COM4013 Introduction to Software Development

- Week 14
- Arrays, Stacks, and Stack Class

- *Umar Arif – u.arif@leedstrinity.ac.uk*

Arrays

- A single variable holds just one value.
- A structure is a way to hold multiple values in a single object. The values can be of different data types.

- Arrays are a technique for declaring many variables of **the same type** in one statement.
- Can use the whole array as a single object.
- Can also use individual elements of the array.

- Declaring:
  - 100 integer variables: `int arr[100];`
  - 3 floating point variables: `float numbers[3];`
  - 10 character variables: `char letters[10];`

- The individual elements of an array are accessed using a subscript (the number inside the square brackets):
  - `arr[0];  // First element`
  - `arr[99]; // Last element`

Arrays

- Arrays can be initialised:

```
int primes[10] = { 1, 2, 3, 5, 7, 11 };
```

- Use a comma separated list, between curly brackets

- Note that 4 elements of the array are uninitilaised
- You do not need to provide the size of your array `[10]` if you intialise the array as above as the compiler will work this out

- You pass an array over to a function using the name of the array only

```
int main( )
{
    int test[SIZE];
    FillArray( test );
}
```

Arrays

```
const int SIZE = 10;

struct SDetails
{
    string name;
    int age;
};

int main()
{
    SDetails manyDetails[SIZE];
    for( int i = 0; i < SIZE; i++)
    {
     cout << manyDetails[i].age << manyDetails[i].name;
    }
}
```

- The same can be done with classes too

Two Dimensional Arrays

- 1 dimensional arrays are like lists
- 2 dimensional arrays are like tables:

- int arr[3][4];

- This creates an array of 12 variables.

```
4 6 1 8
3 5 7 4
1 2 9 5
```

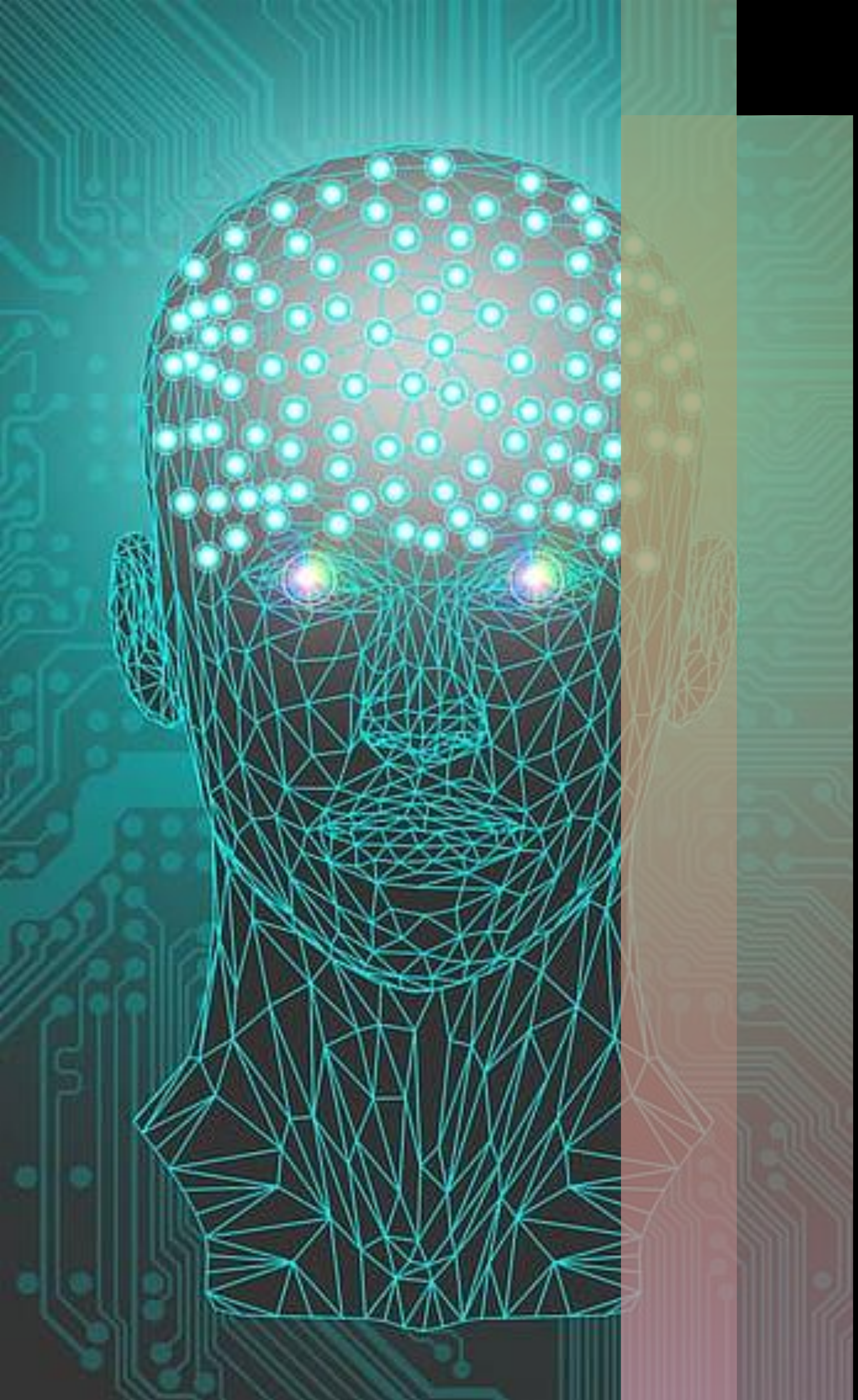
- The array is row by columns and can be initilaised like this:

```
int table[3][5] =
{
   {1, 2, 3, 4, 5},
   {5, 4, 3, 2, 1},
   {1, 3, 5, 3, 1}
};
```

Two Dimensional Arrays

- To access the elements of this array we need a nested for loop.

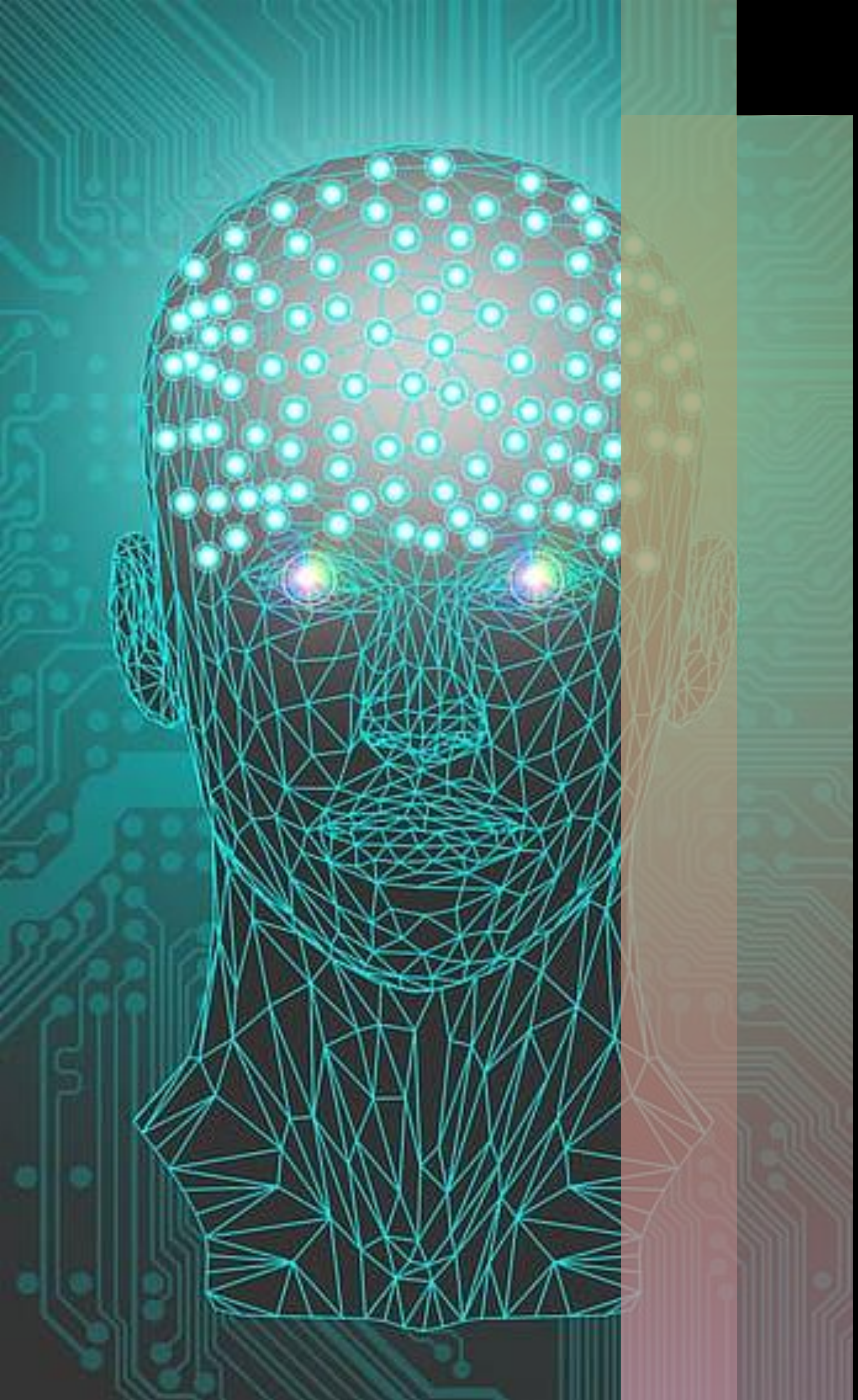
```
const int ROW = 3;
const int COL = 5;

for( int i = 0; i < ROW; i++ )
{
    for( int j = 0; j < COL; j++ )
    {
        cout << table[i][j] << " ";
    }
    cout << endl;
}
```



Stacks

- This is the first of the data structures we are going to examine.
- It follows on from the more trivial classes that we have encountered thus far
  - In stark contrast, a stack is not a "toy" program
  - They are quite important

- Stacks are used at every level of a computer system.
- Several programming languages are based on stack architectures:
  - Java
  - C#
  - Python
  - Pop-11
  - Forth



Real World Uses

- Undo Mechanism in Software
  - Ever wondered how the "Undo" feature in your favorite word processor or graphic design software works?
  - Stacks play a crucial role here
  - Each action you take is **pushed** onto a stack, allowing the software to reverse (**pop**) these actions in the order they were performed
- Web Browser History
  - The back and forward buttons in your web browser navigate through pages you've previously visited
  - This navigation is managed by a stack, where each new webpage you visit is **pushed** onto the stack and pressing "Back" **pops** the last visited page

Normal Function

- In standard program execution, when one function calls another, a sequential process unfolds on the stack.

- For instance:
  - f1 calls f2
  - f1 is placed on the stack
  - Execution shifts to f2
  - f2 calls f3
  - f2 is then added to the stack
  - Execution now proceeds to f3

- In software development, when a function calls another function, the call stack is employed. Each function call is pushed onto the stack, ensuring the program maintains a record of its state.
- When a function completes execution, it is popped off the stack, allowing the program to return to the previous state.

C++ Stacks

- C++ uses stacks to store data local to a function
- As you saw in the last slide, if a function f1 invokes another function f2, the function f1 is placed on the stack and execution passes onto f2. When f2 is finished then f1 is taken off the stack and execution in f1 resumed
- There is a version of a stack in the Standard Template Library (STL) but is still necessary sometimes to program your own

- Stacks are **LIFO containers** (Last In, First Out), meaning that the last element added to the stack is the first to be processed and removed

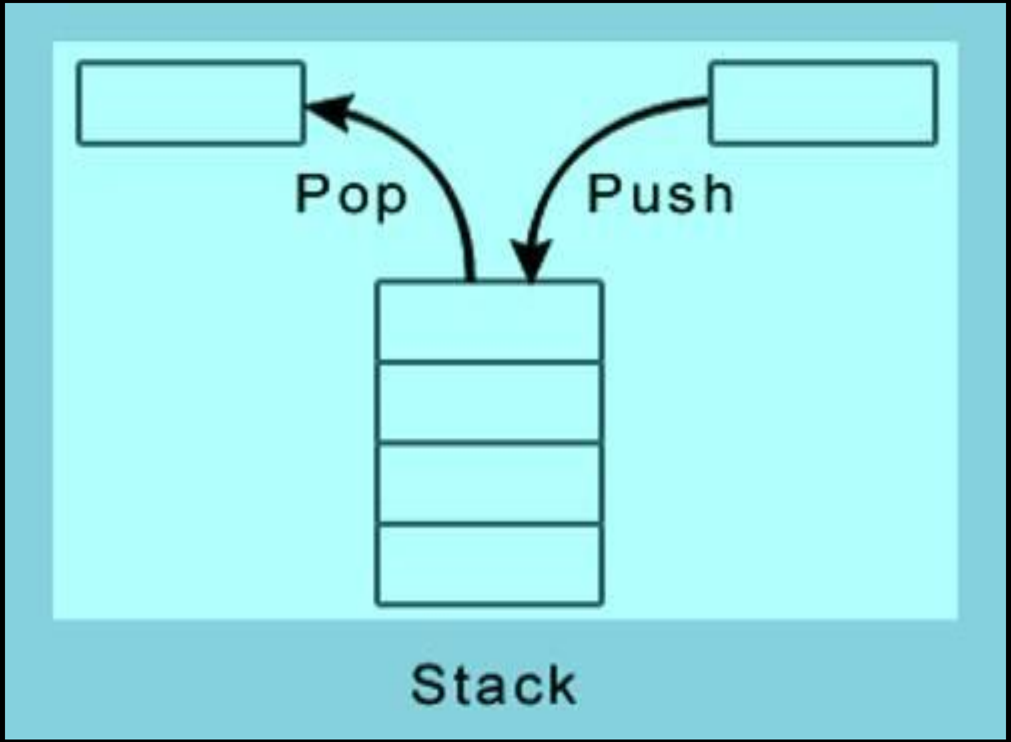
What is a Stack?

- Stacks are a way of storing data
- So, a stack is a "data structure". It is a particular way of organising data
- Stacks allow you add or remove items from the list of data dynamically, i.e. when the program is running
- A stack is a "Last In, First Out "(LIFO) structure
- Other data structures exist, and we may look at some of them later in the module

- A good metaphor for a stack is a stack of plates
- You can add a plate onto the top of the stack
- You can remove a plate from the top of the stack
- You can't remove a plate from the middle or bottom
    - Well, you could, but it wouldn't end well

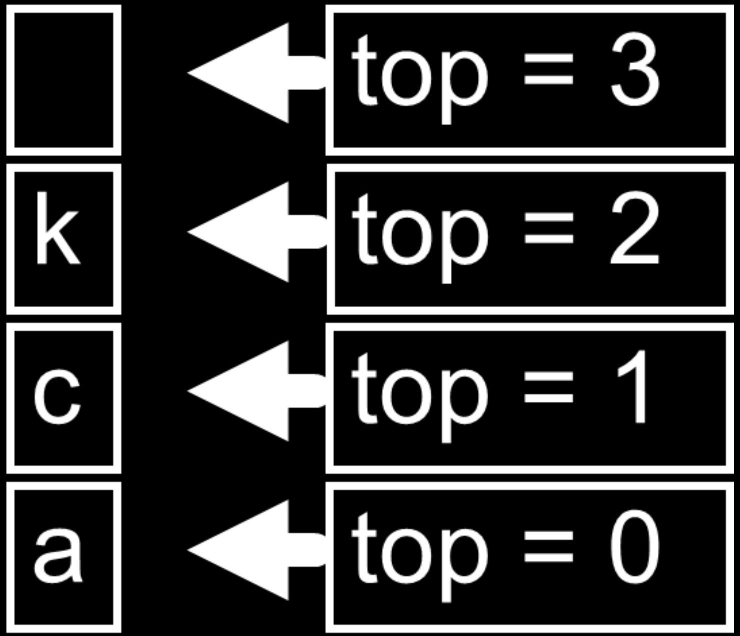
What is a Stack? Data.

- We will use an array to implement a stack. We need an array to store the data. Each element of the array stores one item of data
- We need to keep a record of where the top of the stack is. We will implement this using an integer variable



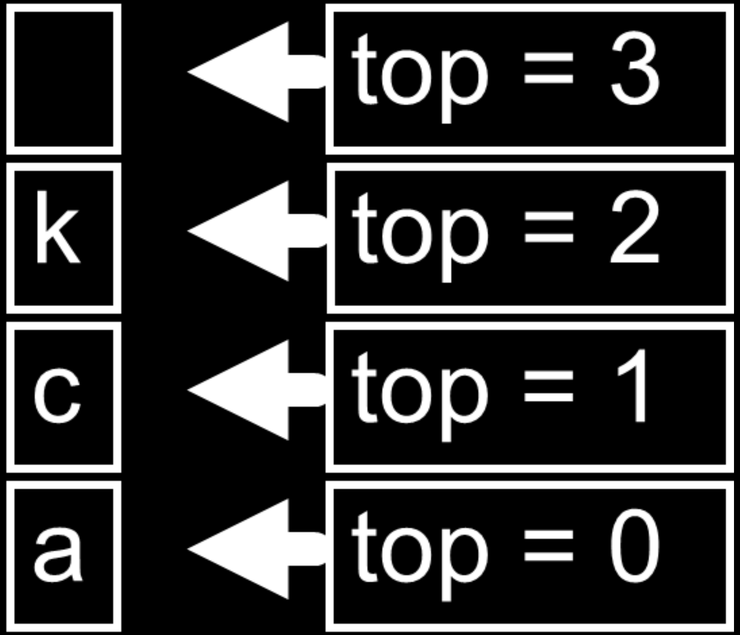
Stacks - Push

- Push 'a' and increment top to 1
- 'a' is placed at the top of the stack (element 0)
- Push 'c' and increment top to 2
- 'c' is added to the top of the stack (element 1)
- Push 'k' and increment top to 3
- 'k' is pushed onto the top of the stack (element 2)
- Now, the stack is at top = 3, indicating the current state after these operations

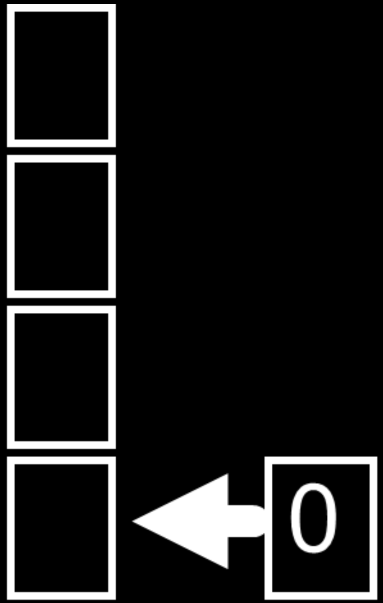


Stacks - Pop

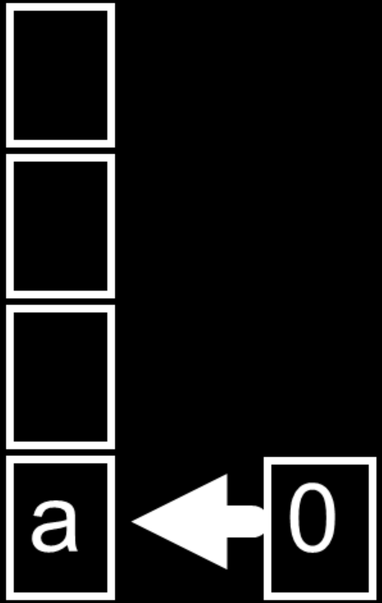
- Pop 'k' and decrement top to 2
- Retrieve and remove 'k' from the top of the stack (element 2).
- Pop 'c' and decrement top to 1
- Retrieve and remove 'c' from the top of the stack (element 1).
- Pop 'a' and decrement top to 0
- Retrieve and remove 'a' from the top of the stack (element 0).
- The operations result in top = 0, indicating an empty stack after the specified pops.



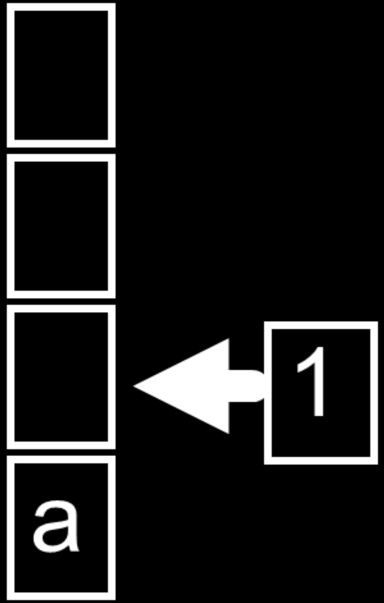
Push (as a static diagram)



- top = 0

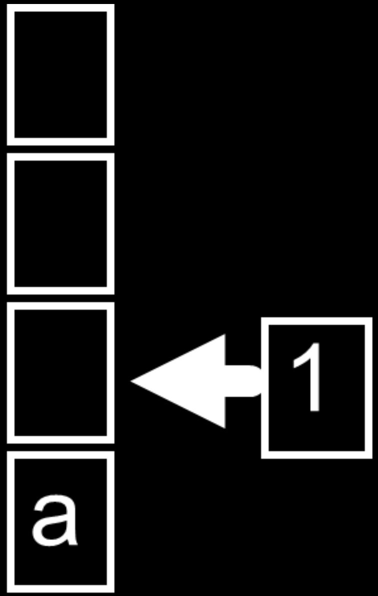


- push 'a' onto the top of the stack

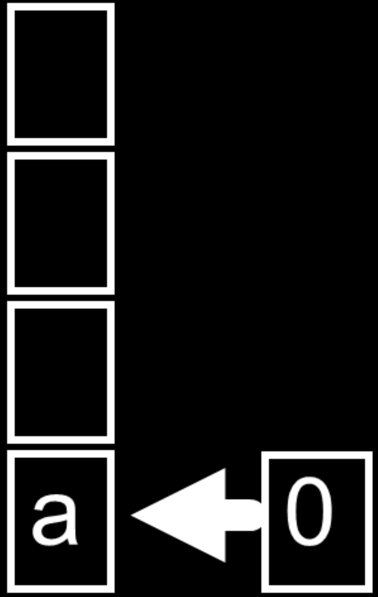


- Increment top of the stack.

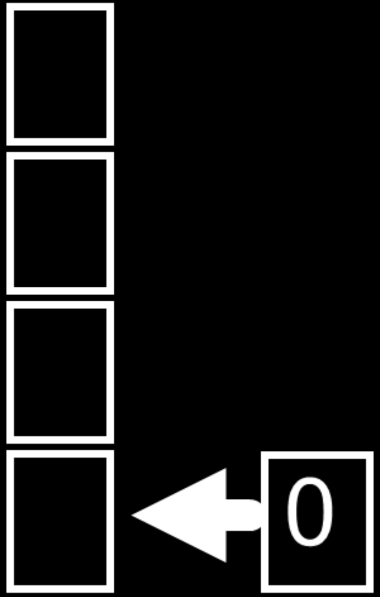
Pop (as a static diagram)



- top = 1



- decrement top of the stack.



- pop 'a' from the top of the stack

Implementing a Stack

- Data:
  - The stack class needs two elements of data: an array and a record of the value of top
  - Only the class needs to see this data
  - We will use functions to facilitate access to the data held on the array
  - This data is **private** to the class
    - Unlike a struct we can hide aspects of the class, so that users are unable to tamper with them

- Operations:
  - We need to implement two operations:
    - Push
    - Pop
  - They will be implemented using member functions (methods).
  - The member functions need to be visible outside the class
  - The member functions are declared as **public**

One extra essential function

- We need to set the stack up in the first place
- Initialisation of the class is quite simple and straightforward (indeed this is typically the case with classes)
- All that needs to be set up at the start is the starting value of the top of the stack
  - The stack top is 0 at the start
- We generally use a **constructor** to do this

- We do not want to go outside the array bounds
- Arrays only have a limited number of space (unlike Python Lists)
- Need to implement two main error checks
- Check that there's room in the array for another data item when we push
- Check that an item exists on the stack before the program attempts to remove it
- Easiest to express this is to check that the stack is not empty when we pop

Class Declaration

```
const int SIZE = 20; // Note the SNAKE_CASE

class CStack // This would be in a .h file
{
    private:
        int top;
        string data[SIZE]; // Stack of words
    public:
        CStack();
        void Push( string newData );
        void Pop( string &oldData );
        bool IsEmpty();
        bool IsFull();
        ~CStack();
};
```

Class methods can be declared either inside or outside the class

Constructor and Destructor

```
// These methods would usually be in the .cpp file
CStack::CStack()
{
    top = 0;
}

CStack::~CStack()
{
    Cout << "Stack object has been destroyed\n";
}
```

- The constructor is pretty simple, as it sets top to 0
- The destructor returns a message when the object is destroyed

Push

```
void CStack::Push( string newData )
{
    if( !IsFull() )
    {
        data[top] = newData;
        top++;
    }
    else
    {
        cout << "stack full" << endl;
    }
}
```

- To declare a method outside the class in C++, specify the return type, followed by the class name, the scope resolution operator (::), and the method name. Include parameters and the method body.

Pop

```
void CStack::Pop( string &oldData )
{
    if( !IsEmpty() )
    {
        top--;
        oldData = data[top];
    }
    else
    {
        cout << "stack empty" << endl;
        oldData = "";
    }
}
```

- The & in string &oldData indicates that oldData is **passed by reference**, allowing the **pop** method to modify its value directly without unnecessary copying.

IsFull and IsEmpty

```
bool CStack::IsFull()
{
    if ( top >= SIZE )
    {
        return true;
    }
    return false;
}

bool CStack::IsEmpty()
{
    if ( top <= 0 )
    {
        return true;
    }
    return false;
}
```

- Returns Boolean value that is used in the other methods

Other Useful Methods

- User defined stack length (not a method but a feature)
- Display the stack
- Get the length of the stack
- Display the stack in reverse order
- Count the number of times a particular data item occurs within the stack
- Search the stack for the first occurrence of a data item
- Find the nth element of a stack
- Copy over the contents of a stack into another stack

DisplayStack

- To use this, fist you must add this to the list of methods in the class itself

```
void CStack::DisplayStack()
{
    for( int i=0; i<top; i++)
    {
        cout << data[i] << endl;
    }
}
```

- Note the way that the for loop is used to go through the whole of the stack
- Note that I'm using top in the for loop
- What other for loop type exists that would also allow me to iterate over my string array?

Main

```
int main()
{
    CStack myStack(); // Declare object directly

    myStack.Push("red");
    myStack.Push("green");

    myStack.DisplayStack();

    string storedData;
    myStack.Pop(storedData);
    cout << "popped: " << storedData; // Green
}
```

Main

```
int main()
{
    CStack * myStack = new CStack(); // Dynamic

    myStack->Push("red");
    myStack->Push("green");
    myStack->DisplayStack();

    string storedData;
    MyStack->Pop(storedData);
    cout << "popped: " << storedData; // Green

    delete ( myStack ); // Destructor comes in play
}
```

- Dynamically allocating memory using the new keyword
- Note that instead of the dot operator we now must use ->
- Also make sure you delete the object to release the memory

Summary

We've covered quite a few topics today:

- Introducing the stack

Any questions?