FEEG 6002

Advanced Computational Methods I

Author: Yusaf Sultan
Lecturers: Ranga Dinesh Kahanda Koralage, Richard Boardman, Ondrej Hovorka
Word Count: 10161

Contents

List of Weeks

List of Figures

List of Tables

1. Lecture 1

START OF WEEK 1

1.1. Hello World in C using Quincy

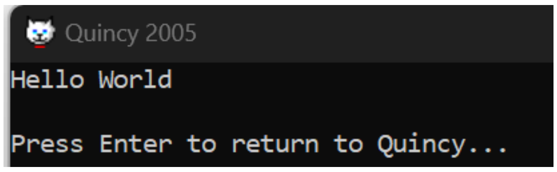
The code required to create an executable file which prints hello world to the console is shown in **Figure 1**.

```c
1 #include <stdio.h>
2
3 int main()
4 {
5   printf("Hello World \n");
6   return 0;
7 }
```

Figure 1: C code which prints hello world to terminal

- **Line 1**: Imports the header file called the **standard input output library** which contains the **`printf`** function that is utilized later in the code. This line is ran before the code is compiled and pastes in all of the functions within the header file.
- **Line 3**: Defines the **`main`** function which is always where execution starts for a C program. The function is defined as an integer using **`int`** and returns 0 to the OS upon successful execution.
- **Line 5** Utilizes the **`printf`** function to print "hello world" to the terminal. This function requires **`\n`** at the end to start a new line and each line of a C program requires a semicolon to signify the end of a line of code.

If this code is created as a C source file in Quincy it can be **saved** using **`ctrl + s`**, **compiled** using **`F5`** and than **executed** using **`F9`**. This will print the hello world text in the Quincy terminal as shown in **Figure 2**.



```
Hello World

Press Enter to return to Quincy...
```

Figure 2: **Figure 1** executable file output

When pressing **`F5`** Quincy sends a command to the C compiler which contains the following key flags:

- **`-ansi`**: Ensures compatibility with C90 standard.
- **`-Wall`**: Enables all warnings.
- **`-pedantic`**: Issues all warnings ISO C requires (C90 when using -ansi).

1.2. Using the Command Line

The command line is a **non-graphical way of working with a computer, via screen and keyboard.** There exist many different terminals, some common window terminal commands are shown below:

- `dir` : Display all files in the current folder.
- `cd \` : Navigate to root folder.
- `cd \<dirname>` : Navigate to the specified folder.
- `cd ..` : Move up one folder.
- `mkdir <newdir>` : Make a new folder.
- `rmdir <rmdir>` : Remove a folder.
- `rename <old> <new>` : Rename a file or folder.
- `move <file1> <file2>` : Move file1 to file2.
- `del <file>` : Delete file.
- `cls` : Clear screen.
- `help` : Shows all commands.
- `copy <src> <dest>`: “Copy a file from src to dest.”,
- `type <file>`: “Print the contents of a file.”,
- `echo <text>`: “Print text to the console.”,
- `exit`: “Close the command prompt.”,
- `path`: “Show system PATH variable.”,
- `ipconfig`: “Display network configuration.”,
- `ping <host>`: “Send ICMP packets to a host.”,
- `tasklist`: “List running processes.”,
- `taskkill /IM <process> /F`: “Kill a running process.”,
- `systeminfo`: “Display detailed system information.”

2. Lecture 2

2.1. Command Line

It was previously states that a command line is a non-graphical way of interacting with the computer using the screen and a keyboard. The command line like many other programming languages is a **REPL** type of program, which stands for:

- **Read**: read user input.
- **Evaluate**: run the command.
- **Print**: show the output.
- **Loop**: wait for the next command.

The command line can be used to manually perform many OS tasks (moving, renaming, deleting files and folders etc.,). Alternatively a bash file (`.bat`) can be used to chain together commands and automate scripts.

GUIs are not easy to program, often add unnecessary complexity to the program and are difficult to automate. The command line is therefore an attractive alternative for user interaction. The shell allows for interaction with the command line and also allows for many different programs to interact with one another. To allow for interactivity, Shell's use the **Unix philosophy** which states:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams (universal interface).

2.2. Input Output (I/O) Streams

The input and output of programs is done via a stream. There exists three standard streams for OSs which are shown in **Table 1**.

Standard Stream	Abbreviation	File Descriptor	Connected to
Input	`stdin`	0	Keyboard
Output	`stdout`	1	Screen
Error	`stderr`	2	Screen

Table 1: Standard streams

2.3. Program I/O in C

The built in C library `stdio` defines these streams. As well as some functions which make use of these streams, as shown below:

- `printf()` → Utilizes the `stdout` to print to the terminal.
- `fprintf()` → Takes an argument (either `stdout` or `stderr`) to either print out to standard out or standard error.
- `getchar()` → Get one character from `stdin` (note that this saves as the ASCII value of the character).
- `putchar()` → Send one character to `stdout` (note that this takes the ASCII number and outputs the corresponding character).

One common tool to stop an infinitely looping code is to use the **`EOF`** (end of file) character, which can be inputted in a windows shell by doing **`ctrl + Z`** and then pressing **`Enter`**.

Note that for **`fprintf()`** if **`stdout`** is passed then it behaves the same as **`printf`**. If **`stderr`** is instead passed than an **unbuffered, undirectable** output is printed to the shell.

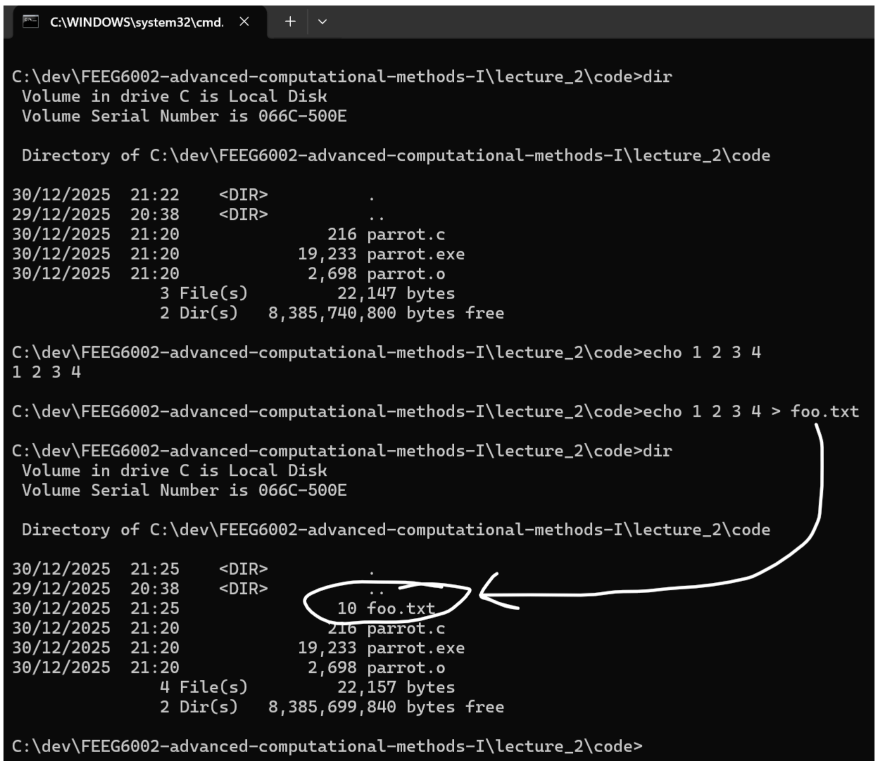
Applying these functions in one program, a parrot code can be created as shown in **Figure 3** which returns the inputted text from the terminal.

```c
#include <stdio.h>
int main()
{
  int c;
  fprintf(stderr, "Enter some characters (Ctrl-D quits)\n");
  while ((c = getchar()) != EOF) {
    putchar(c);
  }
  fprintf(stderr, "Goodbye\n");
  return 0;
}
```

Figure 3: C code which parrots the input and prints it back out to the terminal.

2.4. Program I/O in Terminal

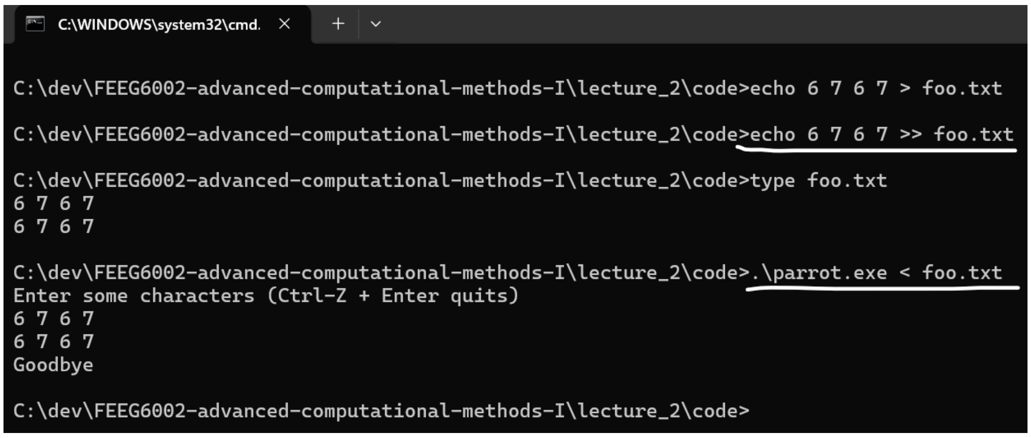
The **`echo`** command can be used to print things within the terminal (**`stdout`**), the standard output can be redirected and written to a file using **`>`** this is shown in **Figure 4**



```
C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>dir
 Volume in drive C is Local Disk
 Volume Serial Number is 066C-500E

 Directory of C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code

30/12/2025  21:22    <DIR>          .
29/12/2025  20:38    <DIR>          ..
30/12/2025  21:20               216 parrot.c
30/12/2025  21:20            19,233 parrot.exe
30/12/2025  21:20             2,698 parrot.o
               3 File(s)         22,147 bytes
               2 Dir(s)   8,385,740,800 bytes free

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>echo 1 2 3 4
1 2 3 4

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>echo 1 2 3 4 > foo.txt

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>dir
 Volume in drive C is Local Disk
 Volume Serial Number is 066C-500E

 Directory of C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code

30/12/2025  21:25    <DIR>          .
29/12/2025  20:38    <DIR>          ..
30/12/2025  21:25                10 foo.txt
30/12/2025  21:20               216 parrot.c
30/12/2025  21:20            19,233 parrot.exe
30/12/2025  21:20             2,698 parrot.o
               4 File(s)         22,157 bytes
               2 Dir(s)   8,385,699,840 bytes free

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>
```

Figure 4: Redirecting terminal **`stdout`**.

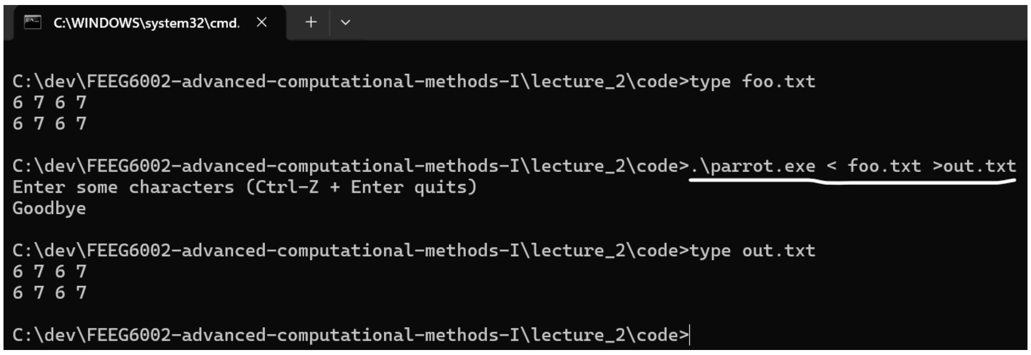
To append to a text file **`>>`** can be used instead. The **`stdin`** can also be redirected using the **`<`** operator, this is shown in **Figure 5**.



```
C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>echo 6 7 6 7 > foo.txt

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>echo 6 7 6 7 >> foo.txt

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>type foo.txt
6 7 6 7
6 7 6 7

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>.\parrot.exe < foo.txt
Enter some characters (Ctrl-Z + Enter quits)
6 7 6 7
6 7 6 7
Goodbye

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>
```

Figure 5: Redirecting terminal **`stdin`** into a executable file.

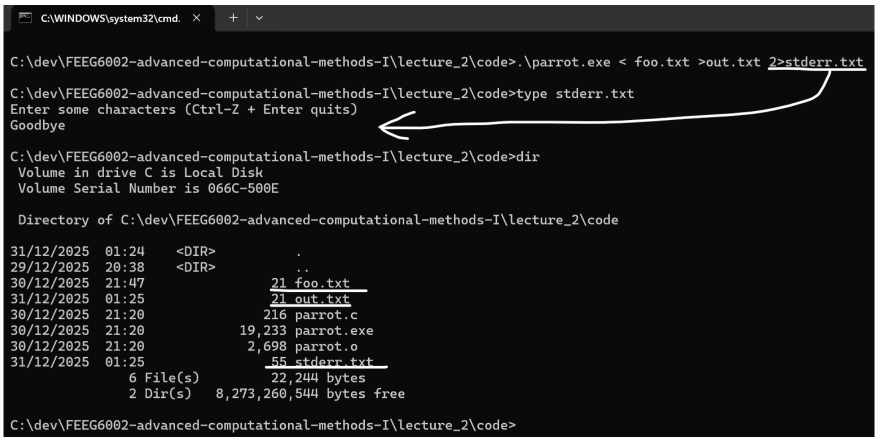
The output of the **`parrot.exe`** can itself be redirected into another text file by utilizing both the **`>`** command, as shown in **Figure 6**.



```
C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>type foo.txt
6 7 6 7
6 7 6 7

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>.\parrot.exe < foo.txt >out.txt
Enter some characters (Ctrl-Z + Enter quits)
Goodbye

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>type out.txt
6 7 6 7
6 7 6 7

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>
```

Figure 6: Redirecting program **`stdout`** into a text file.

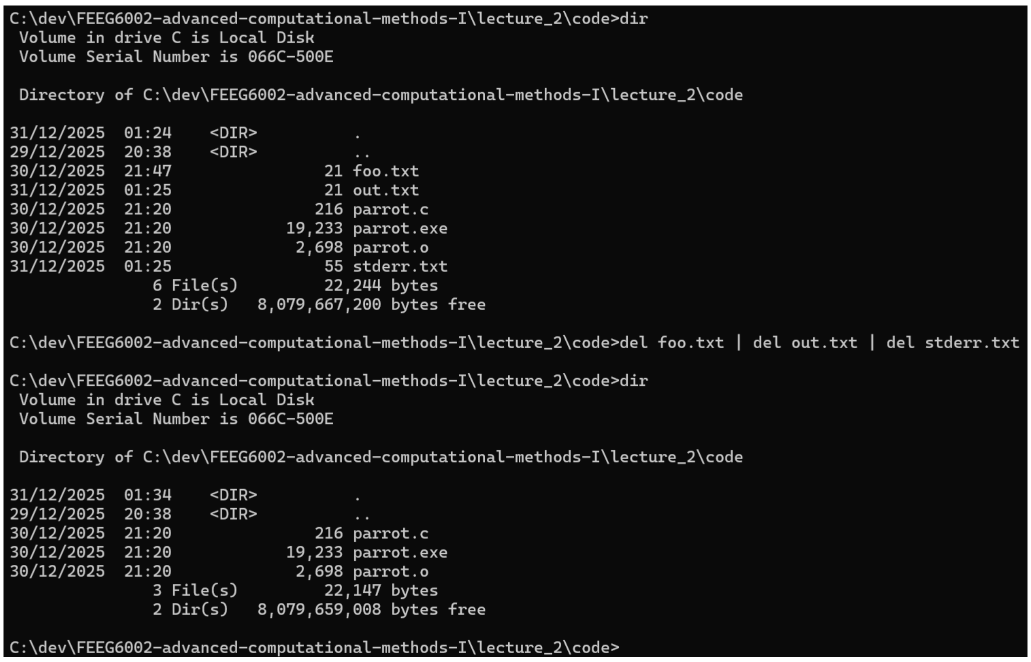
Note that the **`stderr`** from the program is still outputted to the shell, this can also be redirected using the **`2`** file descriptor, this is shown in **Figure 7**.



```
C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>.\parrot.exe < foo.txt >out.txt 2>stderr.txt

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>type stderr.txt
Enter some characters (Ctrl-Z + Enter quits)
Goodbye

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>dir
 Volume in drive C is Local Disk
 Volume Serial Number is 066C-500E

 Directory of C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code

31/12/2025  01:24    <DIR>          .
29/12/2025  20:38    <DIR>          ..
30/12/2025  21:47                21 foo.txt
31/12/2025  01:25                21 out.txt
30/12/2025  21:20               216 parrot.c
30/12/2025  21:20            19,233 parrot.exe
30/12/2025  21:20             2,698 parrot.o
31/12/2025  01:25                55 stderr.txt
               6 File(s)         22,244 bytes
               2 Dir(s)   8,273,260,544 bytes free

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>
```

Figure 7: Redirecting program **`stdout`** and **`stderr`** into a text file.

Note that if anything is ever re-routed to **`dev\null`** then this output is quietly destroyed. Pipes can be used to chain together commands, effectively chaining the **`stdout`** of one command to the **`stdin`** of the second command. An example of using pipes to chain together commands is shown in **Figure 8**.



```
C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>dir
 Volume in drive C is Local Disk
 Volume Serial Number is 066C-500E

 Directory of C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code

31/12/2025  01:24    <DIR>          .
29/12/2025  20:38    <DIR>          ..
30/12/2025  21:47                21 foo.txt
31/12/2025  01:25                21 out.txt
30/12/2025  21:20               216 parrot.c
30/12/2025  21:20            19,233 parrot.exe
30/12/2025  21:20             2,698 parrot.o
31/12/2025  01:25                55 stderr.txt
               6 File(s)         22,244 bytes
               2 Dir(s)   8,079,667,200 bytes free

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>del foo.txt | del out.txt | del stderr.txt

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>dir
 Volume in drive C is Local Disk
 Volume Serial Number is 066C-500E

 Directory of C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code

31/12/2025  01:34    <DIR>          .
29/12/2025  20:38    <DIR>          ..
30/12/2025  21:20               216 parrot.c
30/12/2025  21:20            19,233 parrot.exe
30/12/2025  21:20             2,698 parrot.o
               3 File(s)         22,147 bytes
               2 Dir(s)   8,079,659,008 bytes free

C:\dev\FEEG6002-advanced-computational-methods-I\lecture_2\code>
```

Figure 8: Utilizing pipes in the terminal to chain commands.

3. Lecture 3

START OF WEEK 2

4. Lecture 4

4.1. Basic Data Types

There exists 4 key datatypes within C, these are:

- **`int`** $\rightarrow$ Integer or a whole number. Uses 4 bytes (32 bits) of memory with 1 sign bit and 31 magnitude bits. Can store numbers in the range of $-2,147,483,648$ to $+2,147,483,647$
- **`char`** $\rightarrow$ Stores a single character using 1 byte (4 bits). Stores the character as an ASCII code in memory.
- **`float`** $\rightarrow$ Stores a decimal number in 4 bytes (32 bits). Called single precision, this allows for 6 - 7 decimal digits and has a range from 1.175×10^{-38} to 3.402×10^{38}.
- **`double`** $\rightarrow$ Stores a decimal number in 8 bytes (64 bits). Called double precision, this allows for 15 - 16 decimal digits and has a range from 2.225×10^{-308} to 1.798×10^{308}.
- **`bool`** $\rightarrow$ Can be True or False (1 or 0).

4.2. Signed, Unsigned, Short and Long Datatypes

`int` and **`char`** datatypes can be signed or unsigned, by default they are both signed. Declaring an integer as unsigned means it can **only store non-negative values**. This means instead of the range being centered at zero, all bits are used to store the number. For a 32 bit integer, this means the range for this variable is 0 to 4,294,967,295.

An integer can also be defined as **`short`**, **`long`** and **`long long`**, which specify how many bits are used to store the number. The number of bits for each type as well as their maximum ranges are show in **Table 2**.

Keyword	Windows Size	Linux/macOS Size	Signed	Unsigned Range
short	16-bit	16-bit	±32k	0 - 65k
int	32-bit	32-bit	±2.1B	0 - 4.2B
long	32-bit	64-bit	varies	varies
long long	64-bit	64-bit	±9e18	0 - 1.8e19

Table 2: Integer type sizes and ranges

4.3. Using `printf` to Print Variables

The **`printf`** command can be used to print out all of the variable types perviously mentioned. On top of this, formatting can also be used to print specific outputs. An example code which shows this is shown in **Figure 9**.

```c
# include <stdio.h>

int main(void) {
  int var1 = -42;
  unsigned int var2 = 142;
  float var3 = 3.14;
  double var4 = 3.1428;
  char var5[] = "Hello World";

  printf("Printing int var1=%d\n", var1);
  printf("Printing unsigned int var2=%u\n", var2);
  printf("Printing float var3=%f\n", var3);
  printf("Printing double var4=%f\n", var4);
  printf("Printing double in scientific notation var4=%e\n", var4);
  printf("Printing char[] var5=%s\n", var5);
  return 0;
}
```

```
C:\>.\printing_data_types.e
Printing int var1=-42
Printing unsigned int var2=142
Printing float var3=3.140000
Printing double var4=3.142800
Printing double in scientific notation var4=3.142800e+000
Printing char[] var5=Hello World
```

Figure 9: C code which assigns each variable type and prints it.

Further formatting can be achieved by using specific formatters, this is shown in **Figure 10**.

```c
#include <stdio.h>

int main(void) {

  double pi = 3.1415926535897931;

  printf("As a standard float representation: pi=%f\n", pi);
  printf("In exponential notation: pi=%e\n", pi);
  printf("Whichever of the above two is shorter: pi=%g\n", pi);
  printf("Request 10 digits overall: pi=%10f\n", pi);
  printf("With 3 postdecimal digits: pi=%10.3f\n", pi);
  printf("12 postdecimal digits: pi=%.12f\n", pi);

return 0;
}
```

```
C:\>.\printing_data_types.e
As a standard float representation: pi=3.141593
In exponential notation: pi=3.141593e+000
Whichever of the above two is shorter: pi=3.14159
Request 10 digits overall: pi=  3.141593
With 3 postdecimal digits: pi=     3.142
12 postdecimal digits: pi=3.141592653590
```

Figure 10: C code which customizes the output of **`printf`** using formatting flags.

A summary of which formatters are used with specific datatype is shown below:

- **`%d`** & **`%i`** → Signed **`int`**, **`short`**, **`chars`**.
- **`%ld`** & **`%li`** → Signed **`long`** integers.
- **`%f`** → Floating point representation of **`double`** or **`float`**.
- **`%e`** → Exponential notation of **`double`** or **`float`**.
- **`%g`** → **`f`** or **`e`** depending on which is shorter.
- **`%s`** → Arrays of characters (**`char`**, **`char[]`**).
- **`%u`** → Unsigned integers (unsigned **`int`**, **`short`**, **`char`**).
- **`%<n1>.<n2>f`** → **`n1`** overall digits and **`n2`** decimal digits (This can result in whitespace).

4.4. Using `scanf`

The **`scanf`** command is useful for allowing the user to input a value which can then be saved as a variable, an example program which takes a character and integer input is shown in **Figure 11**.

```c
1   #include <stdio.h>
2
3   int main(void)
4   {
5       int inputted_numb;
6       char inputted_char;
7
8       printf("Enter an integer.\n");
9       scanf("%d", &inputted_numb);
10      printf("Inputted number = %d\n", inputted_numb);
11
12      printf("Enter a character.\n");
13      scanf(" %c", &inputted_char);
14      printf("Inputted character = %c\n", inputted_char);
15
16      return 0;
17  }
```

```shell
C:\>.\scanf_example.exe
Enter an integer.
67
Inputted number = 67
Enter a character.
B
Inputted character = B
```

Figure 11: C code which uses **`scanf`** to take in a user input.

Note that for this code to run a space is required in line 13 before **`%c`** as the **`\n`** from the previous line effects it. Furthermore, the **`&`** refers to the memory address of the variable, not its value.

4.5. Conditional Statements in C

`if-else` and **`if-else if-else`** statements can be used to run a specific block of code when a condition is met. There are two types of conditions that can be specified within a conditional, these are **relational operators** and **logical operators**. These can be combined together to form composite statements, all of these operators are shown below:

- **Relational operators**:
- **`a==b`** → a equal to b
- **`a!=b`** → a not equal to b
- **`a>b`** → a greater than b
- **`a<b`** → a less than b
- **`a>=b`** → a greater than or equal to b
- **`a<=b`** → a less than or equal to b

- **Logical operators**:
- **`&&`** → logical AND
- **`||`** → logical OR
- **`!`** → logical NOT

An example code that uses composite statements using multiple operators is shown in **Figure 12**.

```c
#include <stdio.h>

int main(void)
{
    int user_input;

    printf("Enter an integer.\n");
    scanf(" %d", &user_input);

    if (user_input > -10 && user_input < 0) {
        printf("Input is between -10 and
        0\n");
    } else if (user_input == 5) {
        printf("Input is 5\n");
    } else {
        printf("The input is: %d\n",
        user_input);
    }

    return 0;
}
```

```shell
C:\>.
\conditional_statements.exe
Enter an integer.
5
Input is 5
C:\>.
\conditional_statements.exe
Enter an integer.
-7
Input is between -10 and 0
C:\>.
\conditional_statements.exe
Enter an integer.
52
The input is: 52
```

Figure 12: C code which uses conditional statements.

5. Lecture 5

START OF WEEK 3

5.1. For Loop

A for loop in C requires three input conditions, these are:

- **Initialization** → An expression that is executed once at the start of the loop, often used to initialize a counter variable.
- **Condition** → An expression that is checked before every iteration. If the condition is true, the loop executes, if false, then the loop stops.
- **Update** → An expression applied after each iteration of the loop. It typically increments or updates the loop variable.

Some basic code which utilizes a for loop is shown in **Figure 13**. Note that **i++** is equivalent to **i = i + 1**.

```c
#include <stdio.h>

int main(void){

    int i;
    int N = 5;

    for (i=0; i<N; i++){
      printf("This is iteration %d of %d.\n", i, N-1);
    }
    return 0;
}
```

```
C:\>.\basic_for.exe
This is iteration 0 of 4.
This is iteration 1 of 4.
This is iteration 2 of 4.
This is iteration 3 of 4.
This is iteration 4 of 4.
```

Figure 13: C code which uses a for loop.

Note that using **++i** ≠ **i++**. Consider the code shown in **Figure 14**.

```c
#include <stdio.h>

int main(void) {
    int a = 5;
    int b = 5;

    printf("++a (pre-increment): value = %d\n", ++a);
    printf("b++ (post-increment): value = %d\n", b++);
    printf("b after post-increment = %d\n", b);
    return 0;
}
```

```
C:\>.\pre_post_increment
++a (pre-increment): value = 6
b++ (post-increment): value = 5
b after post-increment = 6
```

Figure 14: C code which uses a for loop.

5.2. While Loop

A while loop will iterate as long as a defined condition is not met. Note that there is a risk of infinite looping if the condition is never met. An example of a while loop in C is shown in **Figure 15**.

```c
#include <stdio.h>

int main(void) {
  int i = 1;

  while (i < 10) {
    printf("i = %d \n", i);
    i = i * 2;
  }
  return 0;
}
```

```shell
C:\>.\while_loop.exe
i = 1
i = 2
i = 4
i = 8
```

Figure 15: C code which uses a while loop.

Typically a **for loop** is used when the **number of iterations is known**. On the other hand, a **while loop** when the number of iterations is unknown but there is some sort of **criteria or tolerance** that can be built into a condition.

5.3. Buffering

The standard output **`stdout`** is often buffered, meaning it stores the output data in a buffer and writes it out all at once. The standard error **`stderr`** is typically unbuffered, which means error messages appear immediately.

6. Lecture 6

6.1. How Computers Store numbers

A computer stores numbers or data in bits, which can either be 1 or 0. A bit is the smallest unit of data, chaining multiple bits together yields the following:

- **1 bit**
- **1 byte** = 8 bits
- **1 kilobyte** = 10^3 bytes (kb)
- **1 megabyte** = 10^6 bytes (Mb)
- **1 gigabyte** = 10^9 bytes (Gb)

Integer numbers can be converted to bits (binary) by expanding power of 2 using **Eq. 1**.

$$D = \sum_{i=0}^{8-1} b_i 2^i \tag{1}$$

Where b_i is either 0 or 1 and i is the number of bits. For example if $D = 13$ then the 8 bit binary representation of this number is 00001101_2. Negative numbers can be represented by using a **sign bit**, which is where the most significant bit represents the sign of the number.

An **unsigned 8 bit integer** can range from 0 (00000000_2) to 255 (11111111_2). A **signed 8 bit integer** can range from -128 (10000000_2) to 127 (11111111_2).

6.2. Data Type Limits

As was stated in **Table 2**, there are limits for the size of each data type. These can be calculated using **Eq. 1** but can also be displayed by C itself, as shown in **Figure 16**.

```c
#include <limits.h>  /* limits for integers */
#include <float.h>   /* limits for floats */
#include <stdio.h>

int main(void) {
  printf("    CHAR_MIN = %12d\n", CHAR_MIN);
  printf("    CHAR_MAX = %12d\n", CHAR_MAX);
  printf("    SHRT_MIN = %12d\n", SHRT_MIN);
  printf("    SHRT_MAX = %12d\n", SHRT_MAX);
  printf("     INT_MIN = %12d\n", INT_MIN);
  printf("     INT_MAX = %12d\n", INT_MAX);
  printf("    LONG_MIN = %12ld\n", LONG_MIN);
  printf("    LONG_MAX = %12ld\n", LONG_MAX);
  printf("     FLT_MIN = %12e\n", FLT_MIN);
  printf("     FLT_MAX = %12e\n", FLT_MAX);
  printf("     DBL_MIN = %12e\n", DBL_MIN);
  printf("     DBL_MAX = %12e\n", DBL_MAX);
  return 0;
}
```

```shell
C:\>.\displaying_limits.exe
CHAR_MIN =         -128
CHAR_MAX =          127
SHRT_MIN =       -32768
SHRT_MAX =        32767
 INT_MIN =  -2147483648
 INT_MAX =   2147483647
LONG_MIN =  -2147483648
LONG_MAX =   2147483647
 FLT_MIN = 1.175494e-038
 FLT_MAX = 3.402823e+038
 DBL_MIN = 2.225074e-308
 DBL_MAX = 1.797693e+308
```

Figure 16: C code which displays the maximum value limits of datatypes.

6.3. Floating Point numbers

Computers can store decimal numbers using floating point representation, the general form of a floating point number is shown in **Eq. 2**.

$$x = a \cdot 10^b \tag{2}$$

Where $\boldsymbol{a} \rightarrow$ mantissa and $\boldsymbol{b} \rightarrow$ exponent. Assuming 32bit precision, and adhering to the IEEE 754 standard, the bits are assigned as follows:

- **1 sign bit.**
- **8 bits for the exponent** representing the powers of 2, with a bias of 127.
- **23 bits for the mantissa** representing the significant digits of the number.

The issue with representing numbers in this way is that **real numbers can only be represented with finite precision**. Often simple division will leave a very small error which can accumulate in a iterative program, **Figure 17** depicts this.

```c
#include <stdio.h>

int main(void) {

    float a, b, result;
    float x, y, z;

    a = 1.0f;
    b = 3.0f;
    result = a / b;
    printf("a / b = %.20f\n", result);

    x = 0.1f;
    y = 0.2f;
    z = x + y;
    printf("0.1 + 0.2 = %.20f\n", z);

    return 0;
}
```

```
C:\>.\numerical_error.exe
a / b = 0.33333334326744080000
0.1 + 0.2 = 0.30000001192092896000
```

Figure 17: C code which depicts the numerical instability with floating point numbers.

6.4. Integer Division

Often integer division in C will truncate decimal places, to combat this the variables must be cast as a float or double either before or during the division, this is shown in **Figure 18**.

```c
#include <stdio.h>

int main(void) {

  int a=10; int b=3; double c;

  c = (double) b/a; /*HERE*/
  printf("a=%d, b=%d, c=%f\n", a, b, c);

  return 0;
}
```

```
C:\>.\integer_division.exe
a=10.000000, b=3.000000, c=0.30000
```

Figure 18: C code which depicts correct integer division.

6.5. Functions in C

Instead of writing sections of code again and again, functions can be used to compartmentalize the code and then these functions can be easily called over and over again. there are four main parts of a function, these are:

- **return type** → Specifies what kind of value the function returns. It can be any valid C data type like **`int`**, **`float`**, **`char`**, etc. If a function does not return anything, its return type is **`void`**.
- **function name** → This is the identifier for the function. The name is used to call the function later in the program.
- **parameters** → Functions can take inputs, known as parameters or arguments, which are passed to the function when it is called. These parameters are declared inside parentheses ().
- **function body** → the code inside the function that performs the task is written within curly braces {}.
- **return** → if the function has a return type other than **`void`**, it must return a value using the return statement.

An example of a function is shown in **Figure 19**

```c
#include <stdio.h>

int power(int base, int n) {
    /* power: raise base to the n-th power; n >= 0 */

    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i) {
        p = p * base;
```

```
C:\>.\function_example.exe
0       1       1
1       2      -3
2       4       9
3       8     -27
4      16      81
5      32     -24
```

```c
    }

    return p;
}

int main(void) {

    int i;

    for (i = 0; i < 6; ++i) {
        printf("%2d %6d %6d\n", i, power(2, i), power(-3, i));
    }

    return 0;
}
```

Figure 19: C code which depicts an example of a function.

Instead of the structure shown in **Figure 19**, a function can instead be initialized before its call, this is called a **functions prototype**, the prototype for the function in **Figure 19** is **`int power(int base, int n);`**. Prototypes are what is used by header files.

Whenever a variable is passed into a function, a copy of that variables contents are used, not the variable itself.

6.6. Symbolic Constants and Types

Symbolic constants and types can be defined at the top of C code and allow for creation of constants which are the same throughout the code. This is achieved by using **`#`** which in C signifies a process the pre-processor must do. An example of these are shown in **Figure 20**.

```c
#include <stdio.h>
#include <math.h> /* include math-header for sqrt-function */

#define N 10   /* total number of lines in table */
#define a 1.0  /* starting with x=a */
#define b 10.0 /* ending with x=b */

#define MYINT int

MYINT main(void) {
    /* print table of N square roots sqrt(x) for x in [a,b] */

    double x; /* being used in for-loop */
```

```
C:\>.\symbolic_things.exe
sqrt(1.000000) = 1.000000
sqrt(2.000000) = 1.414214
sqrt(3.000000) = 1.732051
sqrt(4.000000) = 2.000000
sqrt(5.000000) = 2.236068
sqrt(6.000000) = 2.449490
sqrt(7.000000) = 2.645751
sqrt(8.000000) = 2.828427
sqrt(9.000000) = 3.000000
sqrt(10.000000) = 3.16227
```

```c
    MYINT i;    /* iteration counter for for-loop */

    for (i = 0; i < N; i++) {
        x = a + i * (b - a) / (N - 1); /* compute x */

        /* compute sqrt(x) and print result */
        printf(" sqrt(%f) = %f\n", x, sqrt(x));
    }

    return 0;
}
```

Figure 20: C code which depicts use of symbolic constants.

7. Lecture 7

START OF WEEK 4

7.1. Arrays

Arrays allow for storing multiple values within one organized structure. An example of some code which utilizes an array is shown in **Figure 21**.

```c
#include <stdio.h>

#define N 5

int main(void) {

    int numbers[N]; /* Declare an array of size 5 */
    int i, n;
    int sum = 0;
    float average;

    /* Prompt the user to input values into the array */
    printf("Enter %d numbers:\n", N);
    for (i = 0; i < N; i++) {
        printf("Number %d: ", i + 1);
        scanf("%d", &n);
        numbers[i] = n;
    }

    for (i = 0; i < N; i++) { /* sum of the array elements */
        sum += numbers[i];
    }

    average = (double) sum / N; /* average */

    printf("Sum of the array elements: %d\n", sum);
    printf("Average of the array elements: %.2f\n", average);

    return 0;
}
```

```shell
C:\>.\arrays_example
Enter 5 numbers:
Number 1: 1
Number 2: 2
Number 3: 3
Number 4: 4
Number 5: 5
Sum of the array elements: 15
Average of the array elements: 3.0
```

Figure 21: C code which depicts use of arrays.

When arrays are initialized with the syntax shown in **Figure 21**, this is referred to as the **static definition** of the array. The length of the array is immutable and the memory allocated for teh array is fixed.

7.2. Strings

Strings are essentially just an array of characters. There are two ways to define a string, the first is to define an empty character array and then populate it as shown in **Figure 22**. Note that **\0** is appended at the end, this signifies to C that this is a string and not just an array of characters, allowing it to be printed.

```c
#include <stdio.h>

/* demonstrate string termination */
int main(void) {
    char a[10];

    a[0] = 'h';
    a[1] = 'e';
    a[2] = 'l';
    a[3] = 'l';
    a[4] = 'o';
    a[5] = '\n';
    a[6] = '\0';

    printf("%s", a);

    return 0;
}
```

```
C:\>.\manual_string.exe
hello
```

Figure 22: C code which manually constructs a string.

Another way of creating a string is by defining it at creation of the variable. In this way the length of the string does not need to be explicitly defined as it is implied by the length of the text. Furthermore the **\0** is also not needed, an example of this is shown in **Figure 23**.

```c
#include <stdio.h>

/* demonstrate string termination I */
int main(void) {
    char s[] = "hello\n";
    int i;

    printf("%s", s);
    for (i = 0; i < 7; i++)
        printf("s[%d]='%c'=%3d\n", i, s[i], s[i]);

```

```
C:\>.\auto_string.exe
hello
s[0]='h'=104
s[1]='e'=101
s[2]='l'=108
s[3]='l'=108
s[4]='o'=111
s[5]='
'= 10
s[6]=''=  0
```

```c
    return 0;
}
```

Figure 23: C code which ‘automatically’ creates a string.

Note that in the output of **Figure 23**, the new line character which is the 5th character in the string is printed and the output is put onto a new line.

7.3. String Manipulation

There is a standard library in C that can be used to do string manipulation, this is called **`<string.h>`**. It has within it the following functions:

- **`strlen()`** → Get the length of the string (excluding the null character).
- **`strcpy()`** → Copy one string to another.
- **`strcat()`** → Concatenate (append) two strings.
- **`strcmp()`** → Compare two strings.

An example C code which utilizes these functions is shown in **Figure 24**.

```c
#include <stdio.h>
#include <string.h>

int main(void) {

    char s1[50] = "hello";
    char s2[50] = "world";
    char copy[50];
    int cmp;

    /* strlen */
    printf("Length of s1 = %lu\n", (unsigned long) strlen(s1));
    printf("Length of s2 = %lu\n", (unsigned long) strlen(s2));

    /* strcpy */
    strcpy(copy, s1);
    printf("After strcpy, copy = %s\n", copy);

    /* strcat */
    strcat(s1, " ");    /* add a space */
    strcat(s1, s2);     /* append "world" */
    printf("After strcat, s1 = %s\n", s1);

    /* strcmp */
```

```shell
.\string_manip.exe
Length of s1 = 5
Length of s2 = 5
After strcpy, copy = hello
After strcat, s1 = hello world
s1 comes BEFORE s2 in dictionary order
```

```c
    cmp = strcmp(s1, s2);
    if (cmp == 0) {
        printf("s1 and s2 are equal\n");
    } else if (cmp < 0) {
        printf("s1 comes BEFORE s2 in dictionary order\n");
    } else {
        printf("s1 comes AFTER s2 in dictionary order\n");
    }

    return 0;
}
```

Figure 24: C code which uses functions from the **`string.h`** library to manipulate strings.

8. Lecture 8

8.1. Pointers

Pointers are one of the most powerful concepts within C. The following syntax can be used for pointers:

- **`int *p;`** → Initialize the pointer.
- **`p = &c`** → The pointer now holds the memory address of of **`c`**. We say **`p` points to `c`**.
- **`y = *p`** → The variable **`y`** now contains the value of the address that the pointer was pointing to (effectively **`y = c`**).
- **`*p = 0`** → The variable linked to the memory address of the pointer is now set to 0.

An example showing how pointers work is shown in **Figure 25**.

```c
#include <stdio.h>

int main(void)
{
    int x = 1, y = 2;
    int *pi; /* pi is pointer to int */

    pi = &x; /* pi now points to x */
    y = *pi; /* y is now 1 */
    *pi = 0; /* x is now 0 */
    pi = &y; /* pi points to y */

    printf("x=%d, y=%d \n", x, y);
    printf("address x=%p, address y=%p \n", (void *) &x, (void *) &y);
    printf("pi=%p, *pi=%d \n", (void *) pi, *pi);

    return 0;
}
```

```
C:\.\pointers_basic.exe
x=0, y=1
address x=0060FF0C, address y=0060FF08
pi=0060FF08, *pi=1
```

Figure 25: C code which illustrates the basic principles of pointers.

Note that in **Figure 25**, the **`printf`** statements use a unique syntax. The **`%p`** formatter is used to print memory addresses, however **`(void *)`** is required as **`%p`** expects a generic pointer type. **`void`** is essentially stripping the type of the pointer to make it generic.

8.2. Swapping Variables

Suppose that we wanted to create a function to swap the value of two variables. An initial attempt at this function is shown in **Figure 26** which does not use pointers.

```c
#include <stdio.h>

void swap(int x, int y) {
    int tmp;

    tmp = x;
    x = y;
    y = tmp;
}

int main(void) {
    int a = 1, b = 2;

    printf("Before swap a=%d, b=%d\n", a, b);
    swap(a, b);
    printf("After swap a=%d, b=%d\n", a, b);

    return 0;
}
```

```
C:\.
\incorect_swap.exe
Before swap a=1, b=2
After swap a=1, b=2
```

Figure 26: C code which incorrectly attempts to swap the value of two variables.

Because the variables are passed by value, swapping their copies over does not change over the original values. Instead pointers can be used tom correctly switch over the two variables within the function using their memory addresses. This is shown in **Figure 27**.

```c
#include <stdio.h>

void swap(int *px, int *py) {
    int tmp;

    tmp = *px;
    *px = *py;
    *py = tmp;
}

int main(void) {
    int a = 1, b = 2;

    printf("Before swap a=%d, b=%d\n", a, b);
    swap(&a, &b);
    printf("After swap a=%d, b=%d\n", a, b);

    return 0;
}
```

```
C:\.
\incorect_swap.exe
Before swap a=1, b=2
After swap a=2, b=1
```

Figure 27: C code which correctly swaps the value of two variables.

9. Lecture 9

START OF WEEK 5

9.1. Pointers and Arrays

Pointers and arrays are strongly linked together. Consider an array of 10 elements called **a** (**int a[10]**). If an integer pointer is defined **int *pa** and then told to point to the memory address of the first element in the array **pa = &a[0]** (equivalent to **pa = a** as **a** is a pointer to the first object anyways) then adding 1 to **pa** will move to the next element in the array. This happens regardless to the datatype used, that many bits of memory area always moved when adding an integer.

Whenever an array is passed into a function , a copy of the array is not sent to the function. Instead a pointer to the first element of the array is passed, meaning the following two lines are equivalent.

- **void print_array(int a[], int n);**
- **void print_array(int *a, int n);**

9.2. Sizeof Function

The **sizeof** operator is used tor return the length of a datatype in bytes. This function is included in the standard input output library and is crucial for structs and dynamic memory allocation. An example usecase for this variable is shown in **Figure 28**.

```c
#include <stdio.h>

int main(void) {

    printf("sizeof(char) = %d\n", (int) sizeof(char));
    printf("sizeof(short) = %d\n", (int) sizeof(short int));
    printf("sizeof(int) = %d\n", (int) sizeof(int));
    printf("sizeof(long) = %d\n", (int) sizeof(long int));
    printf("sizeof(float) = %d\n", (int) sizeof(float));
    printf("sizeof(double) = %d\n", (int) sizeof(double));
    printf("sizeof(double *) = %d\n", (int) sizeof(double *));
    printf("sizeof(char*) = %d\n", (int) sizeof(char *));
    printf("sizeof(FILE*) = %d\n", (int) sizeof(FILE *));

    return 0;
}
```

```shell
C:\.\sizeof.exe
sizeof(char) = 1
sizeof(short) = 2
sizeof(int) = 4
sizeof(long) = 4
sizeof(float) = 4
sizeof(double) = 8
sizeof(double *) = 4
sizeof(char*) = 4
sizeof(FILE*) = 4
```

Figure 28: C code which utilizes the **sizeof** operator to display the length of the datatypes.

9.3. Dynamic Memory

So far the memory allocation that has been used is static, meaning that once it has been allocated it cannot be changed. However there also exists dynamic memory, some of the key differences between the two memory allocations are shown in **Table 3**.

Feature	Static	Dynamic
Allocation	Automatic	Manual (**`malloc`**)
Speed	Very fast	Slower
Size	Small	Large
Lifetime	Function scope	Until **`free()`**
Control	Compiler	Programmer

Table 3: Static vs Dynamic memory allocation.

As shown in **Table 3**, dynamic memory must be manually allocated and freed by the programmer within the code. Not doing this will result in memory leeks, if this is forgotten in a loop, then all of the memory of the heap can be allocated. Whenever dynamic memory allocation is used, a conditional is required to asses if there is any space left on the heap before allocation. Some example code using dynamic memory allocation is shown below **Figure 29**.

```c
1   #include <stdio.h>
2   #include <stdlib.h> /* provides malloc */
3
4   int main(void) {
5       int *pi;
6
7       pi = (int *)malloc(sizeof(int));
8       if (pi == NULL) {
9           printf("ERROR: Out of memory\n");
10          return 1;
11      }
12
13      *pi = 5;
14      printf("%d\n", *pi);
15
16      free((void *)pi);
17
18      return 0;
19  }
```

```
C:\.\dynamic_memory.exe
5
```

Figure 29: C code which utilizes dynamic memory allocation.

Note that on line 7 of **Figure 29**, the type of pointer for the **`malloc`** command must be casted, that is because **`malloc`** on its own returns a void pointer type.

10. Lecture 10

10.1. Intro to Structs

Often there is a need to group pieces of data together which may not have the same datatype. In this case a struct can be used to group variables together, a basic example of a struct is shown in **Figure 30**.

```c
#include <stdio.h>

struct person {
    int age;
    double height;
    double weight;
};

int main(void) {
    struct person BillyBob = {34, 1.93, 85.0};
    return 0;
}
```

Figure 30: C code which depicts a basic struct.

10.2. Optimizing Struct Memory Usage

Struct memory usage can be optimizes by ordering the variables in increasing levels of alignment. On a 64bit system, 8 bytes are used per memory address. This means depending on the ordering of the struct, the total memory used can be different. Take the following example struct shown in **Figure 31**.

```c
struct bad {
    char c;     /* 1 byte */
    double d;   /* 8 byte */
    int i;      /* 4 byte*/
};              /* Total struct size: 24 byte */
```

Figure 31: C code which depicts a bad struct.

Because **`char`** is at the top of the struct it is assigned first. **`double`** is assigned next but, because it is bigger than the remaining bytes on that line it is assigned on a new line of memory. **`int`** is then next assigned on another new line, meaning three total lines of memory are used, adding up to 24 bytes. An optimized struct is shown in **Figure 32**.

```c
struct good {
    double d;   /* 8 byte */
    int i;      /* 4 byte*/
    char c;     /* 1 byte */
};              /* Total struct size: 16 byte */
```

Figure 32: C code which depicts a good struct.

In **Figure 32**, the variables are ordered from most to least alignment, meaning that the **`char`** and **`int`** can be on the same line of memory, meaning this struct takes up 8 less bytes of memory.

10.3. Passing Structs into Functions and Accessing Members

Specific variables of a struct can be accessed using **`.<member-name>`** after the name of the struct. Note that if this is then printed, the **correct formatter** must also be used. An example of this is shown in **Figure 33**.

```c
#include <stdio.h>

struct person {
    int age;
    double height;
    double weight;
    char name[10];
};

void printDetails(struct person A) {
    printf("The age is %d.\n", A.age);
    printf("The height is %.1f.\n", A.height);
    printf("The weight is %.1f.\n", A.weight);
    printf("The name is %s.\n", A.name);
}

int main(void) {
    struct person BillyBob = {34, 1.93, 85.0, "BillyBob"};
    struct person Lolipop = {10, 1.5, 40, "Lolypop"};

    printDetails(BillyBob);
    printf("\n");
    printDetails(Lolipop);

    return 0;
}
```

```shell
C:\.\pass_struct_function
The age is 34.
The height is 1.9.
The weight is 85.0.
The name is BillyBob.

The age is 10.
The height is 1.5.
The weight is 40.0.
The name is Lolypop
```

Figure 33: C code which passes a struct into a function and access its members.

10.4. Custom Struct Functions

It is possible to define complex behavior for structs via object-level programming. An example where a struct is defined for a complex number and then a custom add function is then defined is shown in **Figure 34**.

```c
#include <stdio.h>

/* structure declaration */
struct complex {
    double re;
    double im;
};

/* definition of function add */
struct complex add(struct complex a, struct complex b) {
    struct complex c;

    c.re = a.re + b.re;
    c.im = a.im + b.im;

    return c;
}

int main(void) {
    struct complex v = {1.0, 0.0}; /* initialise structures by providing */
    struct complex u = {0.0, 1.0}; /* a list of initialisers */
    struct complex w;

    w = add(v, u); /* sum v and u, return the sum as w */

    printf("%f + %fj\n", w.re, w.im);

    return 0;
}
```

```
C:\.\custom_struct_functio
1.000000 + 1.000000j
```

Figure 34: C code which defines a custom struct function

10.5. Using `typedef`

The custom struct function defined in **Figure 34**, the syntax is a bit cumbersome to use. To combat this, the **`typedef`** function can be used instead, the same struct and function that were in **Figure 34** are defined using **`typedef`** in **Figure 35**.

```c
#include <stdio.h>

/* structure declaration using typedef */
typedef struct {
    double re;
    double im;
```

```
C:\.\using_typedef.exe
1.000000 + 1.000000j
```

```c
} complex;

/* definition of function add */
complex add(complex a, complex b) {
    complex c;
    c.re = a.re + b.re;
    c.im = a.im + b.im;

    return c;
}

int main(void) {

    complex v = {1.0, 0.0}; /* initialise structures by providing */
    complex u = {0.0, 1.0}; /* a list of initialisers */
    complex w;

    w = add(v, u); /* sum v and u and return as w */

    printf("%f + %fj\n", w.re, w.im);

    return 0;
}
```

Figure 35: C code which defines a custom struct function using **`typedef`**.

11. Lecture 11

START OF WEEK 6

11.1. Arrays of Structs

TYo initialize an array of structs, a static array can be defined with the keyword before it being the name of the struct. An example of this is shown in **Figure 36** on line 24.

```c
1  #include <stdio.h>
2  
3  #define N 3
4  
5  /* structure declaration */
6  typedef struct {
7      double re;
8      double im;
9  } complex;
10 
11 /* definition of function add */
12 complex add(complex a, complex b) {
13     complex c;
14     c.re = a.re + b.re;
15     c.im = a.im + b.im;
16 
17     return c;
18 }
19 
20 int main(void) {
21     int i;
22 
23     /* initialise array of structures (three complex numbers): */
24     complex a[N] = {{1.0, 0.0}, {0.0, -1.0}, {-1.0, 1.0}};
25     complex s = {0.0, 0.0}; /* zero complex number */
26 
27     for(i = 0; i < N; i++) { /* add complex numbers */
28         s = add(s, a[i]);
29     }
30 
31     printf("%f + %fj\n", s.re, s.im);
32 
33     return 0;
34 }
```

```shell
C:\>.\struct_array.exe
0.000000 + 0.000000j
```

Figure 36: C code which uses an array of structs.

11.2. Structures and Pointers

The members of a structure can be accessed using a pointer to a structure. This is shown in **Figure 37** in 2 separate ways, one using a more direct syntax and one using a simplified syntax using the **->** operator for C.

```c
#include <stdio.h>

/* structure declaration */
typedef struct {
    double re;
    double im;
} complex;

int main(void) {
    complex v = {0.0, 1.0};
    complex *u = &v;
    /* declaration and definition of a pointer to a structure */

    printf("1: %f + %fj\n", v.re, v.im);
    printf("2: %f + %fj\n", (*u).re, (*u).im); /* dereferencing */
    printf("3: %f + %fj\n", u->re, u->im);    /* dereferencing using new syntax */

    return 0;
}
```

```
C:\>.\struct_pointers.exe
1: 0.000000 + 1.000000j
2: 0.000000 + 1.000000j
3: 0.000000 + 1.000000
```

Figure 37: C code which access members of a structure using a pointer.

11.3. Structs, Pointers and Functions

In lecture 10, the process of passing structures into functions was shown. However, that method passes a copy of the variable into the function, which is highly inefficient. Instead, the address of a struct can be used, negating the need for a copy, this is shown in **Figure 38**.

```c
#include <stdio.h>

/* structure declaration */
typedef struct {
    double re;
    double im;
} complex;

/* definition of function add */
complex add(complex *a, complex *b) {
```

```
C:\>.\struct_functions.exe
1.000000 + 1.000000
```

```c
    complex c;
    c.re = a->re + b->re;
    c.im = a->im + b->im;

    return c;
}

int main(void) {
    complex v = {0.0, 1.0}; /* initialise structures by providing */
    complex u = {1.0, 0.0}; /* a list of initialisers */
    complex w;

    w = add(&v, &u); /* sum v and u (passed as addresses), return as w */

    printf("%f + %fj\n", w.re, w.im);

    return 0;
}
```

Figure 38: C code passes the memory address of a pointer into a function, instead of a copy of the variable.

11.4. Dynamic Memory Allocation with Structs

Dynamic memory allocation using **`malloc`** and **`sizeof`** can be used with structs, an example of this is shown in line 27 of **Figure 39**.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* structure declaration */
5  typedef struct {
6      double re;
7      double im;
8  } complex;
9
10 /* definition of function add */
11 complex add(complex *a, complex *b) {
12     complex c;
13     c.re = a->re + b->re;
14     c.im = a->im + b->im;
15
16     return c;
17 }
```

```
C:\>.\struct_dynamic.exe
a) 1.000000 + 1.000000j
b) 1.000000 + 1.000000j
```

```c

int main(void) {
    complex v = {0.0, 1.0}; /* initialise structures by providing */
    complex u = {1.0, 0.0}; /* a list of initialisers */
    complex w, *z;

    w = add(&v, &u); /* sum v and u (passed as pointers), return as w */
    printf("a) %f + %fj\n", w.re, w.im);

    z = (complex *) malloc(sizeof(complex)); /* dynamic memory allocation */
    *z = add(&v, &u); /* same as above but use a pointer */
    printf("b) %f + %fj\n", z->re, z->im);

    free((void *) z);

    return 0;
}
```

Figure 39: C code passes that uses dynamic memory allocation with structs.

11.5. Pointers as Struct Members

The members of structures can themselves be initialized as pointers, allowing for the call of that member to itself be a pointer. This allows for pointer arithmetic to be done, this can also be combined with dynamic memory allocation. This is shown in **Figure 40**.

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    double *re;
    double *im;
} complex;

int main(void) {
    complex a, b;
    double x, y;

    x = 1.0;
    y = 2.0;

    /* We can treat the structure members as normal pointers. */
```

```
C:\>.\struct_pointer_member
Complex number is: 1.00 + 2.00j
Complex number is: 1.00 + -1.00j
Complex number is: 2.00 + -2.00j
Complex number is: 1.00 + -1.00j
Complex number is: 2.00 + -2.00j
```

```c
    a.re = &x;
    a.im = &y;
    printf("Complex number is: %4.2lf + %4.2lfj\n", *a.re, *a.im);

    /* Now set up the structure member arrays using
       dynamic memory allocation. */
    b.re = (double *) malloc(2 * sizeof(double));
    b.im = (double *) malloc(2 * sizeof(double));

    /* Populate the arrays. */
    b.re[0] = 1.0;
    b.re[1] = 2.0;

    *b.im = -1.0;        /* can also use pointer arithmetic */
    *(b.im + 1) = -2.0;

    printf("Complex number is: %4.2lf + %4.2lfj\n", b.re[0], b.im[0]);
    printf("Complex number is: %4.2lf + %4.2lfj\n", b.re[1], b.im[1]);
    /* Same but now using pointer arithmetic. */
    printf("Complex number is: %4.2lf + %4.2lfj\n", *b.re, *b.im);
    printf("Complex number is: %4.2lf + %4.2lfj\n", *(b.re + 1), *(b.im + 1));

    free((void *) b.re);
    free((void *) b.im);

    return 0;
}
```

Figure 40: C code which uses pointers members within a struct, as well as using dynamic memory allocation.

12. Lecture 12

12.1. Pointers to a Structure with Pointer Members

If there is a structure where in the definition of the structure there is a pointer, then a pointer pointing to the pointer member needs to be handled with careful syntax. This is shown in **Figure 41** on lines 25 and 28.

```c
1  #include <stdio.h>
2  #include <stdlib.h>
3  
4  typedef struct {
5      double *re;
6      double *im;
7  } complex;
8  
9  int main(void) {
10     complex a;
11     complex *b;
12     double x, y;
13 
14     x = 1.0;
15     y = 2.0;
16 
17     a.re = &x;
18     a.im = &y;
19 
20     printf("Complex number is %f + %fj\n", *a.re,
       *a.im);
21 
22     /* Make b point to a. */
23     b = &a;
24 
25     printf("Complex number is %f + %fj\n", *((*b).re),
       *((*b).im));
26 
27     /* Using reference operator: */
28     printf("Complex number is %f + %fj\n", *(b->re),
       *(b->im));
29 
30     return 0;
31 }
```

```
C:\>.\pointing_pointer_memb
Complex number is 1.000000 + 2.000000j
Complex number is 1.000000 + 2.000000j
Complex number is 1.000000 + 2.000000j
```

Figure 41: C code which uses a pointer, pointing to a member of a structure which is also a pointer.

12.2. Using `fscanf` to Read Data From Files

As was shown in previous lectures the **`scanf`** function can be used to read a data input from the console. Though redirection (**`<`**) can be used to feed a file into a function, the **`fscanf`** function is a more robust form of this, shown in **Figure 42**.

```c
#include <stdio.h>

int main(void)
{
    FILE *f; /* pointer to file */

    int s;
    double x, y, z;

    if ((f = fopen("numbers.txt", "r")) == NULL) {
        printf("Cannot open 'numbers.txt' for reading");
        return -1;
    }

    s = fscanf(f, "%lf %lf %lf", &x, &y, &z);
    printf("(%d) %lf, %lf, %lf\n", s, x, y, z);

    if (fclose(f) != 0) {
        printf("File could not be closed.\n");
        return -1;
    }

    return 0;
}
```

```txt
10 12 13
1 3 4
3 4 5
4 5 100
```

```shell
C:\>.\using_fscanf.exe
(3) 10.000000, 12.000000, 13.000000
```

Figure 42: C code which uses **`fscanf`** to read data from a file.

13. Lecture 13

START OF WEEK 7

13.1. Using `fscanf` to Read Multiple Lines From Files

Note that the method shown in **Figure 42** only returns the first line of the text file. To print the entire contents of the file a while loop can be used, this example is shown in **Figure 43**.

C

```c
#include <stdio.h>

int main(void)
{
    FILE *f; /* pointer to file */

    double x, y, z;

    if ((f = fopen("numbers.txt", "r")) == NULL) {
        printf("Cannot open 'numbers.txt' for
        reading");
        return -1;
    }

    while (fscanf(f, "%lf %lf %lf", &x, &y, &z) == 3)
    {
        printf("%lf, %lf, %lf\n", x, y, z);
    }

    if (fclose(f) != 0) {
        printf("File could not be closed.\n");
        return -1;
    }

    return 0;
}
```

txt

```
10 12 13
1 3 4
3 4 5
4 5 100
```

Shell

```
C:\>.
\fscanf_lines.exe
10.000000, 12.000000,
13.000000
1.000000, 3.000000,
4.000000
3.000000, 4.000000,
5.000000
4.000000, 5.000000,
100.00000
```

Figure 43: C code which uses **`fscanf`** to read every data line from a file.

13.2. Reading Every Character From a File

In previous lectures, the **`getchar`** command was used in conjunction with a while loop to extract every input character from the user. In a similar way, the **`fgetc`** command can be used to extract every character from a file, this is shown in **Figure 44**. Note that this will require further processing of the data to remove white space and new line characters.

C

```c
#include <stdio.h>

int main(void)
{
```

txt

```
10 12 13
1 3 4
3 4 5
4 5 100
```

```c
    FILE *f; /* pointer to file */
    char c;

    if ((f = fopen("numbers.txt", "r")) == NULL) {
        printf("Cannot open 'myfile.txt' for reading");
        return -1;
    }

    /* read file content char by char and print to stdout */
    while ((c = fgetc(f)) != EOF) {
        printf("%c", c);
    }

    if (fclose(f) != 0) {
        printf("File could not be closed.\n");
        return -1;
    }

    return 0;
}
```

Shell

```
C:\>.\fgetc.exe
10 12 13
1 3 4
3 4 5
4 5 100
```

Figure 44: C code which uses **fgetc** to read every character from a file.

13.3. Writing to Files

Instead of using the **"r"** flag for the **fopen** command, the **"a"** and **"w"** flags can be used to append and write to a file. Not that the **"w"** flag will overwrite an already existing filename. An example of this is shown in **Figure 45**.

C

```c
#include <stdio.h>

int main(void)
{
    FILE *f; /* pointer to file */

    if ((f = fopen("myfile.txt", "w")) == NULL) {
        printf("Cannot open 'myfile.txt' for writing");
        return -1;
    }

    fprintf(f, "We can now print to the file f using fprintf.\n");
    fprintf(f, "For example, we print a number %d and %d and %d.", 1, 2, 42);

```

Shell

```
C:\>type myfile.txt
We can now print to the file f using fprintf.
For example, we print a number 1 and 2 and 42
```

```c
    if (fclose(f) != 0) {
        printf("File could not be closed.\n");
        return -1;
    }

    return 0;
}
```

Figure 45: C code which uses the **`w`** flag to write into a file.

13.4. Reading Command Line Arguments

At runtime, command line arguments can be passed into the main function of the compiled program. This is done through the **`argc`** and **`argv`** function variables. These are:

- **`argc`**: Argument Count
  - An integer value for the number of command line arguments that have been passed.
  - Note that if none are passed in, then **`argc = 1`** as the name of the program is the first argument and is always passed.
- **`argv`**: Argument Vector
  - A vector of all of the command line arguments that have been passed into the function.
  - **`argv[0]`** is the program name and **`argv[1] -> argv[argc-1]`** are the command line arguments that have been passed in.

An example of a program which makes use of command line arguments is shown in **Figure 46**.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;

    for (i = 0; i < argc; i++) {
        printf("Argument %d = '%s'\n", i, argv[i]);
    }

    return 0;
}
```

```shell
C:\>.\command_args.exe arg1 arg2 arg3
Argument 0 = '.\command_args.exe'
Argument 1 = 'arg1'
Argument 2 = 'arg2'
Argument 3 = 'arg3'
```

Figure 46: C code which uses **`argc`** and **`argv`** to pass in command line arguments.

13.5. Operator Preference

Certain commands are done before others, the order of how commands are computed is shown in **Table 4**. Note that the address operator is done before the pointer content operator. Furthermore to fine tune the running of command, brackets can be used as they are compiled first.

Operator Type	Operators	Associativity
Postfix	(), [], ->, ., expr++, expr–	Left to Right
Unary	++expr, –expr, +, -, !, , &, *, sizeof, (type)	Right to Left
Multiplicative	*, /, %	Left to Right
Additive	+, -	Left to Right
Shift	<<, >>	Left to Right
Relational	<, <=, >, >=	Left to Right
Equality	==, !=	Left to Right
Bitwise AND	&	Left to Right
Bitwise XOR	^	Left to Right
Bitwise OR	|	Left to Right
Logical AND	&&	Left to Right
Logical OR	||	Left to Right
Conditional	?:	Right to Left
Assignment	=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=	Right to Left
Comma	,	Left to Right

Table 4: Order of preference for operators in C.

13.6. 2D Arrays

2D arrays are an array of an array, essentially a table of data with rows and columns. The syntax for defining a 2D array is **`data_type array_name[rows][columns]`**. If allocation and initialization are done at the same time **the number of rows** can be inferred for example **`int a[][3] = {{1, 2, 3},{4, 5, 6}};`**.

13.7. 2D Matrix Pointer Arithmetic

Pointer arithmetic for 2D arrays is slightly more complex than 1D arrays. The key rules for a generic 2D array called matrix are:

- **`matrix + i`**: Pointer to the i-th row.
- **`matrix[i]`**: Pointer to the first element in the i-th row.
- **`*(matrix + i)`**: Now equivalent to **`matrix[i]`** as its dereferenced.
- **`*(matrix + i) + j`**: Points to the j-th element of the i-th row and is equivalent to the following:
  - **`&matrix[i][j]`**
  - **`matrix[i] + j`**
- **`*(*(matrix + i) + j)`**: Dereferences the pointer retrieving the element in in **`matrix[i][j]`**.

14. Lecture 14

14.1. 2D Array Dynamic Memory Allocation

Dynamic memory allocation can be done for 2D arrays by defining array of pointers. These can then be used to define a row within the 2D array, an example of this is shown in **Figure 47**.

```c
#include <stdio.h>
#include <stdlib.h>

#define N 3
#define M 4

int main(void) {
    int i, j, *m[N];

    /* Dynamically allocate memory for each row */
    m[0] = (int *) malloc(M * sizeof(int));
    m[1] = (int *) malloc(M * sizeof(int));
    m[2] = (int *) malloc(M * sizeof(int));

    /* Check if memory allocation was successful */
    if (m[0] == NULL || m[1] == NULL || m[2] == NULL)
    {
        printf("Memory allocation failed.\n");
        return 1; /* Exit if any allocation fails */
    }

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            m[i][j] = i * 10 + j; /* initialise */
        }
    }

    printf("2D array m:\n");
    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            printf("%d ", m[i][j]);
        }
        printf("\n");
    }

    /* Free the allocated memory for each row */
    free(m[0]);
    free(m[1]);
    free(m[2]);
```

```
C:\>.\dynamic_2d_array.exe
2D array m:
0 1 2 3
10 11 12 13
20 21 22 23
```

```c

    return 0;
}
```

Figure 47: C code which uses dynamic memory allocation to create a 2D array.

14.2. Dynamic Row Sizes and Jagged Arrays

In some applications, the number of elements per row needs to be different per array. In this case, the definition used in **Figure 47** is not sufficient. For examples like this the double pointer ** is used, an example of this is shown in **Figure 48** for a jagged or triangular array.

```c
#include <stdio.h>
#include <stdlib.h>

#define N 3

int main(void) {
    int **m;
    int i;

    /* Allocate memory for the array of pointers (rows) */
    m = (int **) malloc(N * sizeof(int *));
    if (m == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    /* Dynamically allocate memory for each row with varying sizes */
    m[0] = (int *) malloc(3 * sizeof(int)); /* 3 elements in row 0 */
    m[1] = (int *) malloc(2 * sizeof(int)); /* 2 elements in row 1 */
    m[2] = (int *) malloc(1 * sizeof(int)); /* 1 element in row 2 */

    /* Check if memory allocation was successful for each row */
    if (m[0] == NULL || m[1] == NULL || m[2] == NULL) {
        printf("Memory allocation for rows failed.\n");
        free(m);
        return 1; /* Exit if any allocation fails */
    }

```

```shell
C:\>.\dynamic_2d_jagged_arr
Jagged 2D array m:
0 1 2
10 11
20
```

```c
    /* Initialize each row with unique values */
    for (i = 0; i < 3; i++) {
        m[0][i] = i;
    }
    for (i = 0; i < 2; i++) {
        m[1][i] = i + 10;
    }
    for (i = 0; i < 1; i++) {
        m[2][i] = i + 20;
    }

    /* Print the jagged 2D array */
    printf("Jagged 2D array m:\n");
    for (i = 0; i < 3; i++) {
        printf("%d ", m[0][i]);
    }
    printf("\n");
    for (i = 0; i < 2; i++) {
        printf("%d ", m[1][i]);
    }
    printf("\n");
    for (i = 0; i < 1; i++) {
        printf("%d ", m[2][i]);
    }
    printf("\n");

    /* Free the allocated memory for each row */
    for (i = 0; i < 3; i++) {
        free(m[i]);
    }

    /* Free the memory allocated for the array of pointers */
    free(m);

    return 0;
}
```

Figure 48: C code which uses dynamic memory allocation to create a jagged 2D array.

15. Lecture 15

START OF WEEK 8

15.1. Linux or Unix Remote Servers

Typically remote servers are used for things such as websites, supercomputers and cloud computing. The default language for such servers is Linux or Unix as:

- The dominant server and supercomputer OS
- Can do everything from the command line
- Efficient package managers for software updates and installation.

16. Lecture 16

16.1. Shell Scripts

Shell scripts are a chain of shell commands and are useful for automating tasks (creating/copying/moving files), simple data processing and running programs sequentially. They are scalable and can have command line arguments easily passed to them.

16.2. Make Files

Are primarily used to build a set of compiled files if there is a change within one of the files. It does this by comparing the timestep of different files within the program. They can also be used to run tasks in parallel, which is particularly useful for parralizable tasks.

16.3. Shell Script Syntax

16.3.1. Variables

All of the common commands that can be ran on a linux or unix shell can be placed sequentially in a .sh file and ran. To define variables the syntax shown in **Figure 49** can be used.

```shell
NAME=world
echo "Hello double quoted $NAME"
echo 'Hello single quoted $NAME (Oops)'

FILE="image"
echo $FILE01.png "(broken)"
echo ${FILE}01.png "(works)"
```

Figure 49: Shell script which depicts how to define variables and use them.

Note that in **Figure 49** using double quoted strings will expand the variable whereas single will not. Using curly brackets will protect the variable and every variable is by default a string.

16.3.2. Command Line Arguments

Command line arguments can be passed into a shell script, these are numbered 0-n where 0 is the name of the program itself. An example of this is show in **Figure 50**.

```shell
  echo "command: $0" #Name of the command
```

```
echo "arg 1: $1" #1st arg
echo "arg 2: $2" #2nd arg
echo "arg 3: $3" #3rd arg
echo "all args: $@"
echo "number of args: $#"
```

Figure 50: Shell script which returns the command line arguments.

16.4. Dependency Graphs and Make Files

An intuitive way of thinking about a program is through the use of a flowchart, which defines the path taken between a source and a target. An example of a flowchart for a simple calculation and plotting program is shown in **Figure 51**.

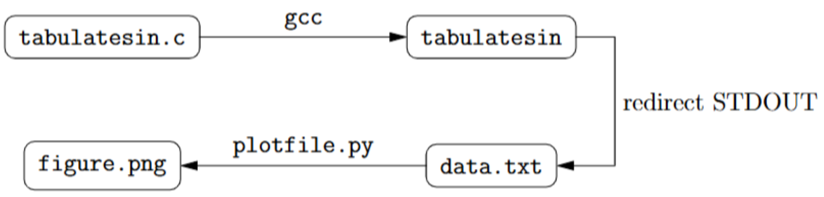


Figure 51: An example of a flowchart for a simple plotting program.

However, for make files it makes more sense to use a dependency graph, which clearly defines the dependencies between different sections of the program, shown in **Figure 52** for the flowchart that was shown in **Figure 51**.

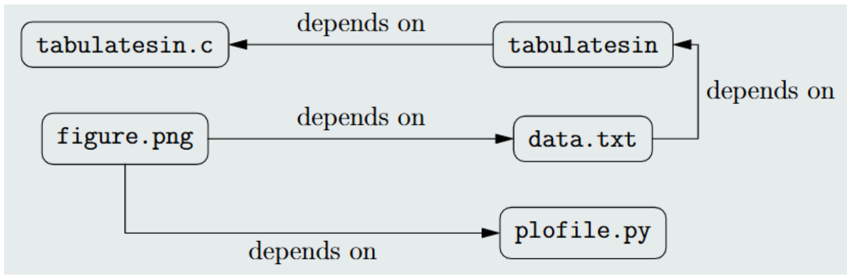


Figure 52: An example of a dependency graph for the program shown in **Figure 51**.

The basic syntax for a make file is shown in **Figure 53** with the target being the target file, the source being teh source file, the tab being a space and the command being teh command needed to generate the target file

```make
target1: source1 source2
[tab] command1a
[tab] command1b
target2: target1
[tab] command2
```

Figure 53: Basic syntax for a make file.

To create the target from the make file, the command **`make target`** is ran. As shown in **Figure 53**, multiple sources can be set to one target file. Now all the essential building blocks are defined the make file for the dependency graph shown in **Figure 52** can be created, this is shown in **Figure 54**.

```make
Figure.png: plotfile.py data.txt
	python plotfile.py data.txt

data.txt: tabulatesin
	./tabulatesin > data.txt

tabulatesin: tabulatesin.c
	gcc -Wall -ansi -pedantic -lm -o tabulatesin tabulatesin.c

clean:
	rm data.txt tabulatesin plotfile.py

plotfile.py:
	wget http://www.soton.ac.uk/~rpb/feeg6002/tools/plotfile.py
```

Figure 54: Makefile for the dependency graph shown in **Figure 52**.

17. Lecture 17

START OF WEEK 9

17.1. Returning 2D Arrays From Functions

As was seen before, it is not possible to directly return arrays from a C function. Instead a pointer to the array can be manipulated and returned. An example of this is shown in **Figure 55** which uses the double pointer syntax to define a 2d array within the function and return that.

```c
#include <stdio.h>
#include <stdlib.h>

double **zero(void)
{
    /* Function allocates memory for and returns 2x2
       matrix with elements initialised to 0. */

    double **m;

    m = (double **) malloc(2 * sizeof(double *)); /* Two double pointers (rows)
    */
    m[0] = (double *) malloc(2 * sizeof(double)); /* Two doubles in row 0 */
    m[1] = (double *) malloc(2 * sizeof(double)); /* Two doubles in row 1 */

    m[0][0] = 0.0;
    m[0][1] = 0.0;
    m[1][0] = 0.0;
    m[1][1] = 0.0;

    return m;
}

int main(void)
{
    double **m;

    m = zero();

    /* Free allocated memory */
    free((void *) m[0]);
    free((void *) m[1]);
    free((void **) m);

    return 0;
}
```

Figure 55: C code which returns a 2d array from a function.

17.2. Representing 2D arrays Using 1D Arrays

By default, when a 2D array is statically defined, the memory is stored contiguously. It is therefore possible to explicitly define a 2D array by defining 1D arrays next to one another, this is shown in **Figure 56** where both code blocks are identical.

```c
#include <stdio.h>
#define N 2
void print_array(int a[][N])
{
    int i, j;

    for (i = 0; i < N; i++) {
        printf("| ");
        for (j = 0; j < N; j++) {
            printf("%d ", a[i][j]);
        }
        printf("|\n");
    }
    printf("\n");
}

int main(void)
{
    int i, j, k;

    int m[N][N] = {{2, 1}, {3, 1}};
    int n[N][N] = {{1, 2}, {3, 2}};
    int o[N][N] = {{0, 0}, {0, 0}};

    /* multiply matrices m * n */
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                o[i][j] += m[i][k] * n[k][j];
            }
        }
    }

    print_array(o);
    return 0;
}
```

```c
#include <stdio.h>
#define N 2
void print_array(int a[])
{
    int i, j;

    for (i = 0; i < N; i++) {
        printf("| ");
        for (j = 0; j < N; j++) {
            printf("%d ", a[i * N + j]);
        }
        printf("|\n");
    }
    printf("\n");
}

int main(void)
{
    int i, j, k;

    int m[N * N] = {2, 1, 3, 1};
    int n[N * N] = {1, 2, 3, 2};
    int o[N * N] = {0, 0, 0, 0};

    /* multiply matrices m * n */
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                o[i * N + j] += m[i * N + k] * n[k * N + j];
            }
        }
    }

    print_array(o);
    return 0;
}
```

Figure 56: C code which defines a 2D array using common syntax [left] and 1d arrays [right]

18. Lecture 18

18.1. Defining Pointers to a Function

A function is stored in memory in the text or code segment not in a heap or stack. It is possible to define a pointer to a function, which will be useful for passing functions into other functions. The syntax to define a pointer to function is shown in **Figure 57**. Note that the second syntax is the more commonly used one.

```c
#include <stdio.h>

/* A normal function with an int parameter
   and void return type */
void fun(int a) {
    printf("Value of a is %d\n", a);
}

int main(void) {
    /* fun_ptr is a pointer to function fun() */
    void (*fun_ptr)(int) = &fun;

    /* The above line is equivalent to
       void (*fun_ptr)(int);
       fun_ptr = &fun; */

    /* Invoking fun() using fun_ptr */
    (*fun_ptr)(10);

  /* Equivelent and Simpler Syntax shown below*/
  void (*fun_ptr)(int) = fun; /* & removed */

  fun_ptr(10); /* * removed */

    return 0;
}
```

Figure 57: C code depicts how function pointers can be defined.

18.2. Defining Arrays of Function Pointers

Given that function pointers have the same function prototype they can be used interchangeably. An example of this is shown in **Figure 58**, where each function has the same prototype of two inputs.

```c
#include <stdio.h>

void add(int a, int b) {
    printf("Addition is %d\n", a + b);
}

void subtract(int a, int b) {
    printf("Subtraction is %d\n", a - b);
}

void multiply(int a, int b) {
    printf("Multiplication is %d\n", a * b);
}

int main(void) {
    /* fun_ptr_arr is an array of function pointers */
    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int ch, a = 15, b = 10;

    printf("Enter Choice: 0 for add, 1 for subtract and 2 "
           "for multiply\n");
    scanf("%d", &ch);

    if (ch > 2) return 0;

    fun_ptr_arr[ch](a, b);
    /* Equivalent to: */
    /* (*fun_ptr_arr[ch])(a, b); */

    return 0;
}
```

Figure 58: C code which uses an array of function pointers all with the same prototype.

18.3. Passing Functions as Function inputs

The key use of function pointers is to use them as inputs to other functions. Typically this is used to pass a numerical model into a solver function. An example of this is shown in **Figure 59**.

```c
  double integrate(double (*f)(double), double a, double b)
  {return 0.5 * (f(a) + f(b)) * (b - a);}

  double square(double x)
  {return x * x;
}

  int main(void)
  {
      double result = integrate(square, 0.0, 2.0);
      printf("%f\n", result);
}
```

Figure 59: C code which shows how a function pointer can be passed into a function.

18.4. Euler Method

The Euler method is a simple numerical method that can be used to solve ordinary differential equations. The equation for which is shown in **Eq. 3**.

$$y_{i+1} \approx y_i + \Delta t f(x_i, y_i) \tag{3}$$

Where $t_i = t_0 + i\Delta t$, essentially propagating a numerical approximation of the ODE given some initial conditions. This can be programmed in C in the following way using function pointers, shown in **Figure 60**.

```c
/* Function euler implements Euler method for solving first
   order ODEs dy/dt = f(t, y). Input parameters are:

   t  - independent variable (time) given as 1D array
   y  - dependent variable given as 1D array
   y0 - initial condition
   f  - pointer to a function to be integrated
   n  - the length of arrays t and y (must be equal length)

   The solution is stored in the array y. */

void euler(double *t, double *y, double y0,
           double (*f)(double, double), int n)
{
    int i;
    float dt = t[1] - t[0]; /* step */

    y[0] = y0; /* initial condition */
    for (i = 0; i < n; i++) {
        y[i + 1] = y[i] + dt * f(t[i], y[i]);
    }
}
```

Figure 60: C code which shows how Euler method can be implemented in C.

18.5. 2nd Order Runge-Kutta Method

Runge-Kutta methods are a collection of numerical methods for solving ODEs. They are more accurate than Euler's method but are more computationally expensive, requiring more computations. The equation for the second order Runge-Kutta method is shown in **Eq. 4**.

$$k_1 = \Delta \cdot f(t_i, y_i) \tag{4.1}$$

$$k_2 = \Delta \cdot f\left(t_i + \frac{1}{2}\Delta t, y_i + \frac{1}{2}\Delta t\right) \tag{4.2}$$

$$y_{i+1} \approx y_i + k_2 \tag{4.3}$$

An implementation of this method in C is shown in **Figure 61**.

```c
/* Function rk implements 2nd order Runge-Kutta method for
   solving first order ODEs dy/dt = f(t, y). Input parameters:

   t  - independent variable (time) given as 1D array
   y  - dependent variable given as 1D array
   y0 - initial condition
   f  - pointer to a function to be integrated
   n  - the length of arrays t and y (must be equal length)

   The solution is stored in the array y. */

void rk2(double *t, double *y, double y0,
         double (*f)(double, double), int n)
{
    int i;
    float dt = t[1] - t[0]; /* step */

    float k1, k2, k3, k4;

    y[0] = y0; /* initial condition */
    for (i = 0; i < n; i++) {
        k1 = dt * f(t[i], y[i]);
        k2 = dt * f(t[i] + dt / 2.0, y[i] + k1 / 2.0);

        y[i + 1] = y[i] + k2;
    }
}
```

Figure 61: C code which shows how RK2 method can be implemented in C.