Systems Programming

UT Knoxville, Spring 2023, COSC 360

Dr. James S. Plank, Alex Zhang

April 23, 2023

Contents

Preface

These notes attempt to give a concise overview of **Systems Programming** course at the University of Tennessee. The contents of these notes come from Dr. James Plank's online COSC 360 notes[1] as well as select comments from his in-person lectures.

[1] `https://web.eecs.utk.edu/~jplank/plank/classes/cs360/lecture_notes.html`

1

Moving from C++ to C

In moving from C++ to C, we lose a lot of nice features: standard template library (STL), `iostream`, objects, methods, and operator overloading to name a few. We can think of C as a subset of C++ in terms of features. Only in rare circumstances will C code not compile in C++.

1.1 Basic Terminal I/O

In C, `stdio.h` provides `printf()` for terminal output, and `fgets()` and `scanf()` for terminal input.

Code Snippet 1.1.1 ▶ Hello World in C

```c
#include <stdio.h>

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

    return 0;
}
```

1.2 Scalar Types

Definition 1.2.1 ▶ Scalar Type

A ***scalar type*** is a type that stores a single value.

In C, there are seven default scalar types:

- `char` — 1 byte
- `short` — 2 bytes

- `int` — 4 bytes
- `long` — 4 or 8 bytes (depending on machine)
- `float` — 4 bytes
- `double` — 8 bytes
- `pointer` — 4 or 8 bytes (depending on machine)

Unlike C++, `bool` is not a default scalar type. As of C99, `bool` is defined in `stdbool.h`. Despite only storing 1 bit of info, `bool` occupies 1 byte of memory.

Scalar variables can be declared in one of three scopes:

- outside of all functions as **global** variables
- inside a function implementation as a **local** variable
- inside a function prototype as a **procedure** parameter

Global variables are allocated to memory once the program starts and only deallocates once the program terminates. **Local** variables are only allocated once the program reaches its function and are deallocated when the function ends.

There are no reference variables (`&`) in C. Function parameters are **always** copied.

1.3 Aggregate Types

Definition 1.3.1 ▶ Aggregate Type

An ***aggregate type*** stores one or more values.

In C, there are only two aggregate types: Array and Structure.

Definition 1.3.2 ▶ Array

An ***array*** is a contiguous piece of memory that stores multiple variables of the same scalar type.

In most programming languages (including C), arrays are represented by the memory address of its first element. Since the array is contiguous, C can easily calculate the memory address of all other elements.

In C, arrays can only be declared either globally or locally. When we pass an array to a function, we are only passing a pointer to its first element.

Definition 1.3.3 ▶ Structure

A ***structure*** or ***struct*** is a contiguous piece of memory that stores multiple variables which can be different scalar types.

In C++, we can use structs and classes interchangeably—the only difference being that, by default, structs make members `public` while classes makes members `private`. In contrast, C does not have classes, and structs in C don't have the following capabilities:

- No **access modifiers** like `public`, `private`, or `protected`
- No **constructors** nor **destructors**
- No **method functions**

In addition, C structs do not create a type by default. We have to explicitly create one using the `typedef` keyword.

Member variables of a struct get padded to be aligned to ensure each variable is aligned to 4-bytes **and** consecutive memory is aligned to 4-bytes.

The assignment operator (=) works as intended with structs but not so with arrays. If we have `int arr1[20]` and `int arr2[20]`, we cannot simply write `arr1 = arr2;`. Instead, we have to iterate through the array and copy each `int` one at a time.

Generally, we always copy a specific number of bytes using the assignment operator. The only exception is when we copy a struct that has an array as a member. Assignment operators for structs copies **all data**.

Code Snippet 1.3.4 ▶ Assignment Operator for Aggregate Types

```c
typedef struct {
    int arr[10];
} MyStruct;

int main() {
    MyStruct ms1, ms2;

    /* will compile; copies 40 bytes from ms1 to ms2 */
    ms2 = ms1;

    /* won't compile (can't use assignment operator on arrays)
    ms2.arr = ms1.arr; */

    /* right way */
    for (int i = 0; i < 10; ++i) {
        ms1.arr[i] = ms2.arr[i];
    }

    return 0;
}
```

Similarly, if a function an argument that is a struct with a member array, it will **copy** the struct as a procedure parameter.

Code Snippet 1.3.5 ▶ Confusing Struct Argument

```c
typedef struct {
    int arr[10];
} MyStruct;

void my_func(MyStruct ms) { ms.arr[9] = -1; }

int main() {
    MyStruct ms;
    ms.arr[9] = 10;
    my_func(ms);

    return 0;
}
```

`s.arr[9]` will still be 10! That's because `my_func()` takes a `MyStruct` parameter, creating a copy of `MyStruct` for use in the function. Hence, we are only changing the copied `MyStruct` which gets deleted at the end of the function. In this case, `my_func()` effectively does nothing.

1.4 Memory and `malloc()`

`malloc()` returns a pointer to a piece of continuous memory allocated only to our program. When we call `malloc()`, we specify a certain number of bytes which is then allocated to the program by the operating system. If the operating system can't allocate that much memory, `malloc` returns `NULL`. To prevent potential bus errors, `malloc()` only returns pointers aligned to 8 bytes.

All `malloc`'ed memory is deallocated either by calling `free()` on the pointer or when the program terminates. `free()` is mostly useful to deallocate data once it's no longer being used. Generally, there is no reason to call `free()` right before the program terminates.

1.5 Strings

Definition 1.5.1 ▸ String, Null Character

In C and C++, a string is a contiguous sequence of characters whose last character is the null character (ASCII value 0).

While C++ has `std::string` which handles the memory of the characters for us, we don't have such luxury in C. Instead, we have to explicitly use an array of `char`, handling it just as we would any other array.

When using a function like `printf("%s", str)`, we are only passing a pointer to the first character of `str`. Then, `printf` iteratively prints each byte in its ASCII representation, stopping once it reaches the null character.

1.6 Libfdr

To ease the transition between C++ and C, Dr. Plank has created a library for formatting input doubly-linked lists, and red-black trees (hence, fdr).[1]

- The ***fields*** library helps parse input from stdin and files
- The ***dllist*** library provides doubly-linked lists
- The ***JRB*** library provides red-black trees, similar to `std::multimap`

[1]Documentation is available on `https://web.eecs.utk.edu/~jplank/plank/classes/cs360/lecture_notes.html`.

File System

Definition 2.0.1 ▸ File System

A ***file system*** is a hierarchy of files and directories.

In the UNIX file system, every user has a home directory, aliased by "~".

Definition 2.0.2 ▸ Signal

A ***signal*** is an interruption in a running program.

Definition 2.0.3 ▸ System Call

System calls allow programs to directly interface with the operating system. In C, system calls appear just as regular procedure calls.

Whereas regular procedures (functions) create a stack frame, system calls incur a signal to the program, interrupting it. Hence, they are very expensive and should be used sparingly. By design, system calls do not produce segmentation faults, allowing for some potential memory errors.

In C, there are five system calls defined in `fcntl.h` for file I/O:

- `open()` – opens a handle to a file for I/O; specifies read and/or write permission
- `close()` – close a handle to a file
- `read()` – reads a specified number of bytes; return number of bytes read
- `write()` – writes specified bytes to the file
- `lseek()` – repositions the file pointer

We generally avoid using these system calls and instead use the `stdio` equivalents: `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fseek()`. For example, `fread()` still calls `read()`, but it will store read bytes into a buffer to sparingly call `read()`.

Every file has three distinct parts:

1. The actual data of the file
2. The ***metadata*** (information about the file)
3. The directory entry

In UNIX, the file has a directory entry and an associated ***inode*** that stores the metadata and the location of the actual bytes of the file.

3 Memory

We can think of memory as just a really long array. In 32-bit systems, we can have up to 2^{32} bytes of memory available to us.

For each process, UNIX declares four regions of memory:

1. ***Text:*** your executable instructions (assembler compiled down to bytes), typically read-only
2. ***Global variables:*** these are determined at compile time
    - "data" – initialized globals
    - "BSS" – uninitalized globals
3. ***Heap:*** This memory is retrieved by `sbrk()` or `mmap()` – typically, we see `malloc()`
4. ***Stack:*** Local variables, function call overhead, etc.

more

This is set up by the operating system in conjunction with the hardware.

Definition 3.0.1 ▸ Memory Page

Memory is partitioned into fixed-size intervals called ***pages***. These are typically 4 or 8 kibibytes.

Whenever we reference some memory, the OS looks at the ***page table*** to find the page it is stored in. If it is an invalid page, the system incurs a segmentation fault.

C has special variables that mark the end of the memory segments for each region (not POSIX standard).

Flush before fork

4

Threads

Why do we care about threads?

1. They excel at handling asynchronous computations/instructions
2. Running threads in parallel offers a performance boost

We will cover POSIX threads, which are standardized among all UNIX operating systems. The general idea is that we have two procedures that look something like:

1. `TCB thread_fork(proc, arg)` that creates a thread which shares the same address space but has its own stack. It runs `proc(arg)` on its own stack.
2. `int thread_join(TCB)` that waits for the thread `TCB` to exit, and returns what the thread returned from `proc`. Although a thread may finish running a procedure, its stack will persist until something calls `pthread_join` on its thread.

In C, these procedures are part of `pthread.h`, named `pthread_create()` and `pthread_join()`.

A ***preemptive*** thread system means that either:

- each thread runs on separate CPUs, or
- the thread system makes each thread run "simultaneously".

Most modern thread systems are preemptive.

Definition 4.0.1 ▶ Race Condition

A ***race condition*** is present in code that relies on separate threads running deterministically, but really they run non-deterministically.

To protect against race conditions, we can specify a certain procedure to only run one thread at a time. In C, this is done using a `mutex`.

Index