## Stack v Heap

- We have seen how to declare basic type variables such as `int`, `double`, etc, and complex types such as arrays and `struct`.
- The standard way of declaring these variables, e.g. `int counter = 1;` puts these variables on the stack.
- To declare variables on the heap in C++ use `new`, e.g. `int *pCounter = new int(1);`

## The Stack

- What is the stack? It's a special region of your computer's memory that stores temporary variables created by each function (including the `main()` function). The stack is a "LIFO" (last in, first out) data structure, that is managed and optimized by the CPU quite closely.
- Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function, are freed (deleted).
- Once a stack variable is freed, that region of memory is available for other stack variables.
- The advantage of using the stack to store variables, is that memory is managed for you. You don't have to allocate memory “dynamically” by hand (using `new` in C++), or free it once you don't need it any more (using `delete` in C++).
- What's more, because the CPU organizes stack memory so efficiently, reading from and writing to stack variables is *very fast*.
- A key to understanding the stack is the notion that when a function exits, all of its variables are popped off of the stack (and hence lost forever).
- Stack variables are local in nature, and thus are related to variable scope.
- There is there is a limit (varies with OS) on the size of variables that can be stored on the stack. This is not the case for variables on the heap.
- If the stack runs out of memory, then this is called a stack overflow error.
- The stack grows and shrinks as functions push and pop local variables. There is no need to manage the memory yourself, variables are allocated and freed automatically.
- Stack variables only exist while the function that created them is running.

## The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- It is a more free-floating region of memory, and is larger. To allocate memory on the heap, you must use `new`. Once you have allocated memory on the heap, you are responsible for using `delete` to deallocate that memory once you don't need it any more.
- Worth mentioning that sequential memory requests may not result in sequential memory addresses being allocated.
- If you fail to `delete` your memory, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- Unlike the stack, the heap does not have size restrictions on variable size, apart from the obvious physical limitations of your computer. Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap.
- Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

## Stack vs Heap Pros and Cons

Stack

- Very fast access.
- Don't have to explicitly de-allocate variables.
- Space is managed efficiently by CPU, memory will not become fragmented.
- Use for local variables only.
- Limit on stack size (OS-dependent).
- Variables cannot be resized.

Once a function call runs to completion, any data on the stack created specifically for that function call will automatically be deleted.

Heap

- Variables can be accessed globally.
- No limit on memory size.
- Relatively slower access.
- No guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed.
- You must manage memory. You're in charge of allocating and freeing variables.
- Variables can be "resized" using `std::copy().` In C++ `copy()` should be preferred to the C calls `realloc` and `memcpy` as neither are safe for arrays of C++ objects.

Any data on the heap will remain there until it's manually deleted by the programmer.

## When to use the Heap?

- When should you use the heap, and when should you use the stack?
- If you need to allocate a large block of memory (e.g. a large array, or a big struct), and you need access to it from multiple functions, then you should allocate it on the heap.
- If you are dealing with relatively small variables that only need to persist as long as the function using them is alive, then you should use the stack, it's easier and faster.
- If you need variables like arrays and structs that can change size dynamically (e.g. arrays that can grow or shrink as needed) then you will likely need to allocate them on the heap, and use dynamic memory allocation functions like copy() to manage that memory "by hand."