.
- Copy the content of the existing buffer to the new buffer.
- Delete the old buffer and change pointer data member `s` to point to the new buffer.

- Convince yourself that `grow()` does not introduce a memory leak. Who deletes the memory allocated by grow?

d) Now we 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 it is calls the `grow()` method to expand the buffer. 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.

e) Run the main program again and verify that the buffer is expanded on the fly to accommodate large input using `inspect()`.

f) Is it still necessary for the user of a stack to specify an initial size? Modify the constructor so that the user doesn't need to specify an initial size but still can if he wants to. Why is it useful for the user to be able to specify an initial size?

Exercise 3.3 – The copy constructor

- 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 effect, but we have never verified if this works properly
- Approach – Verifying the behavior
  a) Go to the `main()` function of your test program and modify it such that it fills the stack with 10 elements.
  b) Next, after the `Stack s` has been filled, make a copy of it using the copy constructor which you call `sclone`.
  c) 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?
  d) 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?
  e) 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?

Exercise 3.3 – The copy constructor

- Approach – fixing the behavior

  f) 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?

  g) Allocate a sufficiently large buffer `s` in the copy constructor and copy the contents of the other `Stack` into the new buffer.

  h) Then run the `main()` program again with all the tests that you inserted for this exercise. Does it behave correctly now?

  i) 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`?

  j) Rewrite the copy constructor so that it uses the `init()` function.

Exercise 3.4 – Static members

- Goal: write a class that counts the number of instances that exist of it
  - Examine the code below

```
int main() {
  Counter a ;
  Counter b ;
  cout << "there are now "
       << Counter::getCounter()
       << "Counter objects" << endl ;
  if (true) {
     Counter c ;
     cout << " and now " << Counter::getCounter() ;
  }
  cout << " and now " << Counter::getCounter() << endl ;
}
```

- Approach – Write a class `Counter` that makes the above fragment of code work.
  a) Write a dummy class `Counter` with an empty constructor and destructor.

Exercise 3.4 – Static members

b) Your class needs to have an integer variable that counts the number of class instances. This variable should be shared between all instances. You can use a `static int` data member to accomplish this. Be sure to both declare the static variable in the class declaration and initialize this variable to zero in its definition.
   - Why doesn't the initialization belong to the class declaration?

c) Modify the constructor and destructor such that they increment and decrement the counter by one respectively.

d) Implement the `getCounter()` method so that it works as used in the example fragment. Note that this implies that you can call `getCounter()` without an available `Counter` object instance, so this function needs to be declared static as well.