CS240 Comprehensive Review

Theo Park

3 May 2022

1 Basics

~~C was developed by Lord Turkstra in 1678.~~
Created by Dennis Ritchie in 1969 - 1973 at Bell Lab, with a desire to make UNIX portable unlike assembly. It's portable, fast, simple, and still widely used, 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 (O3 will make the file bigger; more in 14.1)
- ***-o filename*** Output to filename
- *-ansi* Adhere to ANSI std
- *-std=c99* Adhere to C99 std

Typical homework gcc commands:

```
$ gcc -Wall -Werror -std=c17 -c hwx.c
$ gcc -Wall -Werror -std=c17 -c hwx_main.c
$ gcc -o hwx_main hwx_main.o hwx.c
```

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 non-zero 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
- Moving to the 3rd integer in the file

```
1    fseek(fp, 2 * sizeof(int), SEET_SET); /* Remember 0 is the 1st int */
```

- 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:

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

- Struct definition and declaration:

```
struct my_data {
    int age;
} 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 Endianness

Big Endianness: Most significant bytes come first.
Little Endianness: Least significant bytes come first.
For an integer 305419896, with hexadecimal 0x12 34 56 78, big endian stores it as **12 34 56 78**, and little endian stores it as **78 56 34 12**.

5.4 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 */
    /* 2 byte padding */
    int c; /* 4 bytes */
    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.

5.5 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,
    RED,
};
enum color my_fav_color = TURQUOISE;
enum color my_2nd_fav = TEAL;
printf("My fav color is \%d\n", TURQUOISE); /* Will print out 4 */
/* TEAL == 0, RED == 5; number keeps increment until it's reinitialize */
```

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 ≪ ≫
  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 to 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
```

9 Linked List and Dynamic Memory Allocation

9.1 $\rightarrow$ Operator

```
struct my_data my_var = { 19 };
struct my_data *my_ptr = &my_var;
printf("%d\n", (*my_ptr).age); /* Will print 19 */
printf("%d\n", my_ptr->age); /* Will also print 19! */
```

Be aware that (*ptr)->age and *ptr->age (which is the same as *(ptr->age)) are different.

9.2 malloc(), calloc(), and free()

malloc(int size) allocates a memory in the heap memory region. free(void *ptr) frees the region allocated in the heap. calloc(int n, int s) allocates n chunks of memory of size s and set all the bytes to 0.

- **Always check return value of malloc (pointer) using assert**.
- **Always set the ptr to NULL after free**.
- Rule of thumb is that you malloc for any non-array pointers.

```c
typedef struct my_data {
    char str1[5];
    char *str2;
    int integer;
} my_data_t;

my_data_t *make(char *my_str1, char *my_str2, int my_integer) {
    my_data_t *my_var = malloc(sizeof(my_data_t));
    assert(my_var != NULL);

    assert(strlen(my_str1) <= 4); /* Arr w/ fixed size; no malloc
       needed, but make sure its length is 4 at max */
    strcpy(my_var->str1, my_str1);
    my_var->str1[4] = '\0';

    my_var->str2 = malloc(strlen(my_str2) + 1); /* Pure ptr,
       malloc strlen + NUL */
    assert(my_var->str2 != NULL);
    strcpy(my_var->str2, my_str2);

    my_var->integer = my_integer; /* Fixed 4 bytes, no malloc
       needed */

    return my_var;
}
```

- **Do not return address of something that is stack allocated!** Only return the ptr to dynamically allocated var.

9.3 Linked List

```c
typedef struct node {
    int val;
    struct node *next;
} nt;
```

- Adding an element to the head

```c
nt *add_head(nt *head, nt *element) {
    /* Treat 2nd element as the new head */
    element->next = head;
    return element;
}
```

- Adding an element to the tail

```c
nt *add_tail(nt *head, nt *element) {
    if (head == NULL) {
        return element;
    }
    nt *head_archive = head;
    /* Traversing the list to find tail */
    while (head->next != NULL) {
        head = head->next;
```

```
        }
        head->next = element;
        return head_archive;
    }
```

- Deleting an entire list

```
    void delete(nt *head) {
        nt *temp = head;
        while (head != NULL) {
            temp = head->next;
            free(head);
            head = temp;
        }
    }
```

9.4 Valgrind

Is useful to identify leaks and errors

```
$ valgrind ./executable
$ valgrind --leak-check=full ./executable
```

10 Doubly Linked List, Pointer to Pointer, and Other Ptr Related

10.1 DLL

```
typedef struct double_node {
    int val;
    struct double_node *prev;
    struct double_node *next;
} dnt;
```

- Prepend

```
    void prepend(dnt *list_element, dnt *element) {
        assert(list_element & element);
        if (list_element->prev != NULL) {
            list_element->prev->next = element;
        }
        element->prev = list_element->prev;
        element->next = list_element;
        list_element->prev = element;
    }
```

- Append

```
    void append(dnt *list_element, dnt *element) {
        assert(list_element & element);
        if (list_element->next != NULL) {
            list_element->next->prev = element;
        }
        element->next = list_element->next;
```

```
            element->prev = list_element;
            list_element->next = element;
        }
```

- Remove an element

```
        void remove(dnt *element) {
            if (element->next != NULL) {
                element->next->prev = element->prev;
            }
            if (element->prev != NULL) {
                element->prev->next = element->next;
            }
            element->prev = NULL;
            element->next = NULL;
            free(element);
            element = NULL;
        }
```

10.2 Ptr 2 Ptr

When passing a pointer to a function, you are passing it by a value, not by reference. Thus, you have to pass a pointer to pointer if you want the changes in the function to be reflected in the function that it was called.

- Creating a DLL node (single ptr version)

```
        dnt *create(int my_val) {
            dnt *new_node = malloc(sizeof(dnt));
            assert(new_node != NULL);
            new_node->val = my_val;
            return new_node;
        }
```

- Creating a DLL node (double ptr version)

```
        void create(int my_val, dnt **make_it_pt_here_plz) {
            dnt *new_node = malloc(sizeof(dnt));
            assert(new_node != NULL);
            new_node->val = my_val;
            *make_it_pt_here_plz = new_node;
        }
```

10.3 Faces of Zero

- 0 as an int
- 0x0 in hexadecimal
- '\0' NUL terminator
- 0.0 float val
- NULL == ((void *)0)

10.4 Function Pointer

Lesser useful example:

```
int multiplication(int val1, int val2) {
    return val1 * val2;
}
int main {
    int (*func_ptr)(int, int) = NULL;
    func_ptr = multiplication;
    int result = func_ptr(1, 2); /* Or (*func_ptr)(1, 2) */
    printf("%d\n", result)
}
```

More useful example; passing it as an arg:

```
int struct_val_multiply(dnt *my_struct, int my_val,
                        int (*multiplication_ptr)(int, int)) {
    return multiplication_ptr((my_struct->val), my_val);
    /* Ik it's a bad example. Just remember the syntax */
}
```

10.5 Recursion

Function calls are stored in stack memory.
Thus iterative implemnentation is often more efficient, resursive implemnentation is often more convenient.

```
int fibonacci(int n) {
    if (n == 0) return 1;
    if (n == 1) return 1;
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

Don't forget to have a base case!

11 Tree

```
typedef struct node {
    int val;
    struct node *left;
    struct node *right;
} tree_node;
```

11.1 Tree Traversal

- **Inorder: L-Node-R; Least to Greatest**.
- Reverse: R-Node-L; Greatest to Least.
- **Prefix: Node-L-R**.
- **Postfix: L-R-Node**.

11.2 Dynamic 2D Array

I got a headache.

12 Type

12.1 First-Class Types

You can assign any first-class types to each other, unlike second-class types where only the same types are compitable.

- void
- char
- short
- int
- float
- long
- double
- long long

12.2 Type Modifiers and Qualifiers

- Integer type (char, short, int, long, long long) can have **signed or unsigned modifier**.
- **Volatile qualifier means that the datum can be modified by something outside of the program**.
- **Const qualifier means that datum cannot be modified**

```
const int *ptr; /* Ptr can be modified, val cannot be modified */
int * const ptr; /* Ptr cannot be modified, val can be modified */
```

12.3 Storage Classes

- **Extern: Signifies that the dataum is definied in some other module**. Use it for an application with multiple C files accessing the same global var, defined it only in one file and use extern in other files.
- Static (Local var.): The datum is initialized only once and retains its value between invocations of the func.
- Static (Global var.): The datum is not visible by other module.

12.4 Cast and Void Type

```
int *i_arr = malloc(sizeof(int) * 2);
char *c_ptr = (char *) i_arr;
int *i_ptr = (int *) c_ptr;
i_ptr[0] = 7;
/* Will this work? */
```

You can complicate casting syntax as well.

```
s = (const struct example * const *) ss;
```

Void types are used when data types are unknown. Its application include callback, which utilizes void pointer to generalize func and accept various func pointers.

```
void callback_func(void (*callback)(void *), void *callback_val) {
    callback(callback_val);
}
```

13 Preprocessor

13.1 #include

Pulls header file into another file (literally copy and paste).

```
#include "something.h" /* Header file from the same directory */
#include <something.h> /* From /usr/include */
```

13.2 #define

```
#define TESTING
int main() {
    int a = 0;
    #ifdef TESTING
    a = 10;
    #else
    scanf("%d", &a);
    #endif
    printf("a: %d\n", a); /* Will print out 10 since TESTING is defined */
}
```

```
#define ABS(x) ( (x) < 0 ? -(x) : (x) )
```

- Macros are replaced by compiler, so it's more efficient.
- Macros can take arguments of any time, so it's more flexible.
- **Make sure to have parenthesis around variables to make them "safer"!!! (More than one digit, etc)**

14 Efficiency Issues and Libraries

14.1 Compiler Efficiency

- -O/-O1 tries to register var and compare multiple lines for efficiency
- -O2 Optimize w/o generating longer code
- -O3 Func inlining, loop unrolling, etc

14.2 Coding Efficiency

- **Use local var if the data is used more than once.**
- Use macros instead of short func.
- Use register var.
- Calculate out of the loop if possible.

14.3 Data Access Efficiency

Reuse allocated memory!

14.4 Static Libraries

- **Static library become a part of executable (needs to be recompiled if changed)**.
- When linking, missing symbols are searched in the libaries. Object files with missing symbols are linked.

```
$ gcc -c file1.c
$ gcc -c file2.c
$ ar -crv libmy_library.a file1.o file2.o # UNIX Specific
$ gcc -o my_program main.c -L. -lmy_library # -L. means search for . dir
```

14.5 Dynamic Libraries

- **Dynamic libraries are loaded on the startup and runtime; no recompile needed**

```
$ gcc -c -fPIC file1.c
$ gcc -c -fPIC file2.c
$ gcc file1.o file2.o -shared -o libmy_library.so
$ gcc -o my_program main.c -L. -lmy_library # -L. means search for . dir
```

14.6 Bottom Line

- Library is efficient; it's faster to link one library with 1000 object files than 1000 object files.
- Declare one data struct per file
- Don't use unnecessary #include
- #include only once (Will create "duplicate declaration" error). **Utilize Include Guard**.

```
#ifndef HEADER_H /* Check if this macro has been created already */
#define HEADER_H /* If not, define it */
/* Header file contents... */
#endif /* HEADER_H - Macro can be named to anything */
```

15 "Random" Number

Computer cannot generate random-number. Only ones that seem random.

```
#include <stdlib.h>
#include <time.h>
int main() {
    srandom(time(0)); /* Feed the seed */
    int x = 3;
    int y = 5;
    int rand = x + random() % (y - x + 1); /* Rand num b/w 3 - 5 */
}
```

16 Graphical Programming

-