When a program runs it has a view of RAM memory. For C programs this memory is broken up into four segments: stack, data, heap, and code.

The stack segment is the part of memory that handles function calls. Each time a function is called a new frame (of reference) is created and put onto the top of the stack. Then when the function exits it's frame is popped off of the stack and execution returns to the previous function.

A frame contains the parameters and local variables defined by the function.

When the function exits and the frame is popped off of the stack the memory containing the frame is released. Releasing the memory does not clear the memory released, any data in released memory remains until it is written over. This means that another program may be allocated this memory, and thereby have access to your data.

Since all data in a frame is released when the function call returns there are only two ways to communicate data back out from the function call.

The first is to use the return keyword. This will return the data to the function that called the function that is returning. For example:

```
int main(int argc, char * argv[]){   //creates a frame with two variables, argc, and argv
        int a = returnFive();     // adds the a variable to the frame with the value returned by returnFive()
}
int returnFive(){   //creates a frame with no variables
        return 5; //returns the value 5
}
```

The second is to pass an address (that is not in the frame) into a function. Then that function can write data to this address (that is not on the frame). Since the data exists outside of the frame of the function being called the data is not affected by the release of the frame. For example:

```
int main(int argc, char * argv[]){
        int a;  //adds a to the frame
        setFive(&a);   //passes the address of a into setFive
}
void setFive(int * addr){  //creates  a frame with the variable addr
        *addr=5;  //writes 5 to the address passed in
}  //frame exits, which releases the variable addr
```

If we instead try to pass back the address of a local variable we can run into trouble since that address may be reallocated (and therefore the data may be written over).

The data segment contains all variables which exist outside of the stack. There are three types of variables with the property: static variables, global variables, and extern variables.

A static variable is a variable that is declared static. The static keyword tells C to place this variable in the data segment instead of the stack segment. This means that variables that are declared static retain their values between function calls. For example:

```
int returnValue(){
        static int a=0;
        return a++;
}
```

This function will increment a each time. Since a exists in the data segment it is not released when the function returns, which allows it to keep its value between function calls.

A global variable is a variable that is declared outside of all function definitions. Since there is no frame to associate it with it is placed in the data segment. For example:

```
int myGlobal=5;             //outside of all functions
int main(int argc, char * argv[]){
        int myLocal=5;   //inside of a function
}
```

An extern variable is a variable that is defined in a different file that is compiled with the current file. Extern variables are also global variables. For example:

File1:

```
int myExternVariable=5;
```

File2:

```
int main(int argc, char * argv[]){
        extern int myExternVariable;
        printf("%i",myExternVariable);
}
```

The heap segment is essentially an extra store of data that you can request access to at any time. Inside of the stdlib.h header there is a function named malloc. Malloc takes a number of bytes and returns an address to the beginning of that many bytes of consecutive memory on the heap. Since this memory is on the heap it is not affected by frames being released. For example:

```
int main(int argc, char * argv[]){
        int * myIntAddr = otherFunc();   //creates a variable in the frame that addresses heap memory.
}
int * otherFunc(){
        int * myIntAddr = malloc(sizeof(int));      //creates a variable named myIntAddr
        return myIntAddr;
} //while myIntAddr (the variable) is released, the memory that it points to is NOT released
```

stdlib.h also defines a function called free. This function tells that OS that memory that was previously allocated on the heap is no longer necessary. If this function is not called then the heap memory will remain allocated until the program exits. For example:

```
int main(int argc, char * argv[]){
        int * myIntAddr = malloc(sizeof(int));
        free(myIntAddr);          //manually releases the memory addressed by myIntAddr
        printf("%p",myIntAddr);          //myIntAddr still exists in the frame and may be reused
}       //now myIntAddr is released as well.
```

With all of these pointers everywhere let's take a moment to discuss segmentation faults. A segmentation fault is (roughly) when you violate memory permissions. For example, you will get a segmentation fault if you try to access an address that is "owned" by another process. Whenever this happens it means that you have a pointer that does not point where you think it does.

From a practical point of view segmentation faults occur in one of three ways:

1) You try to access a command line argument that was not provided
2) You try to read from a file that does not exist
3) You make a mistake doing pointer math.

Point math mistakes are by far the most common, and the most difficult to debug. Whenever you encounter a segmentation fault during pointer math it is best to print out the addresses without printing out their values. If you notice that the math is not working out like you expect it is likely that your pointer is the wrong type. For example, you might have an int* instead of a char *, in which case your math will be off by a factor of 4.

The code segment contains the compiled version of the program that is running. Because of this we can get addresses to functions and then call them from their addresses. A variable that contains the address of a function is called a function pointer. Here is an example:

```
int add(int a, int b){return a+b;} //declares the add function

int main(int argc, char * argv[]){
        int (*func)(int,int)=add; //declares the function pointer 'func' and sets it to the address of add
        printf("%i\n",func(5,6)); //calls the function that is addressed by func. (in this case add).
}
```

Let's break down the syntax for declaring a function pointer. The easiest way to explain this is to transform a function definition into a function pointer. For example:

```
int add(int a, int b){return a+b;} //step 1, drop the body
int add(int a, int b);             //step 2, change the name of the function to the name of the variable.
int func(int a, int b);            //step 3, wrap the name of the variable in (* ).
int (*func)(int a, int b);         //step 4, drop the names of the arguments.
int (*func)(int,int);
```

You may notice that this syntax is similar to the syntax to declare a pointer with a specific size.
`int (*twoIntsAddr)[2]`
The only difference is that instead of [2] (the size) we have a list of argument types in parenthesis.

References

Functions

Included in stdlib.h

void * malloc(size_t bytes)

> Allocates the requested number of bytes of memory. Often used with sizeof.
> Returns the address of the first byte.

void free(void * addr)

> Allows the passed address (and all other addresses allocated at the same time) to be reused.
> Always called after malloc.