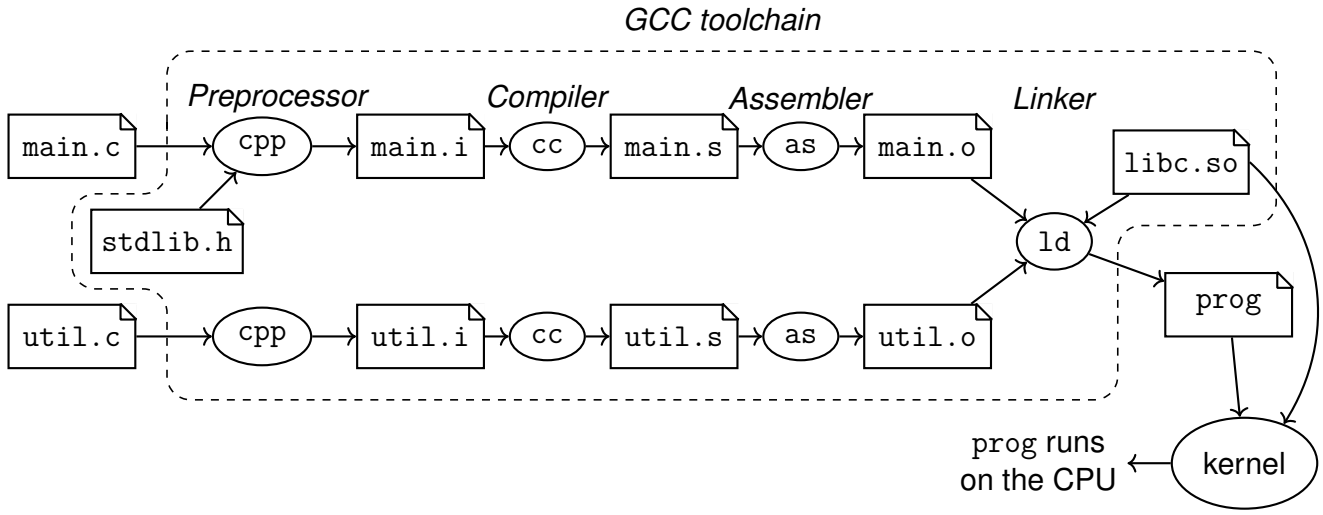
Chapter 2 – Compiling and Debugging a C program

The role of the *computer processor* aka the CPU (e.g. Intel i9, ARM Cortex) is to execute machine language instructions, one after the other. But a modern computer typically runs several programs "at the same time" on the same CPU. The *kernel* (e.g. Linux) is a special software program acting as a "conductor" for other programs: its role is to load executable files from the disk into memory, and somehow have the processor execute them all concurrently. An *Operating System* (e.g. Ubuntu) is a collection of programs, including a kernel, a shell etc, that work well enough together to make the machine usable by humans.

1 The GNU Compiler Collection toolchain

There are several steps necessary to build an executable program from source files, but modern toolchains hide this complexity behind the scenes. The diagram below illustrates what happens when we type `gcc -o prog main.c util.c` to build `prog` from hypothetical source files `main.c` and `util.c`, and then `./prog` to execute it.



Exercise Create a file named `hello.c` and type in the program below. Compile it with command `gcc -o hello hello.c`, then run it with `./hello`. Note: function `puts()` (i.e. "put string") writes some *character string* onto the screen, followed by a newline.

```
#include <stdio.h>

int main() {
    puts("Hello, world");

    return 0;
}
```

Exercise Now we will ask GCC to stop after a particular step:

- `gcc -E` runs the preprocessor, but does not compile.
- `gcc -S` runs cpp and compiles the result, but does not assemble.
- `gcc -c` preprocesses, compiles, and assembles, but does not run the linker.

Using these commands (with `-o filename`), produce `hello.i`, `hello.s`, and `hello.o` respectively.

Exercise Open `hello.i` in a text editor and browse the code until you locate the `main()` function. Everything above comes from file `/usr/include/stdio.h` (open this one too) and was copy-pasted here by the preprocessor. Notice that the comments from `stdio.h` have been removed. In `hello.c`, try and add some */* comments */* anywhere and run the preprocessor again.

Exercise Object files contain binary code, not ascii-encoded text, so we cannot open them directly in a text editor. Type `xxd hello.o > hello.xd` to perform a "hex dump" then open the resulting file in a text editor. It will look more or less like this:

```
00000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 3e00 0100 0000 0000 0000 0000 0000  ..>.............
00000020: 0000 0000 0000 0000 6802 0000 0000 0000  ........h.......
00000030: 0000 0000 4000 0000 0000 4000 0e00 0d00  ....@.....@.....
00000040: 5548 89e5 bf00 0000 00e8 0000 0000 b800  UH..............
00000050: 0000 005d c348 656c 6c6f 2c20 776f 726c  ...].Hello, worl
00000060: 6420 2100 0047 4343 3a20 2847 4e55 2920  d !..GCC: (GNU)
00000070: 3133 2e31 2e31 2032 3032 3330 3631 3420  13.1.1 20230614
00000080: 2852 6564 2048 6174 2031 332e 312e 312d  (Red Hat 13.1.1-
00000090: 3429 0000 0000 0000 0400 0000 2000 0000  4)..........  ...
000000a0: 0500 0000 474e 5500 0200 01c0 0400 0000  ....GNU.........
000000b0: 0000 0000 0000 0000 0100 01c0 0400 0000  ................
```

The first column indicates the position in the file, the middle columns are the contents of the file in hexadecimal (two bytes per column) and the right column is the same contents but interpreted as ASCII. For instance, 0x45 0x4C 0x46 are the ASCII codes of letters "E", "L" and "F", which indicate that our file is encoded in Executable and Loadable Format. Locate the *"hello world"* string in the hex dump, and write down the ASCII codes for each letter.

Exercise Reading hex dumps is very crude. Fortunately GCC offers the `objdump` tool, which purpose is to *disassemble* machine code into a readable listing. Type `objdump -D hello.o > hello.lst` then open the resulting file. It will look more or less like this:

```
hello.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   bf 00 00 00 00          mov    $0x0,%edi
   9:   e8 00 00 00 00          call   e <main+0xe>
   e:   b8 00 00 00 00          mov    $0x0,%eax
  13:   5d                      pop    %rbp
```

- Notice how bytes are now *grouped* into what `objdump` believes are instructions: 0x55 is the machine encoding for `push %rbp`, 0x48 0x89 0xe5 is the machine encoding for `mov %rsp,%rbp`, etc. On Intel processors instructions don't always occupy the same number of bytes
- Notice that memory addresses (for instance, the destination address of the `call` instruction) are not known at this stage. They will be overwritten by the linker.
- Notice how certain parts of the listing don't make sense as instructions: where are the bytes that encode our *"Hello, world!"* string ?

2 The GDB Debugger

One of the reasons programming in C is hard is because the language provides very few abstractions or guarantees against programming errors. To help with that, the GNU toolchain offers the `gdb` debugger, a tool which allows to execute the program step-by-step and inspect memory contents interactively.

Exercise Compile `hello.c` once again, this with `gcc -g` so as to keep *debug info*[1] in the executable: `gcc -g -o hello ./hello.c`. Then, launch the program inside the debugger:

```
$ gdb -q ./hello
Reading symbols from ./hello...
(gdb)
```

Play with these gdb commands:

- `list` shows source code
- `help CMD` gives some help about a command (try it with all commands !)
- **`break`** `LINENUM` or **`break`** `FUNCNAME` inserts a breakpoint at specified location
- `start` adds a temporary breakpoint on `main` and executes until there
- **`continue`** executes the program until it finishes or hits a breakpoint

Exercise Create a file named `fact.c` and type in the program below. Compile it with `gcc -g`.

```
#include <stdio.h>

int factorial(int n) {
    if (n<=1)
        return 1;

    return n * factorial(n-1);
}

int T[10];

int main() {
    for(int i=0; i<10; i++)
    {
        T[i] = factorial(i);
    }

    return 0;
}
```

Play with these gdb commands:

- `step` executes one line of source code, stepping *into* function calls
- `next` executes one line of source code, stepping *over* function calls
- `until` executes until the program reaches the following line in the source code (stepping over calls and loops)
- `where` displays the call stack
- `finish` executes until the current function returns

Advanced usage: `step N`, `next N`, `until LOCATION`. Use `help` to learn more about these commands, and read the documentation at `https://sourceware.org/gdb/current/onlinedocs/gdb.html/Continuing-and-Stepping.html`

Exercise Make the program execute until the loop is finished and show the contents of the array with command `print T`.

Exercise In the loop body, change the `T[i]` to `T[i+1000]`. The program will now try to write far beyond the end of the `T` array, which will typically causes a crash at execution time, with a `Segmentation fault` error message.[2] Execute the program in `gdb` to see what line of source code causes the crash.

[1] debug info = a mapping between memory addresses in the program and line numbers in the source file

[2] note that we have no guarantee that execution will fail. An incorrect C program may run all the way until the end, and it may produce a correct or incorrect result. Or it might crash. Try with `T[i+10]` instead and observe what happens.

Exercise On what iteration does the segmentation fault happen ?

Exercise Use the `x` command to "examine" memory and observe values which have been produced before the crash.
This command is spelled `x/nfu ADDR` where n,f, and u arguments are optional.

- `ADDR` is a memory address or a symbol name
- `n` is the desired length, in number of values (default is 1 value)
- `u` is the size of one value: `b` (bytes), `h` (2 bytes), `w` (4 bytes, default) or `g` (8 bytes)
- `f` is the desired display format: `s` (text string), `i` (machine instructions) or `x` (hexadecimal, default)

For more info, type `help x` or read the GDB user manual at `https://sourceware.org/gdb/current/onlinedocs/gdb.html/Memory.html`

Exercise Other commands are available to interact with the program at low-level:

- `disassemble` shows assembly code for the current function
- `stepi` and `nexti` are similar to `step/next` but they work in terms of machine instructions

Play with these commands too (hopefully you won't need them too often in real life).