Lecture 2 — Anatomy of a C Program

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 9, 2024

Credit: Based on original by Michael Cooper-Stachowsky

Beyond Basic(s)

We've covered the basics of the C programming language, but that's not enough to write a program well and manage its execution.

Next: more about C, memory, the ABI for Cortex M, compilation, and execution.

No deep-dive on compilation, sorry (take ECE 351?).

Part I

Memory Organization

Stack, Heap, Whatever?

Memory is logically divided into stack memory and heap memory.

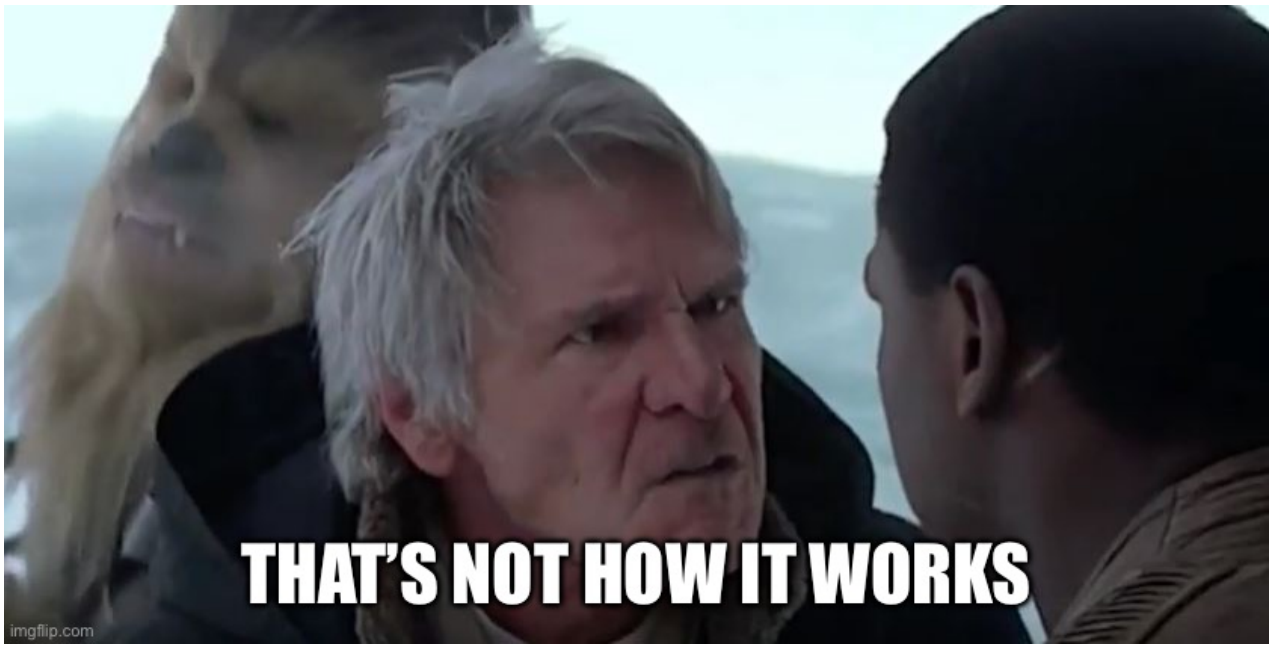
Managing the stack is a huge problem for the OS designer.

Managing heap memory is a problem for the OS designer and something program authors often get wrong.

Stack or Heap?

It’s important to know the difference between stack and heap!

Some languages let you ignore the details, but this is C, so…



Stack Memory

Stack is an area of memory where we add & remove data as it it's a stack data structure: push and pop; LIFO behaviour.

When something is put on the stack, it's in the "next" space.

Stack Memory

The used fraction grows as we push items on it, shrinks as we pop items off.

Stack space is limited!

Stack Memory

The stack is used for:

- Local variables
- The call stack
- Places for return values

The OS sets up the stack for the executing program.
You may have to do it in a microcontroller.

Heap Memory

Heap memory is dramatically larger than stack size.

Things can be added to and removed from the heap memory at any point.

We use `malloc()` for this in C!

Stack or Heap?

```
float* do_something() {
  // x is stack-allocated
  int x = 7;
  // array is stack-allocated
  char array[10];

  // a is stack allocated
  // the array it points to is heap-allocated
  float* a = malloc( 20 * sizeof( float ) );

  // y is stack allocated and the return value is copied here
  int y = do_something_else( x );

  // a is returned to the calling function (value copied)
  return a;
}
```

Temptation: always stack allocate because it is “easier”, but it is often wrong.

Lies About Memory

It turns out that we tell you a lot of lies about memory.

Such fictions are convenient when writing a simple program…

But will be a problem at this level.

Lies About the Stack

There is only one stack!

Actually, one per thread.

Stack grows up!

Implementation specific! Growing up is just convenient in drawing a diagram.

Stack memory is protected!

From the application point of view, maybe? But not in an OS or microcontroller.

Cortex M4 Stack Management

One stack pointer indicates where we are in the stack for the running thread.

There are two “banked” stack pointers (MSP, PSP) for easy switches.

Anywhere in memory could be the stack for a given thread, but only one is referenced by the SP at any time.

MSP and PSP

MSP: Main Stack Pointer

- In single threaded mode, only one stack, only use MSP
- Exclusively used in interrupts (important!)

PSP: Process Stack Pointer

- In multithread mode, each thread has its own stack
- Holds the address of the currently running thread
- Never used in interrupts (processor fault if you try)

The heap is a larger area of memory for dynamic allocation.

Heap underflow isn’t actually a thing, though.

Heap Management

Application programs request memory (e.g., `malloc()`) and return it (`free()`).

The operating system has to manage the heap…

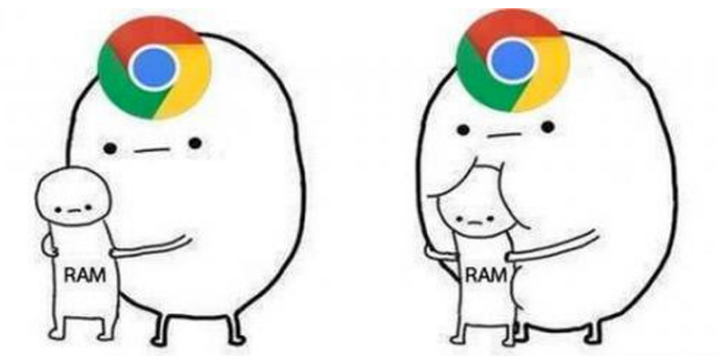
Remember what has been allocated and to whom!

Every allocation must lead to a deallocation.

All Mine

Users forget to free something? Memory Leak.

Leaking memory is unfortunately common. On a desktop…



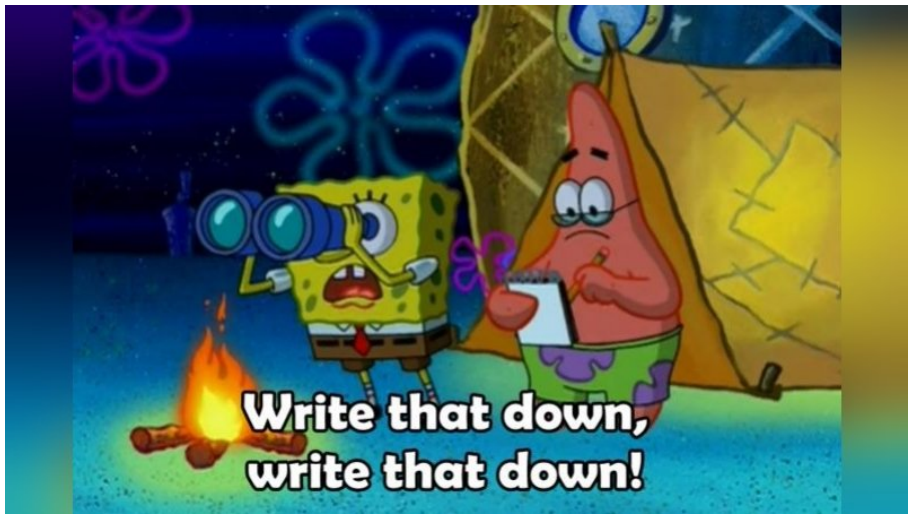
In an embedded system? Crash or overwriting things.

Managing The Heap

Later on, we'll return to how to dynamically allocate memory. For now:

An OS needs to find available memory to satisfy an allocation.

To do so, it has to know what is allocated and where.



Deallocation is just the inverse of allocation.

Memory Alignment

We've introduced the idea of a structure, but we need to think about alignment.

```
struct thing {
  int x;
  double y;
  char z;
}
```

What is the size, in memory, of this structure?

Did you say 13 bytes?

How did we get to that number?

We did the math, right?

1. `int` is 4 bytes
2. `double` is 8 bytes
3. `char` is 1 byte
4. So $4 + 8 + 1 = 13$ $\square$

Except: the C standard says `int` is minimum 2 bytes...

And we didn't think about alignment.

Memory Alignment

Memory alignment depends on the processor, the OS, and the compiler.

Example: in my 20 year old PowerPC system a `bool` compiled by `gcc` is 4 bytes.

It also draws 180W at idle. Beautiful case though.

But Why Alignment?

Why is there alignment? Memory boundaries!

Memory can be byte-accessible without being byte-addressable.

That is to say, we might not be able to directly access any arbitrary byte.
The workaround might involve reading 4 bytes, modifying a bit, and writing 4.

Gotta Go Fast

Even if memory is byte-addressable, it might be slower to have unaligned types.

Aligned in this case: start address is an even multiple of word-size.

An assignment statement might be one assembly instruction when aligned…
But two if it's unaligned, for double the work.

… And ten times the headache: what happens if it's interrupted partway?

Wasting Memory?

Does this mean alignment sometimes forces us to waste memory?

Yes! But usually small amounts and it's a space-for-speed tradeoff.

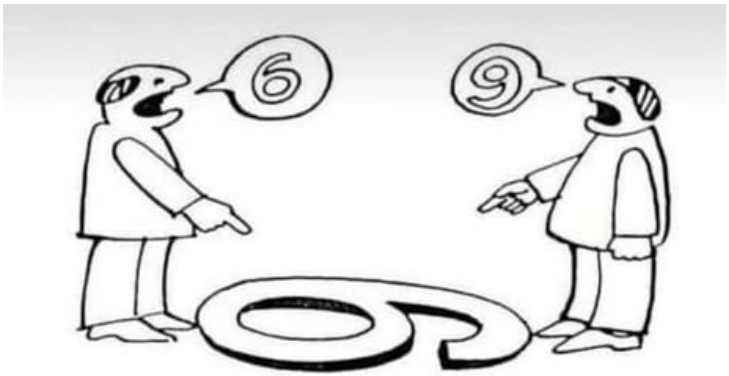
Break Rules?

It’s possible to manually pack a structure, but the need for this is rare.

It’s something we’ll touch on if you take ECE 459 – Programming for Performance.

Is My Green Your Blue?

Another worry: if transmitting data between systems, does it have the same struct representation on the other side?



tRuTh Is ReLaTiVe, EvErYtHiNg DePeNdS oN yOuR pOiNt Of ViEw

Someone drew that number. This someone knows if it's a 9 or a 6. Thus, there exists a truth.

Do We Care?

In short, yes. OS development usually works in arrays of bytes.

These can be converted via pointer-casts to different types.

If the alignment is not respected, we get undefined behaviour.

Also: if a request for an array of 10 `bools` happen, how big is that?

Sort of Packing?

One thing we can do and that we often see in registers is bitfields!

A bitfield uses each bit of a register or struct to indicate individual things.

They are in order from LSB to MSB and any bits you don’t specify are just padding.

Global Variables

What about global variables – where do they go?

They actually have their own segment.

Part II

High-Level Overview of Compilation

What is Compilation?

Compilation is the art of turning the source code into the executable.

It has a lot of steps: read in the files, analyze, link, optimize, generate…

We care about three important parts: preprocessing, object generation, linking.

Preprocessing

You've learned about a few things that have the # in C.

These are precompiler directives.

That's exactly what is sounds like: do this before main compilation.

Precompiler Directives

Some example precompiler directives"

- `#include`
- `#define`
- Macro Expansion
- `#ifdef/ifndef/endif`
- `#pragma`

Macro Expansion

You can do something like `#define min(x,y) ((x)<(y)?(x):(y))`

Hard to get right: need to remember brackets everywhere to avoid issues.

In most circumstances: just write a function!

Macro Expansion

Function-like macros are not functions, they don't really exist.

Think of the define directive as copy-paste.

One of the most important things the compiler does for you is check correctness at compile-time.

Use of define macros or other precompiler magic prevents this from working.

The Compiler is There to Help

If you must use macros: keep them short and simple.

Compilation

Simple projects just contain one code file and maybe a header file.

More complicated projects have many more code and header files.
    Sometimes things in other languages (assembly, Rust, who knows).

The compiler doesn't just copy-paste everything into one megafile...

Object Files

Solution: files are compiled into objects (`*.o`) files.

Object files are like incomplete executables.

- They have executable code in them that can run, but…
- They have unresolved “links“ and no `main` start point.

When making the final executable: bring all the objects together!

Resolving the Links

Resolving links is the job of the linker.

Resolving the Links

The linker combines all executables and makes sure the links are connected to the right places.

The linker also handles referencing shared libraries (e.g., `glibc`).

We won't focus on how the linker does the job – that's for the compiler course!

Part III

Executable Organization

Minimum Requirements for an Executable

The executable code that goes in the `.text` section.

The start symbol/address (`__start()` which calls `main()`.

Other Sections

`.data`: Global or static variables initialized to nonzero values

`.rodata`: Read-Only data

`.bss`: Global/static variables initialized to 0. Just reserved space.

So You Want to Run a Program

A program is your compiled code.

A process is a program in execution.

A process is a program in execution.

1. The instructions and data.
2. The current state.
3. Any resources that are needed to execute.

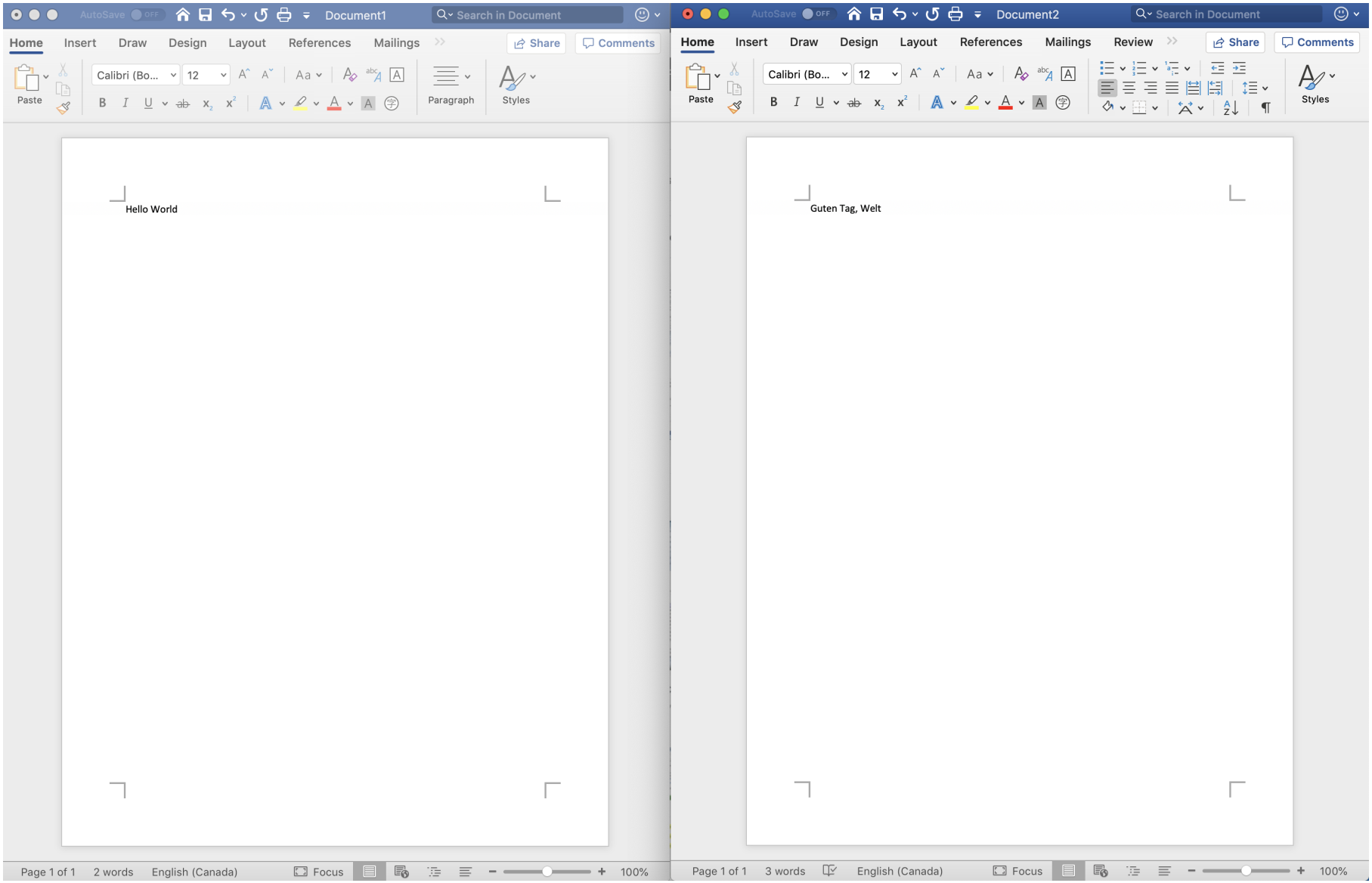
Processes vs. Programs

Note: two instances of the same program running equals two processes.

You may have two windows open for Microsoft Word, and even though they are the same program, they are separate processes.

Similarly, two users who both use Firefox at the same time on a terminal server are interacting with two different processes.

Two Documents, Two Processes



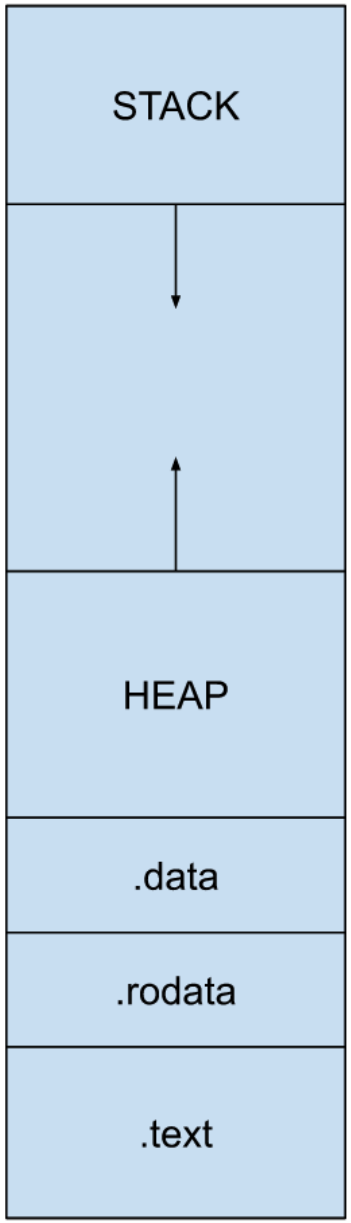
Assigning Locations

At compile time, the compiler resolves static variables/links.

At load-time, the system sets up memory segments and links shared objects.
After that, go to the start symbol!

At run-time: resolve dynamic allocation and, oh yes, execute the code.

Executable in Memory



And Then We Execute

Once the program is launched, what happens next?

The code runs and does whatever the programmer asked it to do…

Unless the operating system or system needs to step in…

But we need to talk more about computer organization first.