Note 3: C Basics

Kevin Moy

May 2020

Abstract

C, probably the worst programming language of all time, is nevertheless an essential and foundational low-level (some consider it mid-level) language that unlocks the interface between software and hardware. We'll learn the basics of it in this note.

1 Review

Remember that N bits represents 2^N things. We have 5 possible integer representations: 2's complement (now universally used in computers), unsigned, one's complement, sign and magnitude, and biased. Unsigned 32-bit (4 byte) integers in C are typed `uint32_t`, while 2's complement 32-bit integers are represented as `int32_t`. *Overflow* occurs when some arithmetic result goes beyond the bounds of the computer's number representation and is thus not representable.

2 Compilation of C Programs

Once we've written a C program to completion, before we run we must **compile** it by turning it into machine code or bytecode - essentially just a string of 1s and 0s that only machines would understand. Note also that a lot of this bytecode is **architecture-specific**- a chunk of 0s and 1s might mean one thing on a 32-bit computer and another on a 64-bit. Note this is a large contrast from Java, which compiles to platform-independent bytecode: one universal meaning on all machines.

In C, our desired product is an **executable**, which should either show up in your working directory with file extension `.exe` (Windows) or `.out` (Bash shells). Running this executable will then run the C program. Compilation is thus mainly a 2 step process:

1. **Compile** `.c` files into object files (`.o` files).
2. **Link** `.o` files into executable.

2.1 Compilation: Pros

The main difference between compiled languages and *interpreted* languages (Python and Scheme) is that compiled languages make intermediate machine/object code, while interpreted languages do not. Why could this be an advantage?

One advantage is general **faster runtime performance**. The fact that C is architecture-dependent isn't just a restriction- C works to optimize a program's runtime to a specific architecture.

Additionally, C saves a lot of compilation time in that only *modified* files need to be re-compiled in multiple runs of a program. So if we already ran a 1000-file project and only tweaked `main.c`, we only have to compile 1 file instead of 1000.

2.2 Compilation: Cons

C's architecture-specific quality is also a disadvantage: all compiled files, include the executable, depend on both the CPU and OS type.

Code must therefore be **ported** to new computer architectures- i.e. compilation/building/linking must be re-done on every computer, which is generally a pretty slow process.

3 C Program: Dissection

Let's dissect the basic structure and components of a C program.

3.1 Main

All C programs begin code execution in the file that contains the `main` function, specifically the function with **signature**:

```
int main(int argc, char *argv[])
```

`main` has two arguments:

- integer `argc`, which specify the number of arguments provided in the command line when running the program (**+ 1 for executable name**).
- Array of char pointers (i.e. array of strings) `argv`, which contain the exact argument strings (does not include executable).

So if we run `test -f 32`, we run executable `test` with arguments "-f" and "32" in `argv`, and `argc` is 3 in this case.

3.2 Variables

Variables in C are extremely similar to Java, except for a few key differences:

- Declared before usage, at beginning of function.

- Unitialized variables hold garbage (not null).

3.3 Memory and Pointers

Memory, conceptually, is a single massive array of empty space. Each cell in the array has some unique address mapped to it. Of course, when we store things in memory, the appropriate cells store their values. It is often easy to confuse memory addresses with values stored at that address- do NOT confuse them.

Pointers are special variables that hold **addresses** of other variables. An address refers to, or *points* to a specific cell in the memory array. Interestingly, these pointers are actually stored in memory cells themselves, so they are in effect an address located at a (different) address!

Pointers are *incredibly* important to both efficiency and storage of large objects. Pointers can point to any data type (`int`, `char`, `struct`, etc.), but normally only point to a single declared type. (There is an exception with the **generic pointer**, with type `void *`- this can point to any arbitrary type. But use these sparingly, as they can easily become dangerous (you point to and edit stuff you don't want to) if you don't know their purpose).

The `&` operator gets the memory address of a variable. We indicate pointer declaration with an asterisk before the variable name. For example, in the program below:

```
int x = 10;
int* ptr;
ptr = &x;
```

We first create an integer variable `x` and initialize its value to 3. Then, we create a pointer variable `ptr`, and store the *address* of x, denoted `&x`, in `ptr`. Conceptually, we think of `ptr` as now pointing to the memory location of `x`.

Right now, the value of `ptr` is the memory address of x, some nondeterministic hex value. To get the actual value stored at that address, we need to **dereference** the pointer, by using an asterisk on the pointer itself:

```
printf("p points to %d\n", *p)
```

We can also utilize the dereference operator to change the value at a memory address, by combining dereference and assignment. For example, if we want to change the `x` to 100 through `ptr`:

```
*ptr = 100;
```

3.4 Pass-By-Value

C is a **pass-by-value language**. This means that functions receive a *copy* of the argument passed in- changing this copy will not change the original. For example, if we have function `foo(x,y)` and we call with pre-initialized arguments:

```c
void dub_inc(int x, int y) {
    x = x + 1;
    y = y + 1;
}
int x, y;
x = 1;
y = 1;
dub_inc(x,y);
```

`x` and `y` will remain 1 after `dub_inc` ends. Unless declared otherwise (with `extern` or `static`), variables in C have **local scope**: They only are "known" in the function they are declared in. So a hundred functions in a file can declare an integer variable `x` without any issues, although this is definitely bad for readability.

However, note that with pointers, we can easily implement a pass-by-reference function into C. Try to figure out how to edit `dub_inc` above to "truly" increment both arguments passed in.

Note that if we pass in arrays, which are technically pointers to the first element, then we will be sort of implementing **pass-by-reference**: we're not changing the pointer value (an address), but the array pointed TO by that pointer.

4 Conclusion

We've now covered the basics of C. Important principles as far as C coding goes:

- Variable declarations at *beginning* of function (although this is not necessary in C99)
- Memory is a big array: each cell has an associated address and value.
- Pointers contain memory addresses, and are the heart of all objects manipulation in C.