# Project 7: Stack

***A generic Stack data structure***

**Revision dated 01/04/18**

**Educational Objectives:** After completing this assignment the student should have the following knowledge, ability, and skills:

- Define the concept of Generic Container
- Define the ADT Stack with elements of type T and maximum size N
- Give examples of the use of Stack in computing
- Describe an implementation plan for generic Stack as a class template `Stack<T,N>` based on a dynamically allocated array of `T` objects
- Code the implementation for `Stack<T,N>` using the plan

**Operational Objectives:** Create the generic container class `fsu::Stack<T,N>` that satisfies the interface requirements given below, along with an appropriate test harness for the class.

**Deliverables:** Four files `tstack.h`, `fstack.cpp`, `makefile`, and `log.txt`.

**Assessment Rubric**

```
=============================================
builds:   [0..2 pts each]
  fstack.x [student harness]              x
  fstack_char.x                           x
  fstack_int.x                            x
  fstack_String.x                         x
  fstack_T.x                              x
tests:    [0..5 pts each]
  fstack_char.x                           x
  fstack_int.x                            x
  fstack_String.x                         x
  fstack_T.x                              x
log.txt       (assessed with project 8)   x
code quality  (assessed with project 8)   x
dated submission deduction [2 pts each]: ( x)
                                          --
total                          [0..40]:  xx
=============================================
```

Note: Projects 7 and 8 will be assessed together for a total of 100 points for both projects. The above is only an indication of the weighting of the first part of the coupled assignment.

**Abstract Data Types**

An *abstract data type*, abbreviated ADT, consists of three things:

1. A set of elements of some type `T`
2. Operations that may modify the set or return values associated with the set
3. Rules of behavior, or axioms, that determine how the operations interact

The operations and axioms together should determine a unique characterization for the ADT, so that any two implementations should be essentially equivalent. (The word *isomorphic* is used to give precision to "essentially equivalent". We'll look at this in the next course.)

**Stacks**

The *stack* ADT is used in many applications and has roots that pre-date the invention of high-level languages. Conceptually, stack is set of data that can be expanded, contracted, and accessed using very specific operations. The stack ADT models the "LIFO", or last-in, first-out, rule. The actual names for the stack operations may vary somewhat from one description to another, but the behavior of the abstract stack operations is well known and unambiguously understood throughout computer science. Stacks are important in many aspects of computing, ranging from hardware design and language translators (compilers) to algorithm control structures.

Typical uses of ADT Stack are (1) runtime environment for modern programming languages (facilitating recursive function calls, among other things), (2) control of the depth first search and backtracking search algorithms, (3) hardware evaluation of postfix expressions, and (4) various compiler operations, such as converting expressions from infix to postfix.

**Abstract Stack Interface**

The stack abstraction has the following operations and behavior:

- **`Push(t)`** Inserts the element `t` into the stack
- **`Pop()`** Removes the last-inserted element; undefined when stack is empty
- **`Top()`** Returns the last-inserted element; undefined when stack is empty
- **`Empty()`** Returns true iff the stack has no elements
- **`Size()`** Returns the number of elements in the stack

**Stack Implementation Plan**

We will implement the stack abstraction as a C++ class template

```
template < typename T , size_t N >
Stack;
```

with the following public methods:

```
// Stack < T , N > API
void     Push       (const T& t); // push t onto stack; error if full
T        Pop        ();           // pop stack and return removed element; error if stack is empty
T&       Top        ();           // return top element of stack; error if stack is empty
const T& Top        () const;     // const version
size_t   Size       () const;     // return number of elements in stack
size_t   Capacity   () const;     // return storage capacity [maximum size] of stack
bool     Empty      () const;     // return 1/true if stack is empty, 0/false if not empty
void     Clear      ();           // make the stack empty
```

There should be a full complement of self-management features:

```
Stack              ();             // default constructor
Stack              (char ofc, int dir); // sets ofc_ and dir_
Stack              (const Stack&); // copy constructor
~Stack             ();             // destructor
Stack& operator =  (const Stack&); // assignment operator
```

The element and size data will be maintained in private variables:

```
private:
  const size_t capacity_;  // = N = size of array   - fixed during life of stack
  T *          data_;      // dynamically allocated array of T objects    - where T objects are stored
  size_t       size_;      // current size of stack - dynamic during life of stack
```

The display and its settings will be maintained as follows:

```
public:
  void     Display    (std::ostream& os) const; // output stack contents through os - depends on ofc_ and dir_
  void     SetOFC     (char ofc);
  void     SetDIR     (int dir);
  void     Dump       (std::ostream& os) const; // output all private data (for development use only)

private:
  char    ofc_;
  int     dir_;
```

The class constructors will have responsibility for initializing variables and allocating the array `data_`. Note that `capacity_` is a constant, so it must be initialized by the constructor in the initialization list and it cannot be changed during the life of a stack object; `capacity_` should be given the value passed in as the second template argument `N`. The destructor will have the responsibility for de-allocating memory allocated to `data_`. Values stored in the `data_` array and the `size_` variable will be correctly maintained by the push and pop operations, using the "upper index" end of the data as the top of the stack. The data in the stack should always be the array elements in the range `[0..size_)`, and the element `data_[size_ - 1]` is the top of the stack (assuming `size_ > 0`).

Please note that the dynamically allocated array `data_` must be allocated by constructors at runtime and de-allocated by the destructor at runtime. Thus the underlying "footprint" of the stack object remains fixed as the size changes, even when the size is changed to zero. The only calls to operators `new` or `delete` should be in constructors and destructors in this implementation.

This implementation will have the requirement on clients that the maximum size required for the stack is known in advance and determined by the second template argument - see requirements below.

**Procedural Requirements**

1. Begin your log file named `log.txt`. (See Assignments for details.)
2. Create and work within a separate subdirectory `cop3330/proj7`. Review the COP 3330 rules found in Introduction/Work Rules.
3. After starting your log, copy the following files from the course directory `[LIB]` into your `proj7` directory:

   ```
   proj7/*
   area51/fstack*
   ```

4. Define and implement the class template `fsu::Stack<T,N>` in the file `tstack.h`. Be sure to make log entries for your work sessions.
5. Devise a test client for `Stack<T,N>` that exercises the Stack interface for at least one native type and one user-defined type `T`. Repair your code as necessary. Put this test client in the file `fstack.cpp`. Be sure to make log entries for your work sessions.
6. Turn in `tstack.h`, `fstack.cpp`, `makefile`, and `log.txt` using the `submit.sh` submit script.

   ***Warning:*** *Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

**Code Requirements and Specifications**

1. Your `makefile` should build the target `fstack.x`. If you have other targets to build, that is not a problem as long as this target is there.

2. `Stack` should be a proper type, with full copy support. That is, it should have a public default constructor, destructor, copy constructor, and assignment operator. Be sure that you test the copy constructor and assignment operator.

3. The `Stack` constructor should create a stack that is empty but has the capacity to hold `N` elements, where `N` is the second template parameter with type `size_t`. Note that this parameter should be given the default value of 100. This has the effect of making a declaration such as

   ```
   fsu::Stack<int> s;
   ```

   legal and create a stack with capacity 100.

4. Use the implementation plan discussed above. No methods or variables should be added to the class, beyond those specified above and in the implementation plan.

5. The `Display(os)` method is intended to output the contents through the `std::ostream` object `os` in either bottom-to-top order or top-to-bottom order, depending on the sign of the variable `dir_`. The variable `ofc_` is a single *output formatting character*. (The four popular choices for `ofc` are `'\0'`, `' '`, `'\t'` and `'\n'`.) Default values for these class variables should be `dir_ = 0` and `ofc_ = '\0'`.

   The implementation of `Display` must recognize two cases:

   i. `ofc_ == '\0'`: send the contents to output with nothing between the elements
   ii. `ofc_ != '\0'`: send the contents to output with `ofc_` preceeding each element of the stack

   The direction variable `dir_` is an integer indicating the direction in which to display the stack. Again `Display` must recognize two cases:

   i. `dir_ >= 0`: send the contents to output in bottom-to-top order
   ii. `dir_ < 0`: send the contents to output in top-to-bottom order

   Thus, for example, in default mode `s.Display(std::cout)` would send the contents of `s` to standard output in bottom-to-top order with no separating character. After the calls `SetOFC('\n')` and `SetDIR(-1)`, `s.Display(std::cout)` would send the contents with a newline preceding each element and in top-to-bottom order.

   Note that top-to-bottom order is more natural for vertical displays (`ofc_ = '\n'`) and bottom-to-top more natural for horizontal displays (`ofc_ = '\0', ' ', or '\t'`).

6. The output operator should be overloaded as follows:

   ```
   template < typename T , size_t N >
   std::ostream& operator << (std::ostream& os, const Stack<T,N>& s)
   {
     s.Display (os);
     return os;
   }
   ```

7. The class `Stack` should be in the `fsu` namespace.

8. The file `tstack.h` should be protected against multiple reads using the `#ifndef ... #define ... #endif` mechanism.

9. The test client program `fstack.cpp` should adequately test the functionality of stack, including the output operator. It is your responsibility to create this test program and to use it for actual testing of your stack data structure.

**Hints**

- Your test client can be modelled on the harness `fbitvect.cpp` distributed as part of a previous assignment.

- Keep in mind that the implementations of class template methods are in themselves template functions. For example, the implementation of the `Stack` method `Pop()` would look something like this:

  ```
  template < typename T , size_t N >
  T Stack<T,N>::Pop()
  {
    // yada dada
    return ??;
  }
  ```

- We will test *your* implementation `tstack.h` using *our* test client `fstack.cpp`.

- There are two versions of `Stack::Top()`. These are distinguished by "`const`" modifiers for one of the versions. The implementation code is identical for each version. The main point is that "`Top()`" can be called on a constant stack, but the returned reference may not be used to modify the top element. This nuance will be tested in our assessment. You can test it with two functions such as:

```
char ShowTop(const fsu::Stack<char>& s)
{
  return s.Top();
}

void ChangeTop(fsu::Stack<char>& s, char newTop)
{
  s.Top() = newTop;
}
```

Note that `ShowTop` has a `const` reference to a stack, so would be able to call the const version of `Top()` but not the non-const version, but that suffices. `ChangeTop` would need to call the non-const version in order to change the value at the top of the stack. A simple test named "`constTest.cpp`" is posted in the distribution directory.

- It is possible to trap stack "overflow" and "underflow" errors in Pop and Top without resorting to a program exit. The tricks are as follows:

  1. In constructors, ensure that the stack has capacity at least 1.
  2. Then `data_[0]` is available for return-by-reference *after sending an error message via `std::cerr`*.
  3. Similarly `T()` (the default `T` object) can be used to return-by-value, again *after sending an error message via `std::cerr`*.

- You can test your display set methods with this sort of code:

```
...
case 'o': case 'O':   // void SetOFC()
  std::cin >> ofc;
  if      (ofc == '0')               ofc = '\0';
  else if (ofc == 'b' || ofc == 'B') ofc = ' ';
  else if (ofc == 't' || ofc == 'T') ofc = '\t';
  else if (ofc == 'n' || ofc == 'N') ofc = '\n';
  else
  {
    std::cout << " ** bad ofc: only 0, b|B, t|T, n|N accepted\n";
    break;
  }
  q.SetOFC(ofc);
  break;
case '<': // SetDIR(bottom-to-top) - horizontal displays, top at right
  q.SetDIR(+1);
  break;
case '>': // SetDIR(top-to-bottom) - vertical displays, top on top
  q.SetDIR(-1);
  break;
...
```

This is helpful because it is hard to enter the typical choices for ofc with the keyboard. Basicly, this maps user input to arguments for SetOFC as follows:

```
kb input --> ofc argument  character name
--------     ------------  --------------
       0     '\0'          null character
       b     ' '           blank
       B     ' '           blank
       t     '\t'          tab
       T     '\t'          tab
       n     '\n'          newline
       N     '\n'          newline
```

Note that these are the only characters that seem useful for output formatting. Any other input is rejected by the sample test code case above.