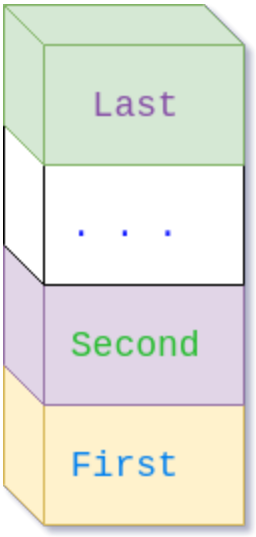
Stack

A stack is a data structure where data is inserted and removed at one end and one end only. This end is known as the **top**. The other end is known at the **bottom**.



A stack can be thought of like a stack of papers in a printer. The first piece of paper that is inserted is the last piece of paper to be used. The last piece of paper inserted is the first piece of paper to be used. This process is known as **LIFO** (**L**ast **I**n **F**irst **O**ut). The pieces of paper are always inserted on top of the existing stack of papers.

There are two terms in regards to insertion and deletion of data when it comes to a stack. **Push** is when you insert data into the stack and **pop** is when you delete data from the stack.

In order to create your own stack data structure, you need to have a few mandatory functions. These functions mimic the functions provided for you in the Standard Template Library stack class. A **push(DATA)** function is needed to insert data into the stack where **DATA** is a value of the type of the stack. A **pop()** function is used to delete data from the stack. A **size()** function returns how many pieces of data are in your stack. An **empty()** function returns a `bool` true if empty and false if data exists in the stack. Lastly, a **top()** function that returns the data that is on the top of your stack.

It is wise to make your stack out of a template so you can have the flexibility to insert any simple base type into the stack. When you make a template stack, only one type is allowed per stack. Remember the type is denoted in angle brackets, such as **Stack<char> myTemplateStack;** One way to start with the creation of your stack is to use a `struct` for your data and implement the stack using a **linked list**. If you further decide to go the Object-Oriented route with a `class`, continue to use a `struct` for the data and implement a pointer to the `struct` as a `private` attribute of the class. For an example without a class implementation, see **Stack.cpp**. For an example with a class implementation, see **TemplateStackClass.cpp**. Both programs are explained via comments.

When using the Standard Template Library version of a stack, you must include the preprocessor directive **#include <stack>**. If you are not **using namespace std;**, you need to either put **using std::stack;** at the top of your program or prepend stack with **std::** as in **std::stack<int> myStack;**

Since a stack's data is inserted and deleted from only one end, there are no iterators to be used.

A stack in the Standard Template Library is implemented using either a **vector**, **deque**, or **list**. The default data structure for a stack is a **deque**. This means you are able to put a deque into a stack. An example of this is as follows:

```
std::deque<int> myDeque(5, 50); //deque with 5 elems with value of 50
std::stack<int> myStack(myDeque); //stack 5 elements with value of 50
```

Now, if you want to put a **vector** or a **list** into a stack, you must explicitly override the default deque implementation as such (keep in mind you must have declared and initialized the vector or list beforehand:

```
std::stack<char, std::vector<char>> myStack(myVector);
std::stack<double, std::list<double>> myStack(myList);
```

To initialize an empty stack, you do the following:

```
std::stack<int> myStack;
```

The following are member functions for the **stack** class. There aren't many due to the simplistic properties of a stack.

To check whether the stack is empty, use the **empty()** member function. This will return true if empty and false if there are some contents in the stack. An example of this is as follows: `if(myStack.empty())`

To see the size of the stack, use the member function **size()**. This will return a **size_t** (like an **int** but cannot be a negative number) that is the number of elements currently in the stack. An example of this is as follows: `cout << myStack.size();`

To access the next element in the stack, use the member function **top()**. This returns a reference to the top element of the stack (the last element that was pushed onto the stack). An example of this is: `cout << myStack.top();` Since this returns a reference, you are also able to change the value of the top element like: `myStack.top() = 4;`

To push data onto the stack, use the member function **push(DATA)**. This will place data onto the top of the stack. An example of this is: `myStack.push('a');` This will insert the **char a** onto a stack of type **char**.

To remove the top element of the stack, use the member function **pop()**. This will delete the element at the top of the stack making the second-to-last inserted element the new top element if it exists. If there are no further elements, the stack is empty. An example is simply: `myStack.pop();`

To find out more information on stacks, please visit **http://www.cplusplus.com/reference/stack/stack/**