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 (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes 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 by hand, or free it once you don't need it any more. 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). Thus stack variables are **local** in nature. This is related to a concept we saw earlier known as **variable scope**, or local vs global variables. A common bug in C programming is attempting to access a variable that was created on the stack inside some function, from a place in your program outside of that function (i.e. after that function has exited).

Another feature of the stack to keep in mind, is that 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 allocated on the **heap**.

To summarize the stack:

- The stack grows and shrinks as functions push and pop local variables
- No need to manage memory yourself, variables are allocated & freed automatically
- the stack has size limits
- 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 `malloc()` or `calloc()`, which are built-in C functions. Once you have allocated memory on the heap, you are responsible for using `free()` to deallocate that memory once you don't need it any more. If you fail to do this, 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). As we will see in the debugging section, there is a tool called `valgrind` that can help you detect memory leaks.

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. We will talk about pointers shortly.

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
- local variables only
- limit on stack size (OS-dependent)
- variables cannot be resized

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 `realloc()`

Examples

Here is a short program that creates its variables on the **stack**. It looks like the other programs we have seen so far.

```
#include <stdio.h>

double multiplyByTwo (double input) {
  double twice = input * 2.0;
  return twice;
}

int main (int argc, char *argv[])
{
  int age = 30;
  double salary = 12345.67;
  double myList[3] = {1.2, 2.3, 3.4};

  printf("double your salary is %.3f\n", multiplyByTwo(salary));

  return 0;
}
```

```
double your salary is 24691.340
```

On lines 10, 11 and 12 we declare variables: an `int`, a `double`, and an array of three doubles. These three variables are pushed onto the stack as soon as the `main()` function allocates them. When the `main()` function exits (and the program stops) these variables are popped off of the stack. Similarly, in the function `multiplyByTwo()`, the `twice` variable, which is a `double`, is pushed onto the stack as soon as the `multiplyByTwo()` function allocates it. As soon as the `multiplyByTwo()` function exits, the `twice` variable is popped off of the stack, and is gone forever.

As a side note, there is a way to tell C to keep a stack variable around, even after its creator function exits, and that is to use the `static` keyword when declaring the variable. A variable declared with the `static` keyword thus becomes something like a global variable, but one that is only visible inside the function that created it. It's a strange construction, one that you probably won't need except under very specific circumstances.

Here is another version of this program that allocates all of its variables on the **heap** instead of the stack:

```
#include <stdio.h>
#include <stdlib.h>

double *multiplyByTwo (double *input) {
  double *twice = malloc(sizeof(double));
  *twice = *input * 2.0;
  return twice;
}
```

```
int main (int argc, char *argv[])
{
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n", *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);

  return 0;
}
```

As you can see, using `malloc()` to allocate memory on the heap and then using `free()` to deallocate it, is no big deal, but is a bit cumbersome. The other thing to notice is that there are a bunch of star symbols * all over the place now. What are those? The answer is, they are **pointers**. The `malloc()` (and `calloc()` and `free()`) functions deal with **pointers** not actual values. We will talk more about pointers shortly. The bottom line though: pointers are a special data type in C that store **addresses in memory** instead of storing actual values. Thus on line 5 above, the `twice` variable is not a double, but is a **pointer to a double**. It's an address in memory where the `double` is stored.

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 to keep that variable around a long time (like a global), 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 `malloc()`, `calloc()`, `realloc()` and `free()` to manage that memory "by hand". We will talk about dynamically allocated data structures after we talk about pointers.

nullptr

The C++ keyword nullptr is a pointer that points to the memory address 0x0.

- nullptr represents a pointer to “nowhere”
- Address 0x0 is reserved and never used by the system.
- Address 0x0 will always generate an “segmentation fault” when accessed.
- Calls to delete 0x0 are ignored.