

---

# COMP2017 / COMP9017

---

# Tutorial 5

---

## File IO, Function Pointers and Signals

### File IO and Function Pointers

As per the unix philosophy, *"Everything is a file"*, this means that we can typically get majority of our information from unix processes, pipes and memory mapped files.

Processes running inside UNIX have a table set aside for keeping track of what files they have open, called the file descriptor table. The entries within the table typically are a simple numerical identifier that is known as a file descriptor. The initial values, 0, 1, 2, represent standard in, standard out, and standard error. Any other subsequently opened files are stored at the next largest value. When a process opens a file (or any other file-like object), the next free file descriptor is used. For example, if the numbers 0, 1, 2, 4, 6 were in use, the next open file would take the value 3.

### **stdio.h** functions and macros

By including the `stdio.h` header we have access to the `fopen`, `fwrite`, `fread`, etc. However, these functions are library functions for system the `open`, `write`, `read`, etc system calls. These calls utilise a `FILE*` object that contains platform specific information.

Using the `man` command, open up `stdio` man page like so: `man 3 stdio`

You can get a quick description of the function and how it is used.

### **fcntl.h** and **unistd.h** functions and macros

Typically we do not need to look further than the `stdio` functions however when interacting with certain devices, you may want to have full control over the io device and specify the IO modes such as block and nonblocking, buffering modes, etc. To, open a file using a sys-call we can use the `open()` function which will return an integer (file descriptor) that corresponds to the handle attached to the process.

In relation to file descriptors, we have access to `read()` and `write()` system calls that operate on the file descriptors. These the the most primitive input/output functions that are commonly exposed on unix systems.

## Write, Read and Flush

`fwrite` and `fread` are an abstraction that utilises binary stream input and output functions. Once a `FILE*` object has been initialised, you can perform file stream operations such as `fread` and `fwrite`.

These functions operate in a similar pattern where they require you utilise a buffer for copying data and specifying the size of the data and number of elements. This can be used in conjunction with any kind of pointer or array of any type assuming the bounds of the buffer supplied are not violated. You will need to acknowledge that you are reading and writing binary data and the structures you utilise will be interpreted based on your assumptions.

Usage example:

```
//Assume file has been opened for reading
...
int numbers[24];
size_t bytes_read = fread(numbers, sizeof(int), 24, f);
```

`fread` will return how many bytes have been read and will move the file cursor (where we are currently reading at) forward `'sizeof(int)*24'` bytes.

```
int numbers[6] = {1, 2, 3, 4, 5, 6};
size_t bytes_written = fwrite(numbers, sizeof(int), 6, f);
```

Since it is file stream this may mean any contents you have written using `'fwrite'` may not have been flushed or written to disk. To immediately write to disk, you

```
//If you want data immediately flushed
fflush(f);
//if you have finished writing and want to close
fclose(f);
```

- For functions related to text formatting and printing, look into file stream counterparts for `gets`, `printf`, `scanf` and `getc`.

## Blocking and Non-blocking IO

These two dictate how a function will behave when executing them in an IO context. Blocking functions will wait until data in the buffer has been flushed to it before proceeding onto the next instruction.

However, non-blocking will not wait for the data to be entered into the buffer and will immediately check the buffer and grab the data it can. This can seem counter-intuitive because someone may ask the question "Why would we want this if we can get incomplete data?". Typically non-blocking is used in conjunction to signaling and interrupts, for the signal to tell the program that the buffer is ready and for the program to read the buffer. Commonly referred to as event driven.

## Question 1: Writing to standard out without printf

Given that processes have file descriptors for `stdin`, `stdout` and `stderr`, use the system calls to write "Hello World" using the `stdout` file descriptor. Use `write` for this question.

## Question 2: Working with a stream

We will provide a program that will create a file that the process will stream to. Your program will need to receive those packets and decode them. With the given data data layout:

Structure

```
id: unsigned byte
locked: unsigned byte
buttons: 2 bytes
    x: unsigned bit
    y: unsigned bit
    z: unsigned bit
    w: unsigned bit
    a: unsigned bit
    b: unsigned bit
    c: unsigned bit
    d: unsigned bit
    l: unsigned bit
    r: unsigned bit
    st: unsigned bit
    sel: unsigned bit
analog: 8 bytes
    left: float
    right: float
```

After receiving the packet, the reading process needs to output what buttons have been pressed and the current state of the analog sticks. If a button has been released it will need to show this state information between one read and another. If a button has maintained its current value for more than 3 packets, it is considered to be a Hold.

Example:

```
PKT0: Analog Left: 0.0000, Analog Right: 0.0000
PKT1: X: Pressed, Analog Left: 0.0000, Analog Right: 0.0000
PKT2: X: Released, Analog Left: 0.0000, Analog Right: 0.0000
PKT3: X: Pressed, Analog Left: 0.0000, Analog Right: 0.0000
PKT4: X: Pressed, Analog Left: 0.2400, Analog Right: 0.0000
PKT5: X: Pressed, Analog Left: 0.2400, Analog Right: 0.0000
PKT6: X: Hold, Analog Left: 0.0000, Analog Right: 0.0000
...
```

## Function pointers

Function pointers are another type within C that allows for variables to hold an address to a function. These allow you to have a pointer variable refer to a function and execute it. The function signature forms part of the type information for the variable which allows the compiler to confirm that an appropriate function with the same signature has been assigned.

Example:

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int (*a)(int, int) = &add;
    //We have set the function pointer a to the address of add
    int result = a(1, 3);
    printf("Result of add: %d\n", result);
}
```

It is common to utilise function pointers as a parameter to a function for generalising certain operations on a collection and to act as a callback. Function pointers can exist within struct fields, although this is considered to be bad practice and an attempt to employ a simplistic object oriented pattern in C.

Example:

```
int calc(int a, int b, int (*c)(int, int)) {
    return c(a, b);
}

int add(int a, int b) {
    return a + b;
}

int main() {
    //We passed add to the calc function,
    //if we have other functions that have a similar signature

    int result = calc(1, 3, &add);

    printf("Result of add: %d\n", result);
}
```

- What would the pros and cons be of using typedef on a function pointer?
- Why would we pass a function pointer to another function?
- Has there been a programming language where you have used something similar?

## Question 3: Sorting and outputting

Extend the following code to implement bubble sort. Bubble sort is an inefficient but simple to implement sorting algorithm. It steps through the array one element at a time and compares each pair of adjacent elements.

The algorithm swaps the pair if they are in the wrong order. This is repeated until the array is sorted.

```
int element_cmp(void* a, void* b);

void bubble_sort(void* elements, size_t n_elements,
                 size_t size_element, int (*cmp)(void*, void*));
```

Output the contents of the sorted list afterwards to verify the elements have been sorted.

The following is the pseudo code for bubble sort.

```
def bubblesort(list, length):
    let i = 0
    while i < length:
        let j = 1
        while j < length - 1:
            if compare(list[ij, list[j+1] > 0):
                swap(list[j], list[j + 1])
            j = j + 1
        i = i + 1
    end def
```

## Question 4: Tiny delegator

You are given a list of string instructions to parse. You are required to build a list of these instructions that map to a basic function pointer and value. This will represent a very simple program that calculates

After the instructions have been created and added to the list, your program will execute list of functions.

Specification:

ADD - Addition  
SUB - Subtraction  
MUL - Multiple  
DIV - Divide

Your program should implement these 4 operations as well as parse each line, your program only has to deal with fixed 32 bit integer numbers (not floating point).

The instructions will be provided in this form:

```
ADD 9 10
SUB % 10
MUL 3 %
DIV % 1
```

The % symbol represents the value from the previous operation, and will need to be referred to by that location.

## Signals

When you press `ctrl+c` or `ctrl+z`, your shell sends the signal `SIGINT` or `SIGSTOP` respectively to the current fore-ground program. Signals are a way to communicate with programs that are currently running. You can send a signal to a program if you know its process ID using the misleadingly named `kill` command.

Give a process we want to send a signal to has the process id of 255, we can send a signal using the `kill` command.

Format

```
kill -<signal type> <pid>
#example
kill -SIGKILL 255
```

Other useful signals to know are `SIGTERM`, which asks the process to terminate, and `SIGKILL` which forces the process to terminate. All signals apart from `SIGKILL` and `SIGSTOP` can be handled by the process by defining a signal handler.

## Custom signal handling

We can setup custom signal handlers to our programs that allow us to setup functions to trigger for certain signals. On the event of a signal being received by the process, it will map the signal to a function, this will interrupt the current execution and start executing the current function binding for that signal.

Example:

```
void interrupt_handler(int sig) {
    printf("I was interrupted!\n");
}

int main() {
    signal(SIGINT, interrupt_handler);
    ... rest of code
}
```

However, the `signal` function has been deprecated in favor of `sigaction` which setup is a little less obvious.

```
void sigint_handler(int signo, siginfo_t* sinfo, void* context) {
    printf("I was interrupted\n");
}

int main() {
    struct sigaction sig;
    memset(&sig, 0, sizeof(struct sigaction));
    sig.sa_sigaction = sigint_handler; //SETS Handler
    sig.sa_flags = SA_SIGINFO;

    if(sigaction(SIGINT, &sig, NULL) == -1) {
        perror("sigaction failed");
        return 1;
    }
}
```

You can get information about `sigaction` function and types from the man pages.

## Question 5: A custom signal handler

The C standard library provides the `signal.h` header for handling signals.

Compile and run the following program:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdbool.h>

int main(void) {

    signal(SIGINT, SIG_IGN);

    while (true) {
        sleep(1);
    }

    return 0;
}
```

Attempt to kill the program via `SIGINT`. Find a way to kill the program.

*Hint:* You may want to use the `kill` command from another terminal. Alternatively, you could kill the program within the same terminal.

*Can you write a signal handler such that it will miss signals?*



## Question 6: Tell me the time!

After discovering that you can assign a function to a signal handler. Write a program that will wait for signal (specifically `SIGUSR`) to be sent to the process and output the current time the signal was received. Your program must *gracefully* exit when it receives `SIGINT`.

```
$ ./handle
Thu Apr 15 12:22:34 2018
Thu Apr 15 12:23:12 2018
Thu Apr 15 12:25:41 2018
^C
Shutting down
```

*Hint:* You will need to use the `time.h` header and the utilise the `time_t` and `time` function.

## Question 7: Assignment 2

If you have finished all tutorial questions, please feels free to utilize the rest of the tutorial time to complete Assignment 2.

[Information/Resource Megathread](#)

## Question 8: Quiz

In the last 10 minutes of the tutorial, you will be required to undertake a quiz that will test your knowledge of the previous week's content. It will be a simple assessment that corresponds to 1% of your final grade. You can access the quiz on Ed this week.