C in the UNIX Environment

Jacob Koziej (EE '25) Charles Van West (EE '24)

The Cooper Union for the Advancement of Science and Art

August 19, 2022

Background

- There are a few things that we must get out of the way before we can start diving into some C code.
- By modern standards, C is a pretty low-level language. As such, it helps to have a bit of understanding of what's happening behind the scenes after we write some code.

Memory

- As you probably know from pop-culture, or from your own adventures into the world of digital electronics, traditional computers represent all data as either zeros or ones (bits).
- All of these bits have no intrinsic meaning, and it is up to *us*, the programmer, to decide how to interpret any sequence of zeros and ones.

Lets look at the following 32-bits:

```
0110 1000 0110 1001 0010 0001 0000 0000
```

The following bit pattern (in big endian[1]) can be interpreted as:

`"hi!\0"`	$\Longrightarrow$	C-style ASCII string
`1751720192`	$\Longrightarrow$	unsigned 32-bit integer
`4.40368053225e+24`	$\Longrightarrow$	32-bit float (IEEE 754[2])

- Since C is a low-level language, there is very little in the way of stopping us from interpreting memory in any way we'd like!
- Clever programmers can exploit data's representation in memory to squeeze out every last bit of performance!
- Be respectful of this power! It is just as easy to corrupt memory and cause a program to crash & burn!

[1]Endianness - Wikipedia
[2]IEEE 754 - Wikipedia

With this in mind, let's cover some of the most common primitive data types (on the `x86_64` platform):

`char`	$\Longrightarrow$	an 8-bit integer (used for characters)
`int`	$\Longrightarrow$	a signed 32-bit integer
`long`	$\Longrightarrow$	a signed 64-bit integer
`float`	$\Longrightarrow$	a signed 32-bit floating point value (IEEE 754)
`double`	$\Longrightarrow$	a signed 64-bit floating point value (IEEE 754)

Note that according to the ISO C standard, a `char` can be signed or unsigned depending on the compiler implementation!

Compilers

"Programming is the art of creating a set of instructions for a machine to execute."

- Since a computer can only understand ones and zeros, it's a huge pain to write out everything in raw machine code. (Keep in mind that someone out there had to do this!)
- The next best thing to writing raw machine code is human readable machine code called assembly.
- Although assembly makes writing machine code a far more enjoyable experience, it is still a far cry from the convenience of a language like C.
- Depending on who you ask, C might be a high-level language. (Jacob & Charles fall into that category ;))

For an example, let's look at `JAVK`[1] assembly & machine code:

```
| _start:                                              <-+->        0x0000:
|         # clear necessary registers                    |
|         and     z                                    <-+->                3f
|         mva     i                                    <-+->                88
|         mva     j                                    <-+->                89
|                                                        |
|         # we'll write our results to 0x8000            |
|         lnh     0x8                                  <-+->                b8
|         mva     i                                    <-+->                88
|                                                        |
|         # we'll use 0x55 as our right operand          |
|         lnh     0x5                                  <-+->                b5
|         lnl     0x5                                  <-+->                a5
|         mva     b                                    <-+->                81
|                                                        |
|         # 0xff + 0x55                                  |
|         lnh     0xf                                  <-+->                bf
|         lnl     0xf                                  <-+->                af
|         add     b                                    <-+->                01
|         stb     0x0                                  <-+->                d0
                                                         |
---------------------------------------------------------+--------------------------------
                       Assembly                          |          Machine Code
```

[1] github.com/javk-cpu

- To get around the headache of machine code, very smart people came up with programming languages.
- One such language is C, which originated at Bell Labs back in 1972!
- Today C is the most popular programming language in the world, and there's a safe bet that almost any computer architecture you'll come across in the wild has some form of a working C compiler!

- Coming up with a language is only half the battle, the far more difficult part is writing a working compiler (the program that converts easy to write code to machine code).
- Without compilers, most of today's software we take for granted would simply be infeasible to write and/or maintain.
- Modern compilers do far more than just convert our code to machine code: they apply optimizations, give warnings about undefined behavior, and output helpful error messages when they run into syntax errors!

Although C is an amazing language, its age does show...

```
static const unsigned char bar = '*';

static volatile const void *foo = &bar;

void ****(*(*fizz)(void *buzz))(void****);

int x = ~((1 << 16) >> 4) ^ ((0xc001 | 0xbaad) & 0xffff);
```

In case you're wondering, yes, all of the code above is **valid** C code, and yet, this is a rather tame example compared to the code found at The International Obfuscated C Code Contest[1].

[1] www.ioccc.org

First Steps

Now that we've covered some basics, it's time to jump into an editor to write a timeless classic!

Hello World!

As per tradition, let's write a hello world program:

```
#include <stdio.h>

int main(void)
{
        printf("Hello World!\n");

        return 0;
}
```

We'll save the file as `hello-world.c` and then compile it with:
`cc -o hello-world hello-world.c`
Next, we'll run the program like so: `./hello-world`
Finally, we should see `Hello World!` printed to our terminal!

Let's Break it Down

`#include <stdio.h>` — include the system's standard buffered input/output header.

`int main(void)` — our `main()` function which returns an `int` and takes `void` (no) inputs. Everything inside the `{ }` is considered the body of `main()`.

`printf("Hello World!\n");` — print the format string `Hello World!\n` to `stdout`. The `\n` is an **escape sequence** for a newline character.

`return 0;` — return a successful exit code of 0 to the host operating system.

POSIX Programmer's Manual

- As we may be aware from the intro to the shell, there's a hefty little utility called `man`.
- Thankfully for us, people have spent a significant amount of time documenting most of the functions we'll find in the C Standard Library (libc) along with system calls (functions that directly interface with the host operating system) in the POSIX Programmer's Manual.
- Let's see how we can navigate this documentation to get a jump start on other useful functions!

If we run `man stdio.h`, we'll see a lot of useful information:

- `NAME` — a short description of the header's purpose.
- `SYNOPSIS` — concise usage information.
- `DESCRIPTION` — a general description of all constants, types, and functions we'll find in the header.
- `APPLICATION USAGE` — a description of why we'd want to use the header in our code.
- `SEE ALSO` — other headers that may be of interest.

With this in mind, how about we take a look at the `man` page of `printf()`?

Let's run `man 3 printf`

Note:

By adding the 3 before `printf` we're telling `man` to open the library call section of the man pages. Leaving out this number will cause `man` to open the first man page with the same name (most likely our shell's implementation of `printf`).

Again we'll see some more useful information:

- `DESCRIPTION` — function usage information.
- `RETURN VALUE` — information related to success and error return codes.
- `NOTES` — important things to keep in mind while using the function (or function family).
- `BUGS` — implementation-related bugs to be aware of!
- `EXAMPLES` — example code showcasing possible use cases.

The man page for `printf()` is rather overwhelming. Instead, lets take a look at a far more simple function that behaves similarly for our use case!

Let's open the man page for `puts()` with `man 3 puts`

- Again we see many of the same sections from the previous man page, but we might see that some sections are missing (like `EXAMPLES`).
- Man page length varies based on the complexity of the function, and as we can see, the function family that `puts()` belongs to is rather simple compared to the `printf()` family!

With this in mind, we can see that unlike `printf()`, `puts()` appends a trailing newline character to input strings. From this we can deduce that:

`printf("Hello World!\n");` ⇔ `puts("Hello World!");`

The takeaway here is that there are multiple ways of doing the same thing while programming, and it is up to *us*, the programmer, to decide what's the best tool for the job!

Breaking Things

Now that we have a working hello world program, why don't we break it by visiting the underworld!

Hello Underworld?

Let's modify our existing hello world program to print out hello world to the first 8 underworlds (with a few bugs & errors)!

```
#include <stdio.h>

int main(void)
{
        int i = -1

        while (i > -8) {
                printf("Hello %s World!\n", i);
                i = i - 1;
        }

        return 0;
}
```

Compiler Errors

When we try to compile the previous code, the compiler fails and outputs the following error message:

```
$ cc -o hello-underworld hello-underworld.c
hello-underworld.c: In function ‘main’:
hello-underworld.c:7:17: error: expected ‘,’ or ‘;’ before ‘while’
    7 |                 while (i > -8) {
      |                 ^~~~~
$ _
```

Here the compiler is throwing a helpful error that indicates a missing ; at the end of line 5. Although the compiler printed line 7 to `stderr`, it has given enough information to determine the cause of the error.

To make determining compiler errors easier, it helps to keep the following in mind:

- The compiler parses each file sequentially meaning that an error will always be on or before the line(s) printed to `stderr`.
- When searching for error messages online, make sure to remove system-specific strings to further generalize your search.
- **Read the errors** before searching online! This **cannot** be stressed enough! You'll be surprised by how helpful modern compiler suggestions can be!

Compiler Warnings

After sorting out the syntax error and once again calling the compiler, we'll run into the following warning message:

```
$ cc -o hello-underworld hello-underworld.c
hello-underworld.c: In function ‘main’:
hello-underworld.c:8:40: warning: format ‘%s’ expects argument of type ‘char *’, \
                                  but argument 2 has type ‘int’ [-Wformat=]
    8 |                         printf("Hello %s World!\n", i);
      |                                       ~^            ~
      |                                        |            |
      |                                        char *       int
      |                                       %d
$ _
```

What is interesting is that the compiler successfully compiled our code, and yet if we run our code we'll see the following runtime error:

```
$ ./hello-underworld
[2]    760163 segmentation fault (core dumped)  ./hello-underworld
$ _
```

Needless to say, this isn't what we're expecting!

The important takeaway from this is that even though the compiler created a working program (with warnings), it is not guaranteed to run without crashing! Again, like with compiler errors, it helps to keep the following in mind:

- Compiler writers are extremely intelligent people, there's probably a good reason for why a warning exists!
- You can enable more compiler warning with the `-Wall`, `-Wextra`, and `-Wpedantic` flags.
- When you're trying to do some hacky memory manipulation **and you are confident** that what you are doing will work, you can suppress a compiler type warning by casting the "correct" data type. Just keep in mind that this may lead to undefined behavior!

Runtime Errors

Now that we've replaced the %s with a %d, and our program compiles without any errors or warnings, we still see that something is quite not right:

```
$ ./hello-underworld
Hello -1 World!
Hello -2 World!
Hello -3 World!
Hello -4 World!
Hello -5 World!
Hello -6 World!
Hello -7 World!
$ _
```

It looks like we're not printing hello to the -8th underworld!

From this runtime error, we can deduce that our loop logic is incorrect, and the fix is trivial: change > (less than) to >= (less than or equal to) so that `i` can reach the value of −8.

Again, here are a few things to keep in mind when dealing with runtime errors:

- **Slow down**. Write out your program logic on paper step by step to see where things may have gone wrong.
- **Program in small chunks**. Debugging lots of small errors one at a time is always going to be more enjoyable than dozens of huge errors.
- **Take a break**. As tempting as it can be to sit and read over your code dozens of times, it may help to take a small break and come back with a fresh mind.

Nasty Bugs

This may be a shock, but a compiler is far from a perfect machine! To see this, we'll change %d to %u in our program.

```
$ cc -Wall -Wextra -Wpedantic -o hello-underworld hello-underworld.c
$ ./hello-underworld
Hello 4294967295 World!
Hello 4294967294 World!
Hello 4294967293 World!
Hello 4294967292 World!
Hello 4294967291 World!
Hello 4294967290 World!
Hello 4294967289 World!
Hello 4294967288 World!
$ _
```

Even though we enabled all compiler warnings, no warnings or errors are printed even though we used the wrong format character in our format string! Clearly the output is incorrect, but what are we to make of this?

It goes without saying that C is a *very* broken language. There is really nothing in the way of stopping you from doing silly things like treating signed integers as unsigned integers, just as we have!

The question then becomes how do we deal with errors like these? Clearly printing values would not have worked in this situation. . .

Debuggers

You most certainly can go about debugging with functions in the `printf()` family, but it will only be a matter of time before you realize such a method is rather ineffective. . .

To get around this, really smart people came up with an even more impressive tool called a debugger!

Before we go running off to our shiny new tool, we'll need to recompile our program with debugging information. To do so, add the −g flag:

```
$ cc -g -Wall -Wextra -Wpedantic -o hello-underworld hello-underworld.c
$ _
```

To debug, we'll be using The GNU Project Debugger (GDB). You can open the program we just compiled in GDB like so:

```
$ gdb hello-underworld
GNU gdb (Gentoo 12.1 vanilla) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello-underworld...
(gdb) _
```

Upon typing the command, you'll be dropped into an interactive GDB session. At any time you can return to the shell by typing `exit`.

Now, GDB is quite an overwhelming program, but with just a handful of commands, it can become a powerful tool to add to your arsenal!

- `break` — stop the debugger at either a function or line number.
- `clear` — clears a breakpoint at the specified location.
- `watch` — sets a watchpoint which stops execution of your program whenever the value of an expression changes.
- `print` — prints the value of an expression.
- `step` — step into a function call.
- `next` — step over a function call.
- `continue` — continue execution until the next signal or breakpoint.
- `list` — print the specified function or line.

Before we get to debugging, it will be helpful to enable the terminal user interface (TUI) of GDB. Quit GDB and relaunch it, but this time add the `--tui` flag before the program we'll be debugging:

```
$ gdb --tui hello-underworld
...
(gdb) _
```

Upon relaunching GDB, we will be greeted with a view of our source code along with an interactive GDB shell.

Now that we have GDB in a more *"usable"* state, how about we debug our program using the commands we just learned?

Note:
If the TUI starts to amass graphical artifacts from stuff being printed, you can force a full redraw by typing `refresh` or pressing `ctrl + l`.

Lets set a breakpoint at `main()`:

```
(gdb) break main
Breakpoint 1 at 0x113d: file hello-underworld.c, line 5.
(gdb) _
```

If we were to run our program we do see something interesting occurs:

```
(gdb) run
Starting program: hello-underworld
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, main () at hello-underworld.c:5
5               int i = -1;
(gdb) _
```

What we see is that program execution has actually stopped at the first line of `main()`, just as we instructed!

Since we were having an issue with printing the value of `i`, how about we set a watchpoint?

```
(gdb) watch i
Hardware watchpoint 2: i
(gdb) _
```

Now that we have a watchpoint, lets continue execution of our program until `i` changes:

```
(gdb) continue
Continuing.

Hardware watchpoint 2: i

Old value = 32767
New value = -1
main () at hello-underworld.c:7
7               while (i >= -8) {
(gdb) _
```

If we were to continue program execution until the current process exits, we would observe that `i` is correctly decremented in our loop. So what gives?

To figure this out, how about we now instead set a breakpoint at our `printf()` call:

```
(gdb) break 8
Note: breakpoint 1 also set at pc 0x113d.
Breakpoint 2 at 0x1146: file hello-underworld.c, line 8.
(gdb) _
```

Notice that to stop program execution at the call to `printf()` inside of `main()` we instead now use a line number.

Once again, if we run our program we'll see that program execution stops at the beginning of `main()`, and then after continuing, at line 8 of `hello-underworld.c`:

```
...
(gdb) continue
Continuing.

Breakpoint 2, main () at hello-underworld.c:8
8                     printf("Hello %u World!\n", i);
(gdb) _
```

Now that we're at the point of the program right before the first call to `printf()`, how about we check the value of `i`?

```
(gdb) print i
$1 = -1
(gdb) _
```

Once again, it seems that `i` has the correct value. Let's see what's printed after we call `printf()`:

```
(gdb) next
Hello 4294967295 World!
9                            i = i - 1;
(gdb) _
```

What we see here is that the wrong value of `i` is printed here even though it had the correct value! Let's print the value again, but this time as an unsigned value:

```
(gdb) print (unsigned) i
$1 = 4294967295
(gdb) _
```

From this output that we can deduce that `printf()` must be interpreting `i` as an unsigned integer!

Now of course we knew that this would be the bug in our case, but the important takeaway from this exercise is the process of finding a bug!

GDB is a *very* powerful & complex tool very few people actually have mastered, but even with a few GDB commands under our belt, we can become rather effective at using GDB!

Again, nobody is going to stop you from `printf()` debugging, but you do put yourself at a rather large disadvantage as you cannot utilize any of the memory inspection commands and/or stop execution based on some kind of predefined condition. More importantly, by adding a `printf()` you're actually modifying your program, something you should refrain from while debugging!

External Resources

`Professor Jacob Sorber's Beginner C Videos`
Short & Sweet videos to introduce beginner C topics.

`Beej's Guide to C Programming`
An excellent, but rather *dense* free online book that covers the C programming language.

`The C Programming Language, 2nd Edition`
K&R is the *"de facto"* book to learn the C programming language, but its style is a little dated.