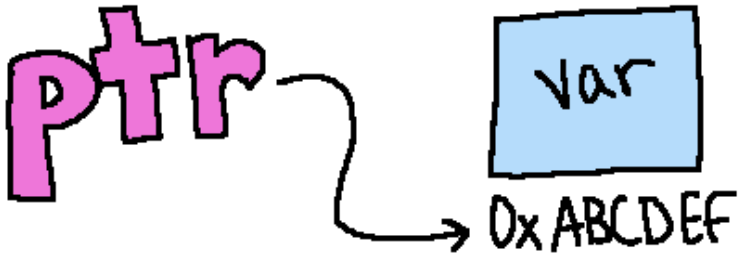
Core Computer Science Notes: Memory, Pointers, & Dynamic Arrays



An overview compiled by Rachel Singh

Last updated July 15, 2020

Contents

Topic 1

Memory Addresses

When we declare a variable, what we're actually doing is telling the computer to set aside some **memory** (in RAM) to hold some information. Depending on what data type we declare, a different amount of memory will need to be reserved for that variable.

Data type	**Size**
boolean	1 byte
character	1 byte
integer	4 bytes
float	4 bytes
double	8 bytes

A **bit** is the smallest unit, storing just 0 or 1. A **byte** is a string of 8 bits. With a byte, we can store numbers from 0 to 255, for an *unsigned* number (only 0 and positive numbers, no negatives).

Minimum value, 0:

128	64	32	16	8	4	2	1
0	0	0	0	0	0	0	0

$$\text{Decimal value} = 128 \cdot 0 + 64 \cdot 0 + 32 \cdot 0 + 8 \cdot 0 + 4 \cdot 0 + 2 \cdot 0 + 1 \cdot 0$$

Maximum value, 255:

128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1

$$\text{Decimal value} = 128 \cdot 1 + 64 \cdot 1 + 32 \cdot 1 + 8 \cdot 1 + 4 \cdot 1 + 2 \cdot 1 + 1 \cdot 1$$

A `char` only needs one byte to store a single letter because we represent letters with the ASCII or UTF8 codes 65 - 90 for upper-case, and 97 - 122 for lower-case.

A	B	C	D	E	F	G	H	I	J	K	L	M
65	66	67	68	69	70	71	72	73	74	75	76	77

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
78	79	80	81	82	83	84	85	86	87	88	89	90

a	b	c	d	e	f	g	h	i	j	k	l	m
97	98	99	100	101	102	103	104	105	106	107	108	109

n	o	p	q	r	s	t	u	v	w	x	y	z
110	111	112	113	114	115	116	117	118	119	120	121	122

Any of these numbers can be stored with 8 bits:

A (65):

128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	1

$$\text{Decimal value} = 128 \cdot 0 + 64 \cdot 1 + 32 \cdot 0 + 8 \cdot 0 + 4 \cdot 0 + 2 \cdot 0 + 1 \cdot 1$$

z (122):

128	64	32	16	8	4	2	1
0	1	1	1	1	0	1	0

$$\text{Decimal value} = 128 \cdot 0 + 64 \cdot 1 + 32 \cdot 1 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 \cdot 0$$

So when we **declare** a variable, the computer finds an available space in memory and reserves the appropriate amount of bytes in memory. For example, our `char` could be assigned the **memory address** `0xabc008` in memory, and its value being stored would look like this:

...007	...008	...009	...00a	...00b	...00c	...00d	...00e	...00f	...010
1	0	1	1	1	1	0	1	0	1

(...007 and ...010 are spaces in memory taken by something else)

We can view the addresses of variables in our program by using the **address-of** operator `&`. Note that we have used the ampersand symbol before to declare pass-by-reference parameters, but this is a different symbol used in a different context.

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int number1 = 10;
7      int number2 = 20;
8
9      cout << &number1 << "\t"
10         << &number2 << endl;
11
12     return 0;
13 }
```

Running the program, it would display the memory addresses for these two variables. Notice that they're 4 bytes apart in memory (one is at ...70 and one is at ...74):

```
0x7ffd3a24cc70  0x7ffd3a24cc74
```

Each time we run the program, we will get different memory addresses, since the operating system reclaims that memory when the program ends, and gives us new memory spaces to allocate next time we run the program.

```
0x7ffe0e708a80  0x7ffe0e708a84
```

When we declare an **array** of integers of size n, the program asks for $n \times 4$ bytes of memory to work with. The two variables above didn't *have* to be side-by-side in memory; they just happened to be because they were declared close together. With an array, however, **all elements of the array** will be **contiguous** (side-by-side) in memory.

Here we have an array of integers:

```
1 int arr[3];
2
3 for ( int i = 0; i < 3; i++ )
4 {
5     cout << &arr[i] << "\t";
6 }
```

And the output:

```
0x7ffd09a130c0  0x7ffd09a130c4  0x7ffd09a130c8
```

Because the elements of an array must be contiguous in memory, we *cannot* resize a normal array. After our array is declared, chances are the memory addresses right after it will be put to use elseware on the computer and will be unavailable to our program and our array.

But, after we learn about pointers, we will learn how we can dynamically allocate as much memory as we need at any time during our program's execution - giving us the ability to "resize" arrays by allocating space for a *new* array, copying the data over to the larger chunk of memory, and deallocating the old chunk of memory from the smaller array.

Topic 2

Pointers

2.1 Creating pointer variables

We can declare special variables that store memory addresses rather than storing an int or float or char value. These variables are called **pointers**.

We can declare a pointer like this: `int* ptrNumber;`
Or like this: `int * ptrNumber;`
Or like this: `int *ptrNumber;`

But note that doing this declares one pointer and several integers:
`int * ptrNumber, notAPointer1, notAPointer2;`
To avoid confusion, declare multiple pointers on separate lines.

If we declare a pointer as an `int*` type, then it will only be able to point to the addresses of **integer variables**, and likewise for any other data type.

Safety with pointers!
Remember how variables in C++ store **garbage** in them initially? The same is true with pointers - it will store a garbage memory address. This can cause problems if we try to work with a pointer while it's storing garbage.

To play it safe, any pointer that is not currently in use should be initialized to `NULL` or `nullptr`.

Declaring a pointer and initializing it to `nullptr`:

```
#include <iostream>
using namespace std;

int main()
{
    int * ptr = nullptr;

    return 0;
}
```

2.2 Assigning pointers to addresses

Once we have a pointer, we can point it to the address of any variable with a matching data type. To do this, we have to use the **address-of** operator to access the variable's address - this is what gets stored as the pointer's value.

Assigning an address during declaration:

```
int * ptr = &somevariable;
```

Assigning an address *after* declaration:

```
ptr = &somevariable;
```

After assigning an address to a pointer, if we `cout` the pointer it will display the memory address of the *pointed-to* variable - just like if we had used `cout` to display the *address-of* that variable.

```
// Shows the same address
cout << ptr << endl;
cout << &somevariable;
```

Here's a simple program that has an integer `var` with a value of 10, and a pointer `ptr` that points to `var`'s address.

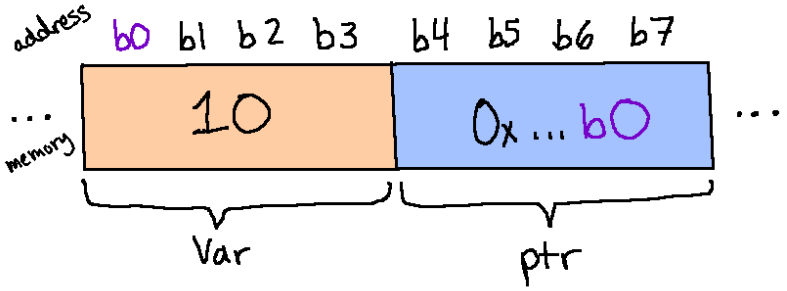
```
#include <iostream>
using namespace std;

int main()
{
    int * ptr;
    int var = 10;

    ptr = &var;

    cout << "var address: " << &var << endl;
    cout << "ptr address: " << &ptr << endl;
    cout << "var value:   " << var << endl;
    cout << "ptr value:   " << ptr << endl;

    return 0;
}
```

The output would look like:

```
var address: 0x7ffc3d6028b0
ptr address: 0x7ffc3d6028b4
var value:   10
ptr value:   0x7ffc3d6028b0
```

Some things to note:

- `var` stores its data at its address `0x7ffc3d6028b0`.
- `ptr` stores its data at its address `0x7ffc3d6028b4`.
- `var`'s value is 10, since it's an integer.
- `ptr`'s value is the address of `var`, since it's a pointer-to-an-integer.



2.3 Dereferencing pointers to get values

Once the pointer is pointing to the address of a variable, we can *access* that pointed-to variable's value by **dereferencing** our pointer. This gives us the ability to read the value stored at that memory address, or overwrite the value stored at that memory address.

We **dereference** the pointer by prefixing the pointer's name with a `*` - again, another symbol being reused but in a different context.

In this code, we point `ptr` to the address of `var`. Outputting `ptr` will give us `var`'s address, and outputting `*ptr` (ptr dereferenced) will give us the value stored at `var`'s address.

```
1 int * ptr;
2 int var = 10;
3
4 ptr = &var;
5
6 cout << "ptr value:         " << ptr << endl;
7 cout << "ptr dereferenced: " << *ptr << endl;
```

Output:

```
ptr value:         0x7ffd21de775c
ptr dereferenced: 10
```

Then, we could also overwrite the value stored at `var`'s address by again dereferencing the pointer and writing an assignment statement:

```
1 *ptr = 20;
```

When we output the value that `var` is storing, either directly through `var` or through the `ptr`, we can see that the value of `var` has been overwritten:

```
1 cout << "var value:         " << var << endl;
2 cout << "*ptr value:        " << *ptr << endl;
```

```
var value:         20
*ptr value:        20
```

2.4 Pointer cheat sheet

Declare a pointer

```
char * ptrChar;
int* ptrInt;
float *ptrFloat;
```

Assign pointer to address

```
char * ptrChar = &charVar;
ptrInt = &intVar;
```

Dereference a pointer
to get the value of the pointed-to variable

```
cout << *ptrChar;
*ptrInt = 100;
```

Assign to nullptr
when the pointer is not currently in use.

```
float *ptrFloat = nullptr;
ptrChar = nullptr;
```

2.5 Invalid memory access with pointers

Remember that when you declare a variable in C++, it will initially store **garbage**. This is true of pointers as well. When you declare a pointer without initializing it to `nullptr`, it will be storing random garbage that it will try to interpret as a memory address. If you **dereference** a pointer that is pointing to garbage, your program is going to run into a problem - a segmentation fault.

A pointer pointing to garbage:

```
1 int main()
2 {
3     int * bob;
4     cout << *bob << endl;
5
6     return 0;
7 }
```

Program output:

```
Segmentation fault (core dumped)
```

How do we check whether memory address is valid? In short, we can't check if an address stored in a pointer is valid. This is why we initialize our pointers to `nullptr` when they're not in use - we know `nullptr` means that the pointer is not pointing to anything.

Topic 3

Dynamically allocating memory

One handy use of pointers is to point at the addresses of other variables that were already declared. However, there is a second main use of pointers: To dynamically allocate memory any time we need new variables that we don't have to pre-write in the program.

3.1 Dynamic variables

At the moment, dynamic variables might seem pointless, but they are really handy for **linked data structures** - a type of structure you can use to store a series of data in that is an alternative to an **array**.

3.1.1 new and delete

We use the **new** keyword to allocate memory for the size of **one** variable data type (as opposed to an array of the variables), and **delete** to free up that memory.

Once we've allocated memory through the pointer, we can use the pointer as normal. The only difference is that we're now accessing an address that was allocated differently from the address of a "normal" variable.

Allocating memory for a single item: `int * ptr = new int;`

Deallocating memory for a single item: `delete ptr;`

Small program using a dynamic variable:

```
1  int main()
2  {
3      int * myNumber;              // Declare pointer
4
5      myNumber = new int;          // Allocate memory
6
7      *myNumber = 10;              // Assign value
8
9      cout << *myNumber << endl;   // Print value
10
11     delete myNumber;             // Free memory
12
13     return 0;
14 }
```

3.2 Dynamic arrays

For our normal arrays, we are unable to resize them during the program's run time because an array requires that all **elements** of the array are **contiguous** (side-by-side) in memory.

However, if we can allocate and deallocate memory any time we want with pointers, we can create a **dynamic array**. If we ever need to resize it, we allocate a new array of a bigger size and copy the data from the old array to the new one.

3.2.1 new[ ] and delete[ ]

When we're allocating space for an array of items, we use a slightly different set of keywords:

Allocate memory for an array: `int * arr = new int[100];`

Deallocate memory for an array: `delete [] arr;`

Example program that lets the user specify how big the array will be.

```
1  int main()
2  {
3      int arraySize;
4      float * pricesArray;
5
6      cout << "How many items? ";
7      cin >> arraySize;
8
9      pricesArray = new float[arraySize]; // Allocate mem
10
11     for ( int i = 0; i < arraySize; i++ )
12     {
13         cout << "Enter price for item " << i << ": ";
14         cin >> pricesArray[i];
15     }
16
17     delete [] pricesArray;          // Deallocate memory
18
19     return 0;
20 }
```

When you've used a pointer to allocate space for an array, you will access items in the dynamic array the same way you would do for a normal array. In other words, you would use the subscript operator [ ] to access elements at particular indices, and you don't need to prefix the pointer with the dereference operator *.

3.2.2 "resizing" a dynamic array

Let's say we're writing a program that stores the user's library of movies they own. It saves a file so they can run the program any time, add or delete movies, close the program and open it again sometime later.

We can't realistically predict how many movies a person would own - some people might be collectors and have tons of movies. At the same time, if we were to over-estimate and make an array with, say, 10,000 elements, it would be a *huge* waste of memory if we had such a big array for someone who only had a few dozen movies.

A dynamic array would work perfectly here because we can initially create an array that's fairly small - maybe 10 items - and if the array fills up, we can then allocate more memory and copy the movies to the new array and

deallocating the old array. If it fills up again, we just repeat the process, allocating more memory, copying the data over, and freeing the old array's memory.

Resizing steps

The resizing process works like this:

1. Create a pointer and use it to allocate a new, bigger array.
2. Use a for loop to copy all the elements from the old-small-array to the new-big-array.
3. Free the memory for the old-small-array.
4. Update the array's pointer, which *was* pointing to old-small-array, to now point to the address of the new-big-array.

Normally, you would store a dynamic array inside of a class that acts as the *interface* for adding and removing data, where it will check if there's enough space before adding new information and resizing the array if not.

Here's an example of resizing a movie array within a MovieLibrary class:

```
1  void MovieLibrary::Resize()
2  {
3      // Allocate more memory
4      int newSize = m_arraySize + 10;
5      string * newArray = new string[newSize];
6
7      // Copy data from old array to new array
8      for ( int i = 0; i < m_arraySize; i++ )
9      {
10         newArray[i] = m_movieArray[i];
11     }
12
13     // Free memory of old array
14     delete [] m_movieArray;
15
16     // Update the pointer to point at new array
17     m_movieArray = newArray;
18
19     // Update the array size
20     m_arraySize = newSize;
21 }
```

3.2.3 Dynamic array example: Movie Library

Here's a small example of a class that contains a dynamic array, that allows users to add additional information, and handles checking for a full array and resizing.

MovieLibrary.hpp:

```
#ifndef _MOVIE_LIBRARY_HPP
#define _MOVIE_LIBRARY_HPP

#include <string>
using namespace std;

class MovieLibrary
{
    public:
    MovieLibrary();
    ~MovieLibrary();

    void ViewAllMovies() const;
    void ClearAllMovies();
    void AddMovie( string newTitle );

    private:
    bool IsFull();
    void Resize();

    string * m_movieArray;
    int m_arraySize;
    int m_itemsStored;
};

#endif
```

The methods **IsFull()** and **Resize()** are set as private because outside users *don't* need to know anything about *how the data is stored*. The outside users shouldn't be able to recall Resize whenever they want.

MovieLibrary.cpp:

Constructor:

```
1 MovieLibrary::MovieLibrary()
2 {
3     m_arraySize = 10;
4     m_movieArray = new string[m_arraySize];
5     m_itemsStored = 0;
6 }
```

In the constructor, we initialize our `m_movieArray` member pointer variable by allocating some memory to start with.

If we *weren't* going to allocate memory in the constructor, then we should initialize `m_movieArray` by setting it to `nullptr`.

We also have two separate size variables - `m_arraySize` keeps track of how many spaces are available in the array, and `m_itemsStored` keeps track of how many items the user has added to the array.

Destructor:

```
1 MovieLibrary::~MovieLibrary()
2 {
3     if ( m_movieArray != nullptr )
4     {
5         delete [] m_movieArray;
6     }
7 }
```

Before the MovieLibrary item is destroyed (e.g., when the program ends and it goes out of scope), we need to make sure to **free the memory that we allocated**. First, we need to check if the pointer `m_movieArray` is pointing to `nullptr` - if it is, we can assume it's not in use and there's nothing to do. But, if `m_movieArray` is pointing to some address, we assume that memory has been allocated here. In this case, we free that memory.

ViewAllMovies:

```
1 void MovieLibrary::ViewAllMovies() const
2 {
3     for ( int i = 0; i < m_itemsStored; i++ )
4     {
5         cout << i << ". " << m_movieArray[i] << endl;
6     }
7 }
```

This is just a simple for loop that goes from index 0 to `m_itemsStored` - 1, displaying each element to the screen.

We aren't iterating until `i < m_arraySize` because if our array size is 10 and the user has only stored 5 movies, we don't need to display 5 empty slots.

ClearAllMovies:

```
1 void MovieLibrary::ClearAllMovies()
2 {
3     delete [] m_movieArray;
4     m_movieArray = nullptr;
5     m_arraySize = 0;
6     m_itemsStored = 0;
7 }
```

If the user decides to clear out their movie array, we could just deallocate all memory reserved for the movie list and reset our size variables. We just need to make sure we're checking to make sure the array is allocated *before* we try to access elements from it or add to it.

IsFull:

```
1 bool MovieLibrary::IsFull()
2 {
3     return m_itemsStored == m_arraySize;
4 }
```

The array is full if the amount of items the user has stored, `m_itemsStored` is equal to the amount of spaces we have allocated for the array, `m_arraySize`.

AddMovie:

```
1  void MovieLibrary::AddMovie( string newTitle )
2  {
3      if ( m_movieArray == nullptr )
4      {
5          m_arraySize = 10;
6          m_movieArray = new string[m_arraySize];
7      }
8      if ( IsFull() )
9      {
10         Resize();
11     }
12
13     m_movieArray[ m_itemsStored ] = newTitle;
14     m_itemsStored++;
15 }
```

Before we add a movie, we need to make sure `m_movieArray` is set up and ready.

First, we check to see if it is pointing to `nullptr`. If it is, we allocate memory.

Next, we check to see if the array is full. If it is, we call Resize().

Finally, once those two checks have been done and the array has been prepared, we can add a new item to the array and increase hte `m_itemsStored` count.

Resize:

```
1  void MovieLibrary::Resize()
2  {
3      int newSize = m_arraySize + 10;
4      string * newArray = new string[newSize];
5
6      for ( int i = 0; i < m_arraySize; i++ )
7      {
8          newArray[i] = m_movieArray[i];
9      }
10
11     delete [] m_movieArray;
12
13     m_movieArray = newArray;
14     m_arraySize = newSize;
15 }
```

This is the same Resize() method as was shown before as an example of resizing a dynamic array.

3.3 STL Vectors and Lists

3.3.1 Vector - A dynamic array

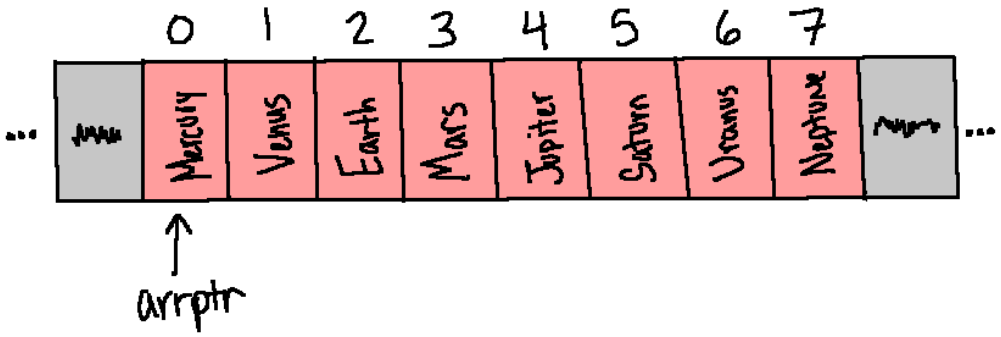
The `vector<>` class that is part of the C++ Standard Template Library is a structure we can use to store data without having to worry about memory allocation or resizing or anything. Behind the scenes, a vector is a **dynamic array**.

Some of the functions it has is:

`push_back( value )`	Adds a new item to the end.
`operator[]`	Access elements of the vector like an array; `cout << myVec[i];`
`size()`	Returns the total amount of elements stored in the vector.
`clear()`	Erases all items in the vector.

Pros and Cons: The good thing about working with arrays is **random access:** we can access any arbitrary element of the array at any position... Item 0? Item 5? Item n-5? All of those are accessible.

A downside to dynamic arrays is the memory allocation part - we often have to allocate a *bit* more memory than we immediately need, and whenever we need to resize the array, the program has to stop, allocate more memory, and copy all the data. This could take a while if we had *a lot of data*.



3.3.2 List - A linked structure

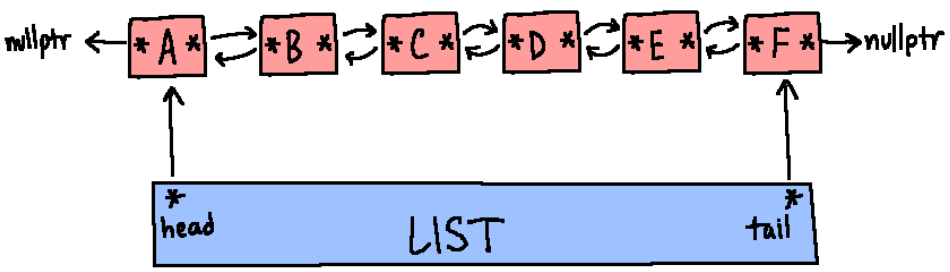
A `list<>` class from the C++ Standard Template Library is another type of structure that also stores its data in a linear order, however, how it's implemented is different from a vector: An STL List is implemented as a **doubly-linked list** - a linking structure that uses pointers in a way so that **only the memory needed is the memory allocated**.

Some of the functions it has is:

`push_back(value)`	Adds a new item to the end.
`push_front(value)`	Adds a new item to the beginning.
`back()`	Access the *last* item of the list.
`front()`	Access the *first* item of the list.
`size()`	Returns the total amount of elements stored in the vector.
`clear()`	Erases all items in the vector.

Pros and Cons: With a linked structure, it is more memory efficient than an array - we only allocate one element's worth of data at a time. Each element of the list also has pointers to the previous and the next item in the list (for a doubly-linked list structure).

On the downside, we **cannot randomly access** items in the list. In order to get item number 3, we would have to start at the **head item**, go to its **next** item, and its **next** item, and its **next** item, and its **next** item - we would have to *walk* through the list one item at a time.



Topic 4

Memory Management

4.1 Types of memory errors

Working with pointers and dealing with memory can lead to writing code that can mess up our program or even slow down our computer (until the next reboot).

Invalid memory access: Invalid memory access happens when a pointer isn't set to `nullptr` when it's no longer in use. If this doesn't happen, the memory address it's pointing to could be invalid, such as if **delete** was used on the pointer, or if the pointer were declared and not initialized.

An invalid memory access would cause a problem once you try to **dereference** the pointer pointing to an invalid address - causing a segfault and your program to crash.

Memory leak: A memory leak occurs when you've allocated memory (via the **new** command) but you never *deallocate it*. C++ won't automatically deallocate this memory for you, and what happens is that chunk of memory ends up being "called for" even after the program has finished running. This memory block will be unavailable to all programs until the user restarts their computer.

If you had many items allocating memory but never freeing them, the resulting memory leaks could cause your computer to slow down over time (until the next reboot).

Missing allocation: Missing allocation occurs if you try to **delete** memory that has already previously been freed. If this happens, the progrma will crash.

4.2 The Stack and the Heap

What *is* the difference between a normal variable...

```
int myNum;
```

... and a dynamically allocated variable?...

```
int * myNum = new int;
```

Well, there are different *types* of memory, and each is stored in a different space.

4.2.1 The Stack

All of our non-dynamic variables we have been declaring, including function parameters and class member variables, get allocated to **the Stack**. The Stack part of memory has a **fixed size**, and it contains a sequence of memory addresses. The Stack automatically handles the memory management for these variables - when we declare a `int count;` variable, it will be pushed onto the Stack, and when it's not in use anymore (when it goes out of scope), it will be removed from (popped off) the Stack.

A "stack overflow" error occurs when so many variables have been declared that the Stack runs out of room. This usually occurs while writing recursive functions that have a logic error, causing them to repeat until we run out of Stack space.

4.2.2 The Heap

The heap is where dynamically allocated data gets stored. The Heap *does not* automatically take care of memory management for us, so that is why we have to deal with **new** and **delete** ourselves (unless we're using a **smart pointer**). Any data in the Heap must be accessed via a pointer. There is no size restriction for the Heap.