FIT2100 Laboratory #2
More on C Programming:
Running I/O System Calls in C
&
Pointers and Dynamic Structures
Week 4 Semester 2 2024

July 24, 2024

Revision Status:

Updated by Dr. Charith Jayasekara, July 2024.

Acknowledgment:
The majority of the content presented in this lab was adapted from the courseware of FIT3042 prepared by Robyn McNamara.

Contents

1 Background **4**

2 Pre-class Reading **4**

2.1 Statements and Blocks 4

2.2 Selective Structures 5

2.2.1 If and If-Else Statements 5

2.2.2 Switch Statement 6

2.3 Iterative Structures 7

2.3.1 For Loop 7

2.3.2 While and Do-While Loops 7

2.4 Pointers in C 8

2.4.1 Pointers and Function Arguments 9

2.4.2 Pointers and Arrays 10

2.4.3 Pointers and Strings 11

2.5 Structs in C 12

2.6 Dynamic Data Structures 14

2.6.1 Memory Allocation and Deallocation 15

2.6.2 Multi-dimensional Arrays 16

2.7 Managing C programs 17

2.7.1 Organising your source files 17

2.7.2 Building complex C programs 18

2.7.3 The `makefile` format 19
2.7.4 Running the `make` command 20
2.7.5 Defining macros in `makefile` 21
2.8 I/O System Calls in C 23
2.8.1 File descriptors 23
2.8.2 Opening and creating files 24
2.8.3 Reading and writing files 25
2.8.4 Using `write()` with strings 26
2.8.5 Moving around files 27
2.9 Debugging C programs 28
2.9.1 Handling errors with defensive programming 29
2.9.2 The `gdb` debugger 29
2.9.3 Dealing with the program bugs 31

3 Assessed Tasks 34

3.1 Task 1 (25%) 34
3.2 Task 2 (25%) 34
3.3 Task 3 (25%) 34
3.4 Task 4 (25%) 35
3.5 Wrapping Up 35

1 Background

In this second lab, we will continue to explore the C programming language.

We will first discuss the various control flow structures used in C; followed by the advanced data structures supported by C, which include pointers, user-defined structures (`struct`), as well as dynamic structures.

Then, you will learn how to manage complex C programs. Finally, you will learn how to perform low-level I/O systems calls in C.

Before you attempt any of the tasks in this laboratory, create a folder named `LAB02` under the FIT2100 folder (`~/Documents/FIT2100`). Save all your source files under this `LAB02` folder.

Before attending the lab, you should:

- Complete your pre-class readings (Section 2)
- Attempt the assessed tasks (Section 3)

2 Pre-class Reading

2.1 Statements and Blocks

Statements Any valid expression terminated with a semicolon (`;`) is regarded as a statement in C.

```
1  x = y + z;
2  scanf("%d", &number);
3  printf("%s", name);
```

Statement blocks Multiple statements (including declarations) can be grouped together using braces (`{}`) to form compound statements or statement blocks. We have seen that the statements of a function (i.e. the function body) are surrounded by a pair of braces.

Statement blocks are used in various control flow structures in C which determine the order in which statements are executed when certain conditions are given.

2.2 Selective Structures

2.2.1 If and If-Else Statements

Both `if` and `if-else` statements are supported in C with the following syntax:

```c
if (x == y)  /* double equal sign */
{
    printf("equal");
}
```

```c
if (x == y)  /* double equal sign */
{
    printf("equal");
}
else
{
    printf("not equal");
}
```

```c
if (x < y)
{
    if (x < z)
        min = x;
    else
        min = z;
}
else
{
    if (y < z)
        min = y;
    else
        min = z;
}
```

Note: The parentheses around the expression (condition) are required; unlike in other programming languages (such as Python) which can be omitted.

Alternatively, the condition expression can be used as a "shorthand" for non-nested `if-else` statements.

```c
printf("%s", (x == y) ? "equal" : "not equal");
```

Try using `if-else` statements yourself:

Write a C program that converts the 24-hour time format into the 12-hour time format. The program should first prompt for a time in the 24-hour format, and then display the equivalent 12-hour format. For instance, if the input is `13:15`, the program should print `1:15 PM`.

2.2.2 Switch Statement

C provides the `switch` statement to handle multiple decisions based around conditions expressed as integer values (`int`) or character constants (`char`).

```
/* day is defined as an integer int */

switch (day)
{
    case 1:
        printf("MONDAY");
        break;
    case 2:
        printf("TUESDAY");
        break;
    case 3:
        printf("WEDNESDAY");
        break;
    case 4:
        printf("THURSDAY");
        break;
    case 5:
        printf("FRIDAY");
        break;
    case 6:
        printf("SATURDAY");
        break;
    case 7:
        printf("SUNDAY");
        break;
    default:
        printf("unknown");    /* for integers not in range of 1-7 */
        break;
}

```

Note: A `break` statement is required at the end of the statement block for each case, which will cause an immediate exit from the `switch` statement.

Try using `switch` statements yourself:

Write a C program that reads a two-digit number and presents the number in English words. The program should make use of `switch` statements to select the corresponding English word associated with each digit of the number. For instance, the input of 68 should be printed as "sixty eight" .

2.3 Iterative Structures

2.3.1 For Loop

The `for` loops in C have very similar structures in Java. The header of a `for` loop consists of three components: the *initialisation*, the *test condition*, and the *increment or decrement step*.

You can omit any of the three components, given the semicolons are kept. However, omitting the condition will result in an "infinite" loop (as this is considered a permanent `true` condition).

```
for (initialisation; condition; step)
{
    /* statements */
}
```

```
#define MAX 10

int number[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

for (int i = 0; i < MAX;  i += 2)
{
    printf("%d\n",  number[i]);     /* 1, 3, 5, 7 and 9 are printed */
}
```

Note: C does not support the convenient way for iterating through the elements of a sequence or a collection (such as the `for-each` loop in Java or the `for-in` loop in Python).

2.3.2 While and Do-While Loops

Both `while` and `do-while` loops are available in C.

```
while (condition)
{
    /* statements */
}
```

```
do
{
    /* statements */
} while (condition);    /* a semicolon is required here */
```

The test condition (expression) in a `while` loop is evaluated before each iteration; whereas for a `do-while` loop, the test condition is evaluated at the end of each iteration.

2.4 Pointers in C

Pointers store addresses in memory. A pointer may contain the address of another variable (as a "reference" to another value). When assigning the address of a variable to a pointer, the pointer is said to "point to" that variable.

Syntactically, C uses the asterisk (*) to indicate or declare a pointer. The pointer type is defined based on the type of the variable that it points to. In the example below, `iptr` is a pointer which refers to a variable of type `int`.

The `&` operator is used to set a pointer to point to specific variable, by assigning the address of that variable to the pointer. To obtain the value of the variable that a pointer refers to (i.e. de-referencing), the * operator is used.[1]

```
/* pointers.c */
  int *iptr;                    /* iptr is a pointer to int */
  int x = 3;

  iptr = &x;                    /* iptr points to x */
  printf("%d\n", *iptr);        /* 3 is printed */
```

Note: A pointer can only be assigned to the type of the variable that it points to (with one exception for the pointer type of `void *`). A pointer can also be assigned with the special value `NULL` when it is not pointing at anything.

Changing pointers When changing the value of the variable that a pointer (`iptr`) is pointing at, that also changes the value of `*ptr`; but not the pointer itself (i.e. it is still having the same address of the variable that it points to).

```
/* pointers.c */
  ++x;                          /* x = 4 and *iptr = 4 */
  ++(*iptr);                    /* x = 5 and *iptr = 5 */

  printf("%d\n", x);            /* 5 is printed */
  printf("%d\n", *iptr);        /* 5 is printed */
```

To change what a pointer points to, we can simple re-assign it with a different variable (or a new memory address) to it.

```
/* pointers.c */
  int y = 10;
  iptr = &y;                    /* iptr points to y now */
  printf("%d\n", *iptr);        /* 10 is printed */
```

[1] See `pointers.c` for the full source code.

Two (or more) pointers can be made to refer to the same variable (i.e. the value of each of these pointers is the same which is the address of that variable.)

```
/* pointers.c */
  int *iptr1, *iptr2;           /* iptr1 and iptr2 are pointers to int */
  int z = 3;

  iptr1 = &z;                   /* iptr1 points to x */
  iptr2 = iptr1;                /* iptr2 points to where iptr1 points */

  printf("%d\n", *iptr1);       /* 3 is printed */
  printf("%d\n", *iptr2);       /* 3 is printed */
```

Void pointers The special type of pointer with the `void` type allows *conversion* between different types. However, it must be used with care since no type checking is performed.

```
/* pointers.c */
  int *int_ptr;                 /* int_ptr is a pointer to an integer*/
  double *dbl_ptr;              /* dbl_ptr is a pointer to a double */
  void *v_ptr;                  /* v_ptr is a pointer to any type */

  int_ptr = dbl_ptr;            /* illegal */
  v_ptr = (void *) int_ptr;     /* legal */
  dbl_ptr = (double *) v_ptr;   /* legal */
```

2.4.1 Pointers and Function Arguments

In C, functions arguments are always *passed by value*. With this, it is impossible for the called function to alter the original variables in the calling function.

The approach known as *passed by reference* can be used to pass pointer arguments to the values to be modified in the called function. As we have seen, pointers refer to the actual addresses of the variables – hence, a pointer can directly access the variable that it points to and modify the content of that variable.

```
/* pass_by_val_and_ref.c */

/* passing arguments by value */
void swap_val(int x, int y)
{
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

```c
/* passing arguments by reference */
void swap_ref(int *xptr, int *yptr)
{
  int temp;

  temp = *xptr;
  *xptr = *yptr;
  *yptr = temp;
}

int main()
{
  int a = 1, b = 2;
  swap_val(a, b);              /* after swap1 was executed, a = 1, b = 2 */
  swap_ref(&a, &b);            /* after swap2 was executed, a = 2, b = 1 */
  return 0;
}
```

2.4.2 Pointers and Arrays

Recall that, as like other programming languages (such as Python and Java), C uses the [] operator to index arrays. This is in fact a "shorthand" of using pointers.

```c
/* pointers_and_arrays.c */

#include <stdio.h>

int main(void)
{
  int i;    /* counter used in loops */

  /* create an array initalised with 5 elements */
  int array[5] = {11, 3, 6, -1, 8};

  /* access array using subscript notation */
  for (i = 0; i < 5; i++)
      printf("a[%d] = %d\n", i, array[i]);

  /* access array using pointer notation */
  for (i = 0; i < 5; i++)
      printf("*(a + %d) = %d\n", i, *(array + i));

  return 0;
}
```

To refer to the `i`-th element of an array, `a[i]` is equivalent to `*(a + i)`, since the name of an array can be regarded as a pointer (i.e. indicating the starting address of the array and also its first element).

Note: Arithmetic operations — addition and subtraction — can be applied on pointers just like any other basic data types.

2.4.3 Pointers and Strings

Recall that C has no built-in string type like Java and Python. In C, a string is in fact an array of `chars` terminated by a '\0' (`null`) character. Likewise, we could also define a string using a pointer.

```c
/* pointers_and_strings.c */
  char array_str[] = "FIT2100";      /* an array of characters */
  char *ptr_str = "FIT2100";         /* a pointer to a string literal */

  printf("%s\n", array_str);
  printf("%s\n", ptr_str);
```

The `<string.h>` library contains many common functions for string manipulation. Many of these functions often manipulate strings using pointers of type (`char *`).

It is the *programmers' responsibility* to ensure sufficient memory is allocated and available for storing whatever the strings will need to store, given that the string functions generally do not perform checking on the memory space.

Note: Each string must be terminated with '\0'; otherwise the string functions may fail (and your program would crash unexpectedly).

String Function	Description
`char * strcpy(char *dst, char *src)`	copy from `src` to `dst`
`char * strcat(char *str, char *new)`	append the string `new` to `str`
`char * strchr(char *str, int chr)`	locate the character `chr` in `str`
`int strcmp(char *str1, char *str2)`	compare `str1` and `str2`
`size_t strlen(char *str)`	return the length of `str` (but not including '\0')

The command-line arguments We have seen that the `main` function often accepts two arguments from the command line: `int argc` and `char *argv[]`. The second argument is in fact an array of character strings or `char` pointers of variable length.

Given that `argv` is a pointer to an array of pointers, we can access each of the argument strings by using the pointer rather than the array index.

```c
/* argc_argv.c*/

#include <stdio.h>

int main(int argc, char *argv[])
{
    while (--argc > 0)
    {
        printf("%s%s", *(++argv), (argc > 1) ? " ": "");
    }
    printf("\n");
    return 0;
}
```

Note: (++argv) causes the pointer to point at `argv[1]` instead of `argv[0]` in the first iteration, since the first argument is in fact the program (command) name.

2.5 Structs in C

Often times, if we need to perform complex computation, we will have to define our own new data structures to make your program more organised and manageable.

We have seen that data of the same type can be organised as arrays. To handle complicated data, C provides a data structure known as `struct` (structures) to group data of different types together under a single name for easy handling.

Structs in C are different to classes in Python or Java with respect to the following:

- No functions or methods are defined in structs;
- Differences in declaration syntax;
- The arrow (->) operator is used for accessing through pointers;
- The `new` operator is not used during instantiation.

Declaring a struct The keyword `struct` is used for declaring a new structure. The structure tag (following the keyword `struct`) is optional; it can however be used as a "shorthand" when creating variables based on this structure.

```
/* structs.c */

struct Point
{
    int x;
    int y;
};
```

Using a struct To create variables of a specific `struct` type, we can do the following if the structure tag is given during declaration.

Each member (field) defined in a `struct` for each variable is accessed using the dot (`.`) notation.

```
/* structs.c */

  point1.x = 0;
  point1.y = 0;

  point2.x = 1;
  point2.y = 1;

  printf("%d %d\n", point1.x, point1.y);  /* point1.x = 0; point1.y = 0 */
  printf("%d %d\n", point2.x, point2.y);  /* point2.x = 1; point2.y = 1 */
```

To access members of a struct via pointers, the arrow (`->`) operator is used.

```
/* structs.c */

  struct Point point3;
  struct Point *ptr;    /* ptr is a pointer to struct Point */

  point3.x = 0;
  point3.y = 0;

  ptr = &point3;

  printf("%d %d\n", point3.x, point3.y);  /* point1.x = 0; point1.y = 0 */
  printf("%d %d\n", ptr->x, ptr->y);      /* ptr->x = 0; ptr->y = 0 */
```

Define a struct type More conveniently, we can define our own type name using the keyword `typedef`. The name of the `struct` type must come at the end, but not after the `struct` keyword. The common practice is to include `typedef` statements in the `.h` header file.

```
/* point.h */

typedef struct
{
  int x;
  int y;
} Point;    /* a semicolon is required here */
```

In your `.c` file, you can then include the `.h` header file and use this `struct` as follows[2]:

```
  Point point1, point2; /* don't need the struct keyword here this time */
```

Note: We can also give *aliases* to built-in data types by using `typedef`. By performing this does not actually make the C compiler produce different code; however it can make your code more readable.

```
  typedef unsigned long int size_t;
```

2.6 Dynamic Data Structures

Suppose that you have declared a `struct` of type `Name` which contains pointers:

```
/* name.h */

typedef struct
{
  char *first_name;
  char *middle_name;
  char *last_name;
} Name;
```

By just declaring an instance (creating a variable) of `Name`, does not mean that memory space is (automatically) allocated to store each of the three strings (`char` pointers). You will have to manually request memory space from the operating system (OS).

[2] See `point_include.c` for the full source code.

2.6.1 Memory Allocation and Deallocation

Dynamic memory allocation The `malloc(size)` function from the `<stdlib.h>` library allocates the number of bytes of memory space based on the value of `size` and returns a pointer to the allocated memory block (from the "heap" segment of the system memory).

```
/* malloc.c */

#include <stdlib.h>

/* function prototype */
void *malloc(size_t size);

  /* allocate space for 10 integers */
  int *iptr = (int *) malloc (sizeof(int) * 10);

  /* allocate space for a string of 20 characters including '\0' */
  char *first_name = (char *) malloc (sizeof(char) * 20 + 1);
```

If the memory allocation is unsuccessful (e.g. out of memory), a NULL value is returned by `malloc` indicating that the memory request cannot be satisfied. Otherwise, `malloc` returns a `void *` pointer, which must be cast to an appropriate type.

Note: The `size_t` is essentially an `unsigned long` integer which indicated the size of a memory block requested in terms of *bytes*.

Releasing dynamic memory Any memory dynamically allocated during run-time must be released (de-allocated) and returned back to the heap by using the `free(ptr)` function (from `<stdlib.h>`), where `ptr` refers to the memory block to be de-allocated.

```
/* malloc.c */

#include <stdlib.h>

/* function prototype */
void free(void *ptr);

  /* de-allocate memory pointed by iptr and first_name */
  free(first_name);
  free(iptr);
```

Programs that fail to free up dynamic memory may suffer from "memory leaks" which may consume the entire system's resources; and result in system slow-down and prevents other programs from using the unused memory.

Note: You should not attempt to de-reference a `NULL` or freed pointer. Attempting to access a de-allocated memory block will cause undefined behaviour that may result in a program crash.

2.6.2 Multi-dimensional Arrays

Pointers and dynamic memory can be used to create multi-dimensional arrays of arbitrary size.

```c
/* multi_dimensional_arrays.c */

int array[][];          // an array of arrays
int **array;            // a pointer to a pointer
```

To allocate memory for a multi-dimensional array, the outer dimension should first be allocated with the memory. This is then followed by filling the inner dimensions.[3]

```c
/* multi_dimensional_arrays.c */

int **create_2Darray (int dimx, int dimy, int initial_value)
{
    int i, j;           /* counters */
    int **array;

    /* allocate an array of pointers */
    array = (int **) malloc (sizeof(int *) * dimy);

    for (i = 0; i < dimy; i++)
    {
      /* allocate an array of integers */
      array[i] = (int *) malloc (sizeof(int) * dimx);

      for (j = 0; j < dimx; j++)
        array[i][j] = initial_value;
    }
    return array;
}
```

As for the de-allocation process, memory is released in reverse order.

```c
/* multi_dimensional_arrays.c */

void free_2Darray (int **array, int dimy)
{
  int i;

  for (i = 0; i < dimy; i++)
  {
    free(array[i]);     /* free the memory of each row */
  }
  free(array);
}
```

[3] See `multi_dimensional_arrays.c` for the full source code.

More on arrays and pointers

What is the difference between the following two declarations?

```
int arr1[10][15];
int *arr2[10];
```

To access an element, `arr1[2][3]` and `arr2[2][3]` are both syntactically legal statements.

However, `arr1` is a true 2-dimensional array with 150 `int`-sized memory locations have been set aside (10 rows x 15 columns). Unlike `arr2` which is an array of pointers, memory space is only allocated to the 10 `(int *)` pointers without initialisation.

Memory must be explicitly allocated to each of the rows in `arr2`. (Note that each row in `arr2` may be of variable length, i.e. each row of `arr2` does not have to point to 15 elements.)

In general, arrays of pointers are useful in particularly when you need arrays of pointers to objects of different length.

```
/* length of each argument string is not known at compile-time */
char *argv[];

/* represent a collection of strings of variable length */
char *month[] = {"Illegal month", "January", ... , "December"};
```

2.7 Managing C programs

Often times, working with a small programming project, you could probably get away with putting all your code in one source file. For much larger projects, you would need to organise your functions together into various source files by grouping related functions together.

A general *rule of thumb* is that your functions should not be much larger than a terminal screenful (approximately 25 lines of code), excluding header comments. If your functions become larger than this, you should consider breaking them down into smaller functions. (Your function names should be descriptive to help other coders to navigate your source files.)

2.7.1 Organising your source files

One characteristic of the C Programming Language is that it encourages the idea of 'separate compilation.' Instead of putting all the code in one file and compiling that one file, C allows

you to write many `.c` files and compile them separately when necessary. The usual way to group related C source files is to put them under the same directory.

What goes into the `.c` file? Anything that causes the compiler to generate code should go into the `.c` source file. A `.c` source file usually consists of:

- implementation of functions (i.e. function bodies)
- declaration of global variables — in order to set aside memory for global variables

Anything that is intended as a message to the compiler should go into the `.h` header file, which include:

- function prototypes
- `struct` definitions
- `typedef` statements
- `#define` statements

Note: Only `.h` files should ever be included using the `#include` statement — you should never include `.c` files.

For example, if your program has a `.c` file named `arithmetic.c`, you would place the function prototypes and other related statements in a header file named `arithmetic.h`, and `#include "arithmetic.h"` both at the top of `arithmetic.c` and at the top of any other file that needs access to those function prototypes. Including just the header information in each file rather than the entire functions themselves avoids a problem where the C compiler 'sees' the same function redefined multiple times.

2.7.2 Building complex C programs

A C program can potentially consist of hundreds of source files (both `.c` and `.h` files), and possibly spanning across many directories.

The standard command given below is used to compile a complex C program with multiple source files. However, *this approach does not scale!*

```
1  $ gcc sourcefile1.c sourcefile2.c ... -o myprogram
```

Whenever the program is rebuilt, this command recompiles every source file, which can be time consuming. It should only be recompiling those source files that have been modified instead of every single file. The solution to this is by utilising the `make` command in C.

`make` is a utility tool in C that automates the building of program binaries from source files. A number of built-in rules are defined, such that `make` knows how to compile source code (`.c` files) into object code (`.o` files). `make` requires a special file called `makefile` that specifies the source files and the targets (which include the executable code) to be build.

2.7.3 The `makefile` format

A `makefile` typically consists of one or more entries. Each entry consists of the following:

- a target (which usually is a file but not always);
- a list of dependencies (files which the target depends on);
- and the commands to execute based on the target and its dependencies.

The basic format for each entry in a `makefile` looks like below:

```
1 <target>: [ <dependency> ]*
2 [ <TAB> <command> <ENDLINE> ]+
```

(**Note:** There must be a <TAB> character at the start of each command defined under the target. Indentation using spaces will *not* work in a makefile.)

An example of a `makefile` [4]:

```
1 sourcefile1.o: sourcefile1.c headerfile1.h
2   gcc -Wall -c sourcefile1.c
```

In the example above, the `-c` option is needed to create the corresponding object file (`.o`) for the given `.c` file. The `-Wall` option (stands for *Warnings: all*), which you might not have seen before, is a way to turn on extra warnings in `gcc`, which can be very useful for finding hidden problems in your code.

[4] The second line starts with a <TAB>, and not spaces.

How and when is a target constructed? Each entry in the `makefile` defines *how* and *when* to construct a target based on its dependencies. The dependencies are used by the `makefile` to determine when the target needs to be reconstructed (by re-compiling the source code).

Each file has a *timestamp* that indicates when the file was last modified.

The `make` command first checks the timestamp of the file (i.e. the target) and then the timestamp of its dependencies (which are also files). If the dependent files have a more recent timestamp than the target file, the command lines defined under the entry of that target are executed to update the target.

Note: The updates are done recursively. If any of the dependencies themselves are also targets (i.e. there is a separate entry in the `makefile` for each of the dependencies), the `make` command will check whether these dependent files need to be updated, before updating the initial target.

How to create the executable code using targets? The purpose of having the `makefile` is to create the executable code (i.e. `a.out`). The executable file is often the first target.

```
1 myprogram: sourcefile1.o sourcefile2.o sourcefile3.o
2   gcc -Wall sourcefile1.o sourcefile2.o sourcefile3.o -o myprogram
```

2.7.4 Running the `make` command

Once you have a `makefile` written, you can make use of the `make` command to update your files whenever you make changes. There are two ways to run `make` as shown below.

```
1 $ make
2 $ make -f <makefilename>
```

If you just type `make` on the command prompt, the file called `makefile` in the current directory will be interpreted and the commands of the first target will be executed, provided that the first target has dependencies listed.

Alternatively, if you have more than one `makefile` with different file names, you will run the `make` command with the `-f` option by supplying the file name of the specific `makefile` that you would like to run.

2.7.5 Defining macros in `makefile`

You can use *macros* in your `makefile`, which allow you to define variables that can be substituted.

You can use the "=" operator to set a value to the macro, and use the "$" operator when using the value of the macro[5]. (Note that the parentheses are required if the macro name has more than one character.)

Important: Don't get confused with the shell command prompt which can also be represented by a "$" symbol.

```
1  CC = gcc
2  CFLAGS = -Wall -c
3
4  sourcefile1.o: sourcefile1.c headerfile1.h
5    $(CC) $(CFLAGS) sourcefile1.c
```

The example above demonstrates two common macros `CC` and `CFLAGS`. Below are some of the other common examples of macros.

Macro	Description
`CC`	the name of the compiler (`gcc`)
`DEBUG`	the debugging flag (`-g`)
`CFLAGS`	the flags used for compiling (e.g. `-c`, `-Wall`)
`LFLAGS`	the flags used for linking (e.g. `-L`, `-I`)
`$@`	the file name of the target
`$^`	the list of dependencies
`$<`	the first item in the dependencies list

[5] If you have used shell variables in Bash before, `makefile` macros have a similar syntax.

Putting it all together Here is a sample `makefile` for creating the executable program called `myprogram`.

```
OBJS = sourcefile1.o sourcefile2.o sourcefile3.o
CC = gcc
CFLAGS = -Wall -c
LFLAGS = -Wall

myprogram: $(OBJS)
  $(CC) $(LFLAGS) $(OBJS) -o myprogram

sourcefile1.o: sourcefile1.c headerfile1.h
  $(CC) $(CFLAGS) sourcefile1.c

sourcefile2.o: sourcefile2.c headerfile2.h
  $(CC) $(CFLAGS) sourcefile2.c

sourcefile3.o: sourcefile3.c headerfile3.h
  $(CC) $(CFLAGS) sourcefile3.c
```

Try creating a `makefile` yourself:

Refer to Task 3 that you have completed for the Week 1 lab. First, organise your code into three separate files: `main.c`, `arithmetic.c`, and `arithmetic.h`.

- `main.c`: consists of the `main()` function
- `arithmetic.c`: consists of the implementation of the four arithmetic functions (i.e. function bodies)
- `arithmetic.h`: consists of the function prototypes of the four arithmetic functions

Then, create a `makefile` and try to compile your program by using the `make` command.

2.8 I/O System Calls in C

Note: You should complete the reading of this section before attempting the practical tasks in the next section.

The Standard I/O Library (`stdio`) provides a collection of *high-level* routines to perform input and output (I/O). This enables C programmers to perform reading and writing data easily and efficiently.

In general, the `stdio` routines perform three important functions for programmers:

- Buffering on input and output data is performed automatically;
- Input and output conversions are performed using routines like `printf` and `scanf`;
- Input and output are also automatically formatted.

However, these functions such as buffering and I/O conversions are not always applicable. In the event where performing I/O directly from a device such as a disk drive, programmers would need to be able to determine the actual buffer size to be used instead of relying on the `stdio` routines.

Thus, a direct interface to the operating system is desirable and can be achieved by issuing *system calls* or using the *low-level* I/O interface. *System calls* are low-level OS/kernel functions.

2.8.1 File descriptors

When dealing with the low-level I/O interface, each of low-level I/O routines requires a valid "file descriptor" to be passed to them — which is used to reference a file to an open stream for I/O.

A file descriptor is simply a small integer, which can be allocated from a file index table maintained for each process (or program) by the operating system.

There are three pre-defined file descriptors. You do not need to open these yourself; the operating system provides these automatically:

- File descriptor `0`: refers to standard input (usually the terminal keyboard)
- File descriptor `1`: refers to standard output (usually the terminal screen)
- File descriptor `2`: refers to standard error output (usually the terminal screen)

Important: When a process (or program) begins its execution, it starts out with three opened files — i.e. the three pre-defined file descriptors 0, 1, and 2. When the process opens its first file, it will then be attached with file descriptor 3.

2.8.2 Opening and creating files

In order to read from or write to a file (including the standard input and output), that file must first be opened. The routine (or function) used to *open* a file or to *create* a file for reading and/or writing is called `open()`.

The `open()` function takes two arguments:

- a character string consists of the path name of a file to be opened;
- a set of flags or integer constants specifying how the file is to be opened;

The `man` page for `open` (`man 2 open`) illustrates how this function is declared internally inside the system libraries, and what headers you must `#include` to use it.

Function prototype for the `open` system call:

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4
5  int open(const char *pathname, int flags);
```

The second argument is constructed by OR-ing together[6] a number of flags or constants (see below) defined in the header file `sys/fcntl.h` for System V systems, and `sys/file.h` for BSD (Berkeley Software Distribution) systems.

A subset of the flags (constants) that control how a file is to be opened are shown in the table on the next page. Note that the first three flags shown (i.e. `O_RDONLY`, `O_WRONLY` and `O_RDWR`) are mutually exclusive.

The `open` routine returns a file descriptor for the file if it is opened successfully. Otherwise, the value of `-1` is returned if it fails to open that file. An error code indicating the reason for failure is stored in the external variable `errno` defined in the header file `errno.h`. You can print out the error message with the `perror` function[7]. Note that `perror` is not a system call.

[6] Use the bitwise-or operator: | to combine multiple flag values together if needed.
[7] See `open_perror.c` for an example.

Flag (Constant)	Description
O_RDONLY	The file is opened for read only
O_WRONLY	The file is opened for write only
O_RDWR	The file is opened for both read and write
O_APPEND	The file is opened in append mode
O_CREAT	Create the file if it does not exist; the mode argument must be supplied to specify the access permission on the file
O_EXCL	Check if the file to be created is already exists; if already exists, open() will fail
O_NONBLOCK or O_NDELAY	The file is opened in non-blocking mode
O_TRUNC	If the file is opened for writing, truncate the length of the file to zero

Often times, the opened file should be closed once a process (or program) has finished using that file. The routine to *close* a file is called `close()`, which takes in only one argument representing the file descriptor of the file to be closed. If the file was closed successfully, the value of `0` is returned; otherwise, the value of `-1` is returned if an error occurs[8].

Function prototype for the `close` system call:

```
1  #include <unistd.h>
2
3  int close(int fd);
```

2.8.3 Reading and writing files

With the low-level I/O interface, the `read()` and `write()` routines are used for reading from and writing to a file respectively.

Function prototypes for the `read` and `write` system calls:

```
1  #include <unistd.h>
2
3  ssize_t read(int fd, void *buf, size_t count);
4  ssize_t write(int fd, const void *buf, size_t count);
```

[8] See `open_perror.c` for an example.

The `read()` system call attempts to read up to `count` bytes from the file referred to by the file descriptor `fd` into the buffer pointed by `buf`. The function returns the number of bytes actually read if no error occurs; other the value of `-1` is returned and the external variable `errno` is set with the error code. If the return value is `0`, that indicates the end of the file has been reached and there is no data left to be read.

The `write()` system call writes up to `count` bytes from the buffer pointed by `buf` to the file referred to by the file descriptor `fd`. If the write is successful, the number of bytes actually written is returned; otherwise the value of `-1` is returned if an error occurs and `errno` is set accordingly. (If nothing was written, the return value is `0`.)

When using these low-level functions, the programmers have to decide on the size of the buffer in bytes (as indicated by the third argument `count` in both functions). Note that each time the `read()` or `write()` function is executed, the operating system will access the disk (or other I/O device). If the buffer size is set to 1, meaning that you would only be able to read or write one byte (one character) at a time — the overall execution of your program is less efficient (especially when dealing with a large file) compared to reading or writing a whole block of characters at a time. In many cases the difference is not noticeable however.

2.8.4 Using `write()` with strings

While many C library functions use the C language convention of 'null-terminated' strings (using a '`\0`' character to mark the end of the string in memory), the underlying operating system does not assign any special meaning to the null '`\0`' character.

Therefore, when making a `write()` system call to the operating system, you must explicitly specify the number of characters to be written, regardless of whether the string is already null-terminated.[9] The `strlen()` function defined in `<string.h>` can be useful here: this function counts the number of characters in a (null-terminated) string by finding the position of the null character.

Common mistake: Beware of trying to use the `sizeof()` function to get the length of a string: depending on how your string is defined, you might be mistakenly getting the size of a `char*` (memory addresses are always 4 bytes in size on a 32-bit machine) or the size of an entire array, rather than the number of meaningful characters defined within it. In general, the `sizeof()` function is not the right tool for this job. It's only really meant for getting the memory size of data types rather than values themselves.

[9] In some cases, such as image and video processing, the null character may even represent useful data rather than the end of the string.

The following example shows how to use the `write()` system call to print a string to standard output (which is file descriptor `1`)[10]:

```
/* write.c */

  char* outputString = "Hello, world!\n";
  write(1, outputString, strlen(outputString));

  // Can we put the whole thing in one line like with a printf()?
  write(1, "This also works\n", 16);

  /*...but note that hard-coding the length like this is
  not considered ideal coding practice for readability! */
```

2.8.5 Moving around files

Often times, we need to be able to move around within a file to access the data stored at a specific location in the file. Each file is associated with a value, known as the *file offset*. It is often set to `0` indicating the beginning of the file when a file is opened or created.

The value of the file offset can be obtained or re-set by using `lseek()`.

Function prototype for the `lseek` system call:

```
  #include <sys/types.h>
  #include <unistd.h>

  off_t lseek(int fd, off_t offset, int whence);
```

The `lseek()` function re-positions the file offset (`offset`) of a file referred to by the file descriptor `fd`, based on the position in the file specified by the last argument in the function `whence`. The values that can be assigned to `whence` are as follows:

Offset (Constant)	Description
`SEEK_SET`	The file offset is set to `offset` bytes from the beginning of the file
`SEEK_CUR`	The file offset is set to `offset` bytes from its current position
`SEEK_END`	The file offset is set to `offset` bytes from the end of the file

[10] See `write.c` for the full source code.

So, if you were to move to the beginning or the end of a file:

```
lseek(fd, 0, SEEK_SET);    /* move to the beginning of the file */
lseek(fd, 0, SEEK_END);    /* move to the end of the file */
```

To obtain the value of the current offset of the file:

```
off_t current;
current = lseek(fd, 0, SEEK_CUR);
```

Note that the `lseek()` returns the new offset value in bytes from the beginning of the file if it runs successfully. If there is an error, the value of `-1` is returned. (The `errno` variable is set to indicate the error.)

2.9 Debugging C programs

Some kinds of errors, such as typos and incorrect algorithms, are equally common in all programming languages. However, there are some kinds of errors that are more likely to affect C programs, such as buffer overflows and misuse of pointers.

The underlying reason is that C generally lacks safeguards that other programming languages may have. C does not attempt to hide implementation details from the programmers; likewise C does not prevent the programmers from attempting to access memory that is not allocated to the programmers' code. One of the common errors under Unix/Linux is "segmentation fault" (`segfault`).

What is a segmentation fault? A segmentation fault occurs when you attempt to access memory that has not been allocated to it. It is generally a signal sent from the operating system (OS) to your program, and causing your program to exit. The OS that manages memory will not let your program read from or write to memory segments that are not owned by your program.

2.9.1 Handling errors with defensive programming

The best approach to handle errors in C is by preventing errors through "defensive programming" techniques.

- Always check return values from functions (such as `malloc()` and `fopen()`)
- Always initialise pointers to `null` on creation if the valid target is not given immediately
- Always check that pointers are not `null` before deferencing them (see below)

```
1  if (ptr_str) { //check not a null pointer
2    *ptr_str = "FIT2100";
3  }
```

Logging and diagnostics Often times, defensive programming is unable to catch all classes of errors. You can however instrument your code by getting it to output informative messages. For instance, `fprintf(stderr)` is one way to handle this[11] — `fprintf()` can also be used to output messages to a logfile. Alternatively, use `printf()` as diagnostic statements in code.

Note that these programming techniques are not specific to the C language or Unix/Linux. You would have certainly used diagnostic output statements in any other programming language that you have programmed in.

2.9.2 The `gdb` debugger

A debugger is a program that runs your program inside it, which allows you to inspect what the program is doing at a certain point during its execution. The debugger also enables you to manipulate the program state while it is running. As such, errors such as segmentation faults may be easier to detect with the help of a debugger.

A debugger that is often used with the C compiler `gcc` under the Unix/Linux environment is `gdb` — the GNU Debugger.

Usually, you would compile a C program as follows:

```
1  $ gcc [flags] <source files> -o <output file>
2
3  $ gcc sourcefile1.c sourcefile2.c ... -o myprogram
```

[11] This sends output to the program's 'standard error' stream; it will be displayed on the terminal even if the program's standard output has been redirected.

To run your program with the gdb debugger, you add a -g option to enable the built-in debugging support which is needed by the gdb, as with `valgrind`.

```
$ gcc [other flags] -g <source files> -o <output file>

$ gcc -g sourcefile1.c sourcefile2.c ... -o myprogram
```

Starting up the debugger To start up gdb, you could just try with one of the following commands. (Note: `myprogram` is the executable program that you would like to debug.)

```
$ gdb
$ gdb myprogram
```

You will then get a prompt that looks as follows:

```
(gdb)
```

If you didn't specify a program to debug while starting up the debugger, you will have to load it with the "`file`" command after started up the debugger.

```
$ gdb
(gdb) file myprogram
```

Running your program with the debugger To run your program (say `myprogram`), just type "`run`":

```
(gdb) run
```

Your program will get executed. There are two possible outcomes here:

- If there are no serious problems (such as, no segmentation fault occurred), your program should run accordingly with the program output presented.
- If your program did have some problems, you should be getting some useful information about the error. (See below for an example of the error message.)

```
Program received signal SIGSEGV, Segmentation fault.
0x00007fff916e4152 in strlen () from /usr/lib/system/libsystem_c.dylib
```

The "`where`" command can be used to find out the line of code where your program crashed. This is in fact the metadata that the -g flag includes in the executable program.

Another command that can be helpful for finding out where your program crashed is the "`backtrace`" command — which produces a stack trace of the function calls that lead to a segmentation fault. (An example of a stack trace is shown on the next page.)

```
(gdb) backtrace
#0  0x00007fff916e4152 in strlen () from /usr/lib/system/libsystem_c.dylib
#1  0x00007fff91729a54 in __vfprintf () from /usr/lib/system/libsystem_c.dylib
#2  0x00007fff917526c9 in __v2printf () from /usr/lib/system/libsystem_c.dylib
#3  0x00007fff9172838a in vfprintf_l () from /usr/lib/system/libsystem_c.dylib
#4  0x00007fff91726224 in printf () from /usr/lib/system/libsystem_c.dylib
#5  0x0000000100000f2c in main () at test.c:13
```

2.9.3 Dealing with the program bugs

If your program is running successfully, you would not need a debugger like `gdb`. But what if your program is not running properly or crashed? If that is the case, you do not want to run your program without stopping or breaking at a certain point during the execution. Rather, what you would want to do is to *step through* your code a bit at a time, until you discover where the error is.

Setting the breakpoints A *breakpoint* instructs `gdb` to stop running your program at a designated point. To set breakpoints, use the command "`break`" in one of the following ways:

- To break at line 5 of the current program:

```
(gdb) break 5
```

- To break at line 5 of a source file named `sourcefile.c`:

```
(gdb) break sourcefile.c:5
```

- To break at a function named `myfunction.c`:

```
(gdb) break myfunction
```

You can set multiple breakpoints within your program. If your program reaches any of these locations when running, your program will stop execution and prompt you for another debugger command.

Once you have set a breakpoint, you can try to run your program with the command "`run`". The program should then stop where you have instructed the debugger to stop — provided no fatal errors occurred before reaching the breakpoint.

To proceed with the next breakpoint, use the "`continue`" command. (You should not try to use the "`run`" command here as that would restart the program from the beginning, which is not going to be helpful.)

Note: Any breakpoints can be removed with the command "`clear`" — e.g. to delete the breakpoint at line 5:

```
(gdb) clear 5
```

Tracing your code When you program hits the breakpoint, you can step through your program one line at a time with the following commands:

- "`step`": go to the next line, stepping *into* any functions that the current line calls
- "`next`": go to the next line, but stepping *over* any functions that the current line calls
- "`finish`": finish the current function and *break* when it returns

Keeping an eye on variables You can also tell the `gdb` debugger to show the values of various variables or expressions at all times using the "`display`" or "`print`" command. Both of these commands take an optional format argument to display the value in a specific format:

- "`print/x`": print in hexadecimal
- "`print/c`": print in character
- "`print/t`": print in binary

```
(gdb) print myvariable
(gdb) print/x myvariable
```

If you would prefer to monitor or watch a particular variable whenever its value changes, use the "`watch`" command. The program will be interrupted whenever a watched variable's value is modified.

```
(gdb) watch myvariable
```

If you ever get confused about any of the `gdb` commands, use the "`help`" command with or without an argument.

```
(gdb) help [command]
```

Finally, to exit the `gdb` debugger is the "`quit`" command.

```
(gdb) quit
```

Graphical Debuggers: Alternatively, you can also use a graphical debugger. In your VM, the following graphical debuggers are pre-installed: `xxgdb`, `DDD`, `nemiver`, and `gdbgui`. Each of these debuggers has it's own pros and cons. Feel free to experiment and use any of them.

Try using the `gdb` debugger yourself:

For the given C program below, run it with the `gdb` debugger. You will need to step through the program code to detect where the error occurred. Once you have found the error, fix it so that the program will run successfully.

```c
/* gdb_practice.c */

#include <stdio.h>

int main(void)
{
   char str1[] = "FIT2100 Operating Systems";
   char str2[] = "Week 3 Tutorial";
   char *ptr = NULL;

   ptr = str1;
   printf("%s\n", ptr);

   ptr = str2;
   printf("%s\n", *ptr);

   return 0;
}
```

3 Assessed Tasks

3.1 Task 1 (25%)

Write a C program that accepts three integers as command-line arguments, then computes the *greatest common divisor* (GCD) among them. For example, the GCD for integers 12, 78, and 48 is 6.
Hints: The command-line arguments in page 12.

3.2 Task 2 (25%)

Write a C function that accepts an unsorted array of integers and the length of the array as arguments. The function should then compute the average value of all the numbers, and return the number closest to the average by returning a *pointer* to that closest element. The function prototype is given as follows:

```
1  int *find_closest_to_average(int the_array[], int num_elements);
```

3.3 Task 3 (25%)

Given the following `struct` type called `UnitCode`:

```
1  /* task3_unit_code.h */
2
3  typedef struct
4  typedef struct
5  {
6    char *FacID;            /* FIT, ENG, MTH, SCI,  etc.*/
7    int UnitID;             /* 1047, 2100, 3142, etc */
8  } UnitCode;
```

Create two variables of type `UnitCode` and initialise them with some appropriate values. Write a C function that determines whether these two `UnitCode` variables are equal to each other.

(Note: The program should perform comparison on all members of the `UnitCode` struct when checking for *equality*. For string comparisons, you may use the `strcmp` function from the `<string.h>` library as described in Section 2.4.3.)

3.4 Task 4 (25%)

Download `task4.c` from Moodle. This `task4.c` file is a partially completed C program, that takes one file name as its command-line argument. It opens the file for reading, counts the number of words in the file, and displays this count on the screen.

Complete the missing code segment in the program as indicated by the comment — /* CODE HERE */. To test the program, create a sample text file and name it as `sample.txt`. The contents of the file are given below by using the `less` command in Unix.

```
$ less sample.txt
This is a sample file.
It contains several words.
Counting the number of words is the task.
```

Note: To run or test the program in Unix, use the following command (You need to compile and create and executable file as "wordcount" to run the following command):

```
$ ./wordcount sample.txt
```

3.5 Wrapping Up

Please upload all your work to the Moodle submission link, which should include your .c source files and your responses to any non-coding tasks. Please note that non-code answers should be provided in plain text files. There is no need to include the compiled executable programs. Ensure to upload all the necessary files before the conclusion of your lab class. Any delay in submitting these files to Moodle will result in a late penalty.