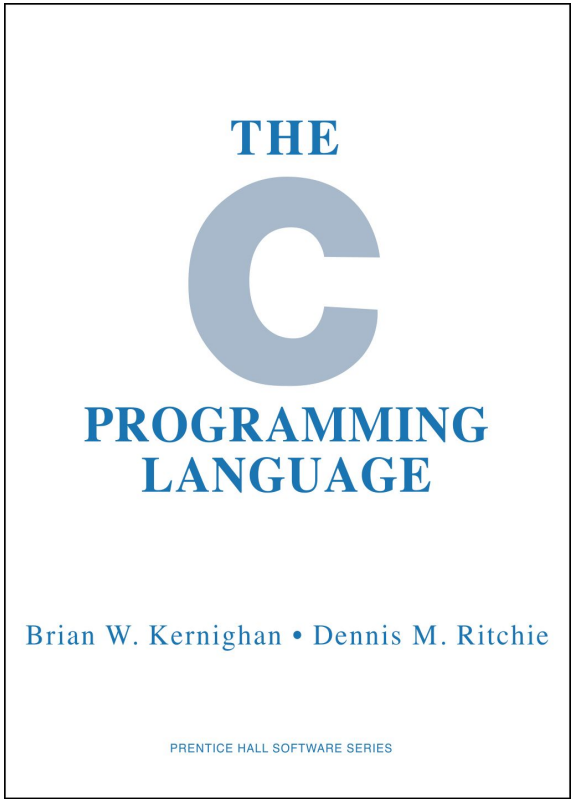

MONTANA
STATE UNIVERSITY

The Stack & The Heap

How Data is Stored in C Programs

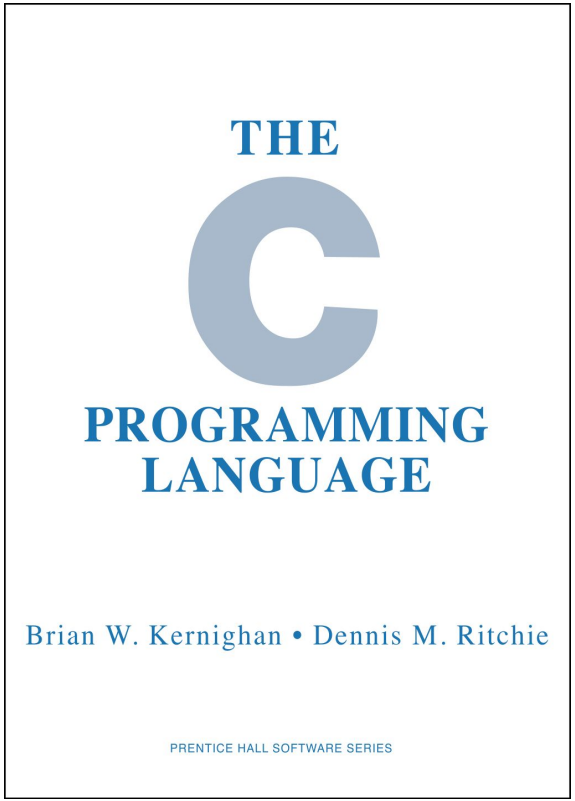
Memory Segments In C

- C has different *segments* of memory
  - Code segment (aka Text Segment) - the instructions of the program are stored here, typically read only!
  - Data & BSS segments - global and static variables are stored here
    - Known, fixed size
    - BSS is zero initialized



Memory Segments In C

- C has different *segments* of memory
  - The Stack - where function parameters, local variables and other function related information is stored
  - The Heap - an area where dynamically allocated memory lives



The Stack

- The stack is where function-related information is store
  - Parameters
    - Kind of - See x64 calling conventions
  - Local variables for a function
  - Return values from functions
  - Return address to jump to when function returns (hidden from us, to an extent)

The Stack

- Recall a stack data structure
- The operations on a stack are
  - Add to the top of the stack
  - Remove from the top of the stack
  - Look At the current top of the stack

The Stack

- You can’t access a plate that isn’t at the top of the stack
- A C function can’t access variables not declared in the current *Stack Frame*
  - A debugger can though, by using data stored in the stack :)

The Stack

- In C, the stack is managed by the C runtime
- When you make a function call, a few things happen:
  - The address of the next instruction is pushed onto the stack
  - A *Stack Frame* is allocated for the new function
  - Parameters are passed appropriately

The Stack

- The CPU then jumps to the function being called
- A stack frame has been *pushed* onto the stack
- When the function completes, the stack frame is deallocated and the CPU jumps back to the return location

The Stack

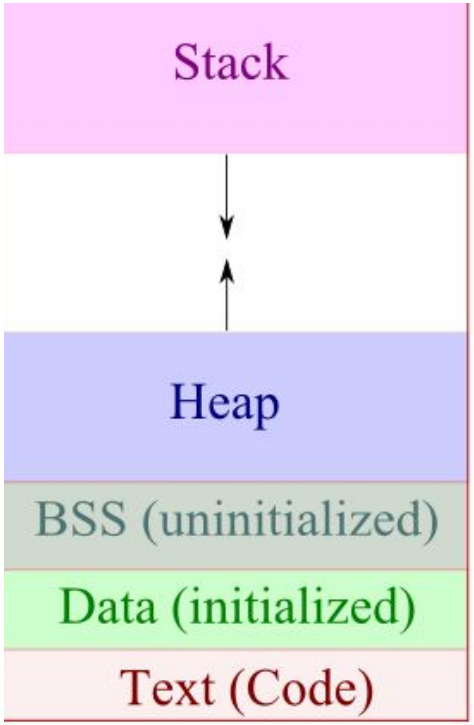
- This is how come *automatic* variables are *automatically* reclaimed: the data lives on the stack and go away when the stack frame goes away

The Stack

- A recursive function demo makes some things clear
  - Notice that each function call has its own copies of the parameter i and the local variable i_address
  - What do you notice about i_address?

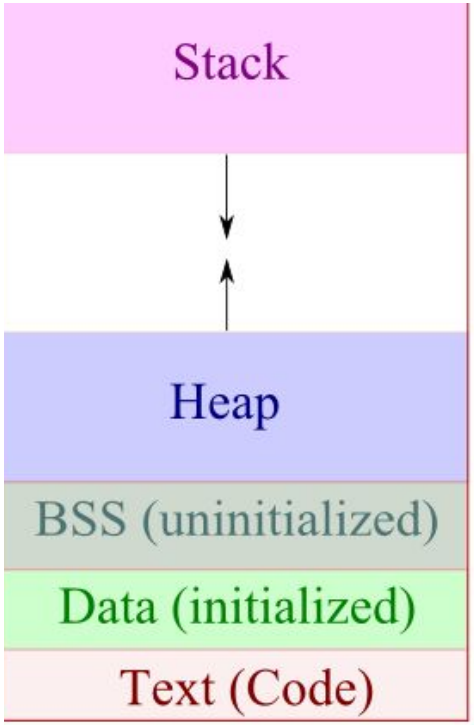
The Stack

- The addresses are going *down* in value
- This is because on x86, the stack starts at a high number and grows *downward*
- The debugger in CLion shows the more intuitive upward layout



The Stack

- However, you should be aware that for historical reasons, the downward growth of the stack is standard
  - See for example the downwards funarg and upwards funarg problems



The Stack

- There is only so much space that the stack is allowed to take up
- Eventually you will run out of space and get a *Stack Overflow*
- In C, this is helpfully implemented as a *SIGSEGV* error

The Heap

- The Stack is very structured
- The Heap is the opposite of that...

The Heap

- The Heap is a collection of stuff
- You can take a big handful of stuff out whenever you'd like
- But you had better put that stuff back when you are done with it
- And you better not use the same stuff in two places

The Heap

- In C, working with the heap means using two main functions:
  - malloc()
  - free()
- Note that these functions are part of <stdlib.h> they are *not* part of the language!

Malloc

- In C, working with the heap means using two main functions:
  - malloc()
  - free()
- Note that these functions are part of <stdlib.h> they are *not* part of the language!

Malloc

- Here is an example using malloc
- We have a pointer to a string in the data segment
- We allocate a 15 byte chunk of memory from the heap
- We copy the string into it
- We print it out
- Finally, we free the memory

```
char *static_string = "Hello, Malloc";
char *str_buff = malloc( size: 15 * (sizeof(char)));
strcpy(str_buff, static_string);
printf( format: "%s", str_buff);
free(str_buff);
```

Malloc

- Note that we had to pass in the amount of memory we wanted
  - How much did we actually need?
  - What if we under-allocated?
- The sizeof operator is commonly paired with malloc
  - Returns the number of bytes required by the given data type

```
char *static_string = "Hello, Malloc";
char *str_buff = malloc( size: 15 * (sizeof(char)));
strcpy(str_buff, static_string);
printf( format: "%s", str_buff);
free(str_buff);
```

Malloc

- This means that the size of the value can be dynamic
  - E.g. based on user input
- It also means that large pieces of data can be allocated and efficiently passed around as a pointer
  - Vs. Copying data on the stack

```
char *static_string = "Hello, Malloc";
char *str_buff = malloc( size: 15 * (sizeof(char)));
strcpy(str_buff, static_string);
printf( format: "%s", str_buff);
free(str_buff);
```

Calloc

- Malloc makes no guarantees about the memory location it handed back to you
- Could have garbage data from previous usages
- If you want data zeroed, you can use the *calloc* function
- Note that you pass two arguments to calloc

```
char *static_string = "Hello, Calloc";
char *str_buff = calloc( nmemb: 16, (sizeof(char)));
strcpy(str_buff, static_string);
free(str_buff);
```

Malloc

- If you wanted to achieve the same guarantee with malloc, you would use the *memset* function

```
char *static_string = "Hello, Calloc";
char *str_buff = calloc( nmemb: 16, (sizeof(char)));
strcpy(str_buff, static_string);
free(str_buff);
```

Free

- Data that you allocate on the heap needs to be freed
- Unless it is assigned to a global variable and can live for the life of your program
- If you don’t free your data, you will get memory leaks

```
char *static_string = "Hello, Calloc";
char *str_buff = calloc( nmemb: 16, (sizeof(char)));
strcpy(str_buff, static_string);
free(str_buff);
```

Free

- If you free your data more than once, you will get double free error messages
- In general, when you allocate an object, you need to think about who *owns* the data
  - If a function returns a pointer to you, do you own it? Probably.

```
char *static_string = "Hello, Calloc";
char *str_buff = calloc( nmemb: 16, (sizeof(char)));
strcpy(str_buff, static_string);
free(str_buff);
```

Malloc/Free

- We are mostly used to garbage collected languages
- The advantage to manual memory management is:
  - Speed
  - (More important) no global GC pauses
- This is why most core server software is written in C, despite it not being very friendly

```
char *static_string = "Hello, Calloc";
char *str_buff = calloc( nmemb: 16, (sizeof(char)));
strcpy(str_buff, static_string);
free(str_buff);
```

Malloc/Free

- Important to understand that malloc/free are a *library*
- The allocator keeps track of what pieces of memory are being used, and reuses freed memory for new malloc calls
- Fragmentation can be an an issue with memory

```
char *static_string = "Hello, Calloc";
char *str_buff = calloc( nmemb: 16, (sizeof(char)));
strcpy(str_buff, static_string);
free(str_buff);
```

Malloc/Free

- The allocator needs to maintain data structures to keep track of things

```
char *static_string = "Hello, Calloc";
char *str_buff = calloc( nmemb: 16, (sizeof(char)));
strcpy(str_buff, static_string);
free(str_buff);
```

Malloc/Free

- The allocator uses the system calls *sbrk()* and *brk()* to ask for more heap memory if necessary
- You can use the brk and sbrk to play directly with memory allocations if you'd like
  - NB: this can really screw up malloc!

```
// get the current pointer of the break
void *initial_location = sbrk( delta: 0);

unsigned int *memoryLocation = (unsigned int *)initial_location;

/* Allocate two more memory locations */
brk( addr: memoryLocation + 2 );

/* Use the ints. */
*memoryLocation = 1;
printf( format: "p address: %u\n", memoryLocation);
printf( format: "p points to %d\n", *memoryLocation);

*(memoryLocation + 1) = 2;
printf( format: "p + 1 address: %u\n", (memoryLocation + 1));
printf( format: "p + 1 points to %d\n", *(memoryLocation + 1));

/* Deallocate back. */
brk(initial_location);
```

Review

- C programs have various memory regions that store different types of data
- The Stack is used to store function-call related information
  - The Stack grows *down* by convention
- The Heap is used to store long-lived and large pieces of data
  - You must manually allocate and deallocate memory from the heap
  - Malloc is the common way to do this, but makes no guarantees about the state of the memory you are going to get
  - calloc and memset can be used to get zero'd out data


MONTANA
STATE UNIVERSITY