CS240 Comprehensive Review

Theo Park

3 May 2022

1 Basics

~~C was developed by our lord Turkstra in 1678.~~
Created by Dennis Ritchie in 1969 (nice) - 1973 at Bell Lab, with a desire to make UNIX portable unlike assembly. It's portable, fast, and simple, so screw you Javascript.

2 Compiling and Linking

2.1 Gcc Flags

- *-c* Compile file into object file
- *-g* Debugging symbols
- ***-Wall*** Include ALL Warning
- ***-Werror*** Turn wanings into errors
- ***-O1, -O2, -O3*** Optimize output code
- ***-o filename*** Output to filename
- *-ANSI* Adhere to ANSI std
- *-std=C99* Adhere to C99 std

2.2 Linking

Object file contains binary code, symbol tables, and is a compiled form of a C module. To make it a complete executable, one must link object files, with one of them containing main().

3 File I/O

3.1 Essentials

- FILE *fopen(char *file_name, char *mode);
  Modes are "r", "w", and "a" (append). Returns file ptr on success, NULL on unsuccess, so one must check the return val of fopen().
- int fclose(FILE *file_pointer);
  It does not set the file ptr to NULL, so you have to manually set it to NULL. Return val check isn't necessary in this class.
- **int fprinf(FILE *stream, const char *format, . . .);**

- **int fscanf(FILE *stream, const char *format, . . .);**
- int access(char *file_name, int mode);
  Used to check if file can be accessed in "R_OK", "W_OK", or "F_OK" (check for existence) mode.
- int feof(FILE *file_pointer);
  Returns non-zero if EOF reached.
- int ferror(FILE *file_pointer);
  Returns 0 if error occurs (e.g disk space full).

3.2 Notes with fscanf()

- Utilize %[] (%[0-9A-z] %[^A-z])
- **Field width specifier (e.g %49s %49[A-z]). Always one less than the buffer size to account for NUL terminator.**
- Assigns variables to pointers; use & symbol for non-strings.
- Returns number of successfully read variables; Check for error using the return value.

3.3 Random Access File I/O

- **int ftell(FILE *file_pointer);**
  Returns current offset from the beginning of the file (SEEK_SET) or -1 in case of error.
- **int fseek(FILE *fp, long int offset, int whence);**
  Whence values include
  - SEEK_SET: Offset relative to beginning of the file
  - SEEK_CUR: Offset relative to the current position
  - SEEK_END: Offset relative to the end of the file
- Example of finding how long the file is:

```
1        fseek(fp, 0, SEEK_END);
2        int len = ftell(fp);
3        fseek(fp, 0, SEEK_SET);
```

4 Struct and Typedef

4.1 Some Syntax

- Typedef and struct definition:

```
1        typedef struct my_data {
2            int age;
3        } my_data_t;
```

- Struct definition and declaration:

```
1        struct my_data {
2            int age;
3        } my_var = { 19 };
```

4.2 Declaration vs Definition

Declaration is announcing the properties of var (no memory allocation), definition is allocating storages for a var.

- Declaration:

```
        struct my_data {
            int age;
        };
```

- Definition and initialization:

```
        struct my_data my_var = { 19 };
```

Put pure declaration (struct, func prototype, extern) outside of the func, put definition inside func.

4.3 Initializing/Assigning Elements in Struct

```
    struct my_data {
        int age;
        int random_stuffs[2];
    };
```

Initialization is what you expect.

```
    struct my_data my_var = { 12, { 3, 4 } };
```

Usually, assigning elements are done individually

```
    struct my_data my_var = { 122, { 33, 44 } }; /* Oh no mistakes */
    my_var.age = 12;
    my_var.random_stuffs[0] = 3;
    my_var.random_stuffs[1] = 4;
```

In C99, you can do it all at once with compound literal.

```
    my_var = (struct my_data) { 12, { 3, 4 } };
```

5 Memory

5.1 Array and String

- When a global array is initialized, or a local array is partially initialized, any non-initialized elements are 0.
- **Use strncpy(str1, str2, num), with last arg being sizeof(str1)** (in stack allocated memory settings).
- strcmp(str1, str2) returns 0 if they are the same, positive number if str2 comes before str1, negative if str1 comes before str2.

5.2 Memory Map

Cry.

5.3 Padding

Structure is padded in C when smaller size variable is followed by bigger size variable. Padding of sizeof(bigger_var) - sizeof(smaller_var) will be added in between for a faster access.

```
struct example {
    char a; /* 1 bytes */
    char b; /* 1 bytes */
    int c; /* 4 bytes */
    /* Prev. element adding up to 6, thus 2 bytes of padding */
    struct example *next; /* 8 bytes */
};
```

Size of struct example is 16 bytes (in 64 bit system). Remember that the size of structure should be a multiple of the biggest variable.

5.4 Binary File I/O

- **int fwrite(void *ptr, int size, int num, FILE *fp);**
  Usually, write one item at a time. having target structure as the 1st arg, sizeof(struct) as the second arg, and 1 as the 3rd arg.
- **int fread(void *ptr, int size, int num, FILE *fp);**
  Same thing as fwrite, just reading. Oh right, did you know that when you read long, it's stored in reverse order? Damn like it should totally be on the exam because no one will get it right.
- One more tip: If you want to know the offset of the struct you just read, do

```
        ftell(fp) / sizeof(struct example) - 1;
```

6 Struct Wannabes

6.1 Bitfields

You can create fields within struct that do not contain round number of bits.

```
struct my_bitfields {
    unsigned int sign: 1;
    unsigned int exp: 11;
};
```

Have no clue why this is useful.

6.2 Union

Declare just like struct

```
union my_union {
    int combined_bytes;
    char four_bytes[4];
};
```

If you don't specify, it assumes that you're initializing the first field. C99 has designated initializer.

```
union my_union my_var = { .four_bytes = { a, b, c, d} };
```

6.3 Enum

Attaching a label to a value. You can either assign a value or not. **Not semicolon, use commas**.

```
enum color {
    TEAL,
    TURQUOISE = 4,
};
enum color my_fav_color = TURQUOISE;
enum color my_2nd_fav = TEAL;
printf("My fav color is \%d\n", TURQUOISE); /* Will print out 4 */
```

6.4 Bitwise Operators

I know this doesn't belong here but I hate it and didn't want to give it a new section.

- & | ^ ~
- Shift operator $\ll \gg$
  Every shift left is equivalent to multiplication by two.

```
        y = x << 4 /* Equivalent to y = x * 2^4 */
```

7 Pointer Pointer Pointer!

& Operator is used to determine the address of an element. * operator is used to manipulate the contents at the address of a variable.

7.1 Array and Pointer (They are the samething)

- Arrays are equivalent to pointer (not the other way around), and ptr can be used as arrays.
- Differences are
  - You cannot assign something new to array unlike ptr.

    ```
    ptr = arr; /* allowed */
    arr = ptr; /* NOT allowed */
    ```

  - Array definition allocates memory for every elements in the array, whereas you need to dynamically allocate memory for a var that ptr points to (ptr itself is allocated; 8 bytes).

7.2 Pointer Arithmetic

You can move around the array using arithmetic on ptr.

```
int arr[10] = { 1, 2, 3, 4, 5 };
int *ptr = arr; /* Points to the 1st element in the arr - 1 */
ptr = &arr[1]; /* Now points to the 2nd element in the arr - 2 */
ptr++; /* Points to the 3rd element - 3 */
ptr += 3; /* Points 6th element - 0 - Remember uninitialized elements are
   0 */
```

8 GDB

```
$ gcc -g debugging.c -o debugging # Don't forget -g flag
$ gdb debugging
(gdb) run
(gdb) bt # Backtrace the crash point
(gdb) quit
```