More Bash

Dr. Gareth Roy (x6439)
gareth.roy@glasgow.ac.uk

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

User Input

- Instead of getting input via passed arguments we can also get it interactively.
- **read** can by used to interactively get input from the User.
- read can take a variable to store or if no variable is specified it defaults to **$REPLY**
- Useful flags that can be passed are:
  - **-p** - specifies a prompt.
  - **-s** - specifies that input is not printed to screen
  - **-n** - specifies the number of characters to read.
- **Note:** It's usually a good idea to check input is present and well formed before using it.

1. gareth@brutha: ~/Lec05 (ssh)

```
#!/bin/bash

# read can be used to get input from the user and store
# it in a variable
echo "Type a word: "
read WORD

# test to see if we catually got a word.
if [ -z ${WORD} ]; then
        echo "No really, you need to type in a word!"
        exit 1
else
        echo "Your word was - ${WORD}."
fi

# specifying -s means any characters typed won't be echoed
# to the screen.
echo "Type a secret word!!!!: "
read -s SECRET
echo "Your secret was - ${SECRET}."


# If we don't specify a variable read defaults to $REPLY
# -p allows us to passs a string as a prompt for input.
while true
do
        read -p "Do you want to exit? [y/n]: "
        if [ ${REPLY} == "y" ]; then
                break
        elif [ ${REPLY} == "n" ]; then
                echo "Continuing on..."
        else
                echo "${REPLY} not an understood answer."
        fi

done
```

All

2. gareth@brutha: ~/Lec05 (ssh)

```
gareth@brutha:~/Lec05$ ./simple.sh
Type a word:
Hello
Your word was - Hello.
Type a secret word!!!!:
Your secret was - Sssssh.
Do you want to exit? [y/n]: n
Continuing on...
Do you want to exit? [y/n]: a
a not an understood answer.
Do you want to exit? [y/n]: y
gareth@brutha:~/Lec05$
```

File Input

- **read** can also be used to read files.
- Often you want to be able to read a particular file line by line and act based on the contents of each line.
- Combining **read** which a **while** loop allows us to do this.
- We create a while loop, whose conditional is to read some data.
- While there is data to read the loop will continue. When all the data has been read, the loop will terminate.
- The data can either be **piped (|)**, or **redirected (<)** into the while loop.

1. gareth@brutha: ~/Lec05 (ssh)

```bash
#!/bin/bash

# Check to see if we've been given a input file
# We could also check $# > 0
if [ -z $1 ]; then
        echo "Please specify an input file!"
        exit 1
fi

# Assume that a file is passed as the first argument to the script
# The data can be sent to the while loop either by using a pipe
cat $1 | while read LINE
do
        echo "1: ${LINE}"
done

# Or by redirecting the the contents of the file to STDIN
while read LINE
do
        echo "2: ${LINE}"
done < $1

exit 0
~
~
```

2. gareth@brutha: ~/Lec05 (ssh)

```
gareth@brutha:~/Lec05$ cat test_file
10 rations of food
2 blue potions
1 scrol entitiles "exyc ahks"
gareth@brutha:~/Lec05$ ./simple2.sh test_file
1: 10 rations of food
1: 2 blue potions
1: 1 scrol entitiles "exyc ahks"
2: 10 rations of food
2: 2 blue potions
2: 1 scrol entitiles "exyc ahks"
gareth@brutha:~/Lec05$
```

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

Declaring Functions

- A function in Bash consists of a label and a block of code (denoted by {} brackets).
- Functions must be declared before they are used.
- Functions can be declared in two ways:
    - using the **function** keyword and supplying a name, and code block
    - supplying a name filled by **()**, along with a code block.
- Functions are called in the same ways as command, by simply writing their name
- As with C functions are a good way to be able to reuse pieces of code without having to copy and paste.

```
function name {
    commands
    commands
}

name
```

```
name() {
    commands
    commands
}

name
```

Declaring Functions

- In this example we declare two functions:
    - **hello**
    - **world**
- The hello function echo's a string without appending a newline (**-n**)
- The world function simply echo's a string.
- In the main body of the script we use a for loop, to loop 5 times.
- During each loop we call the hello and world functions.

1. gareth@brutha: ~/Lec05 (ssh)

```bash
#!/bin/bash

# Function definitions happen before the are used
# They have a label, and a body denoted by {}
# They can be declared using the function keyword
function hello {
        echo -n "Hello, "
}

# Or by appending () to the label of the function
world() {
        echo "World! (well P2T)"
}

# The main body of the script
# Will loop 5 times, and call the helloworld five times
for i in {1..5}
do
        hello
        world
done

exit 0

~
```

2. gareth@brutha: ~/Lec05 (ssh)

```
gareth@brutha:~/Lec05$ ./simple3.sh
Hello, World! (well P2T)
Hello, World! (well P2T)
Hello, World! (well P2T)
Hello, World! (well P2T)
Hello, World! (well P2T)
gareth@brutha:~/Lec05$
```

All

Passing Arguments

- Passing parameters to functions in Bash is similar to passing arguments to the script.
- When the function is called Bash passes a list of arguments to it.
- **$@** - can be used to see all of the arguments passed to the function.
- **$#** - is the number of arguments passed to the funciont.
- **$1** - is the first argument passed to the function.
- **$2** - is the second argument passed to the function, and so on.
- If there are a large number of parameters passed to the function, **shift** can be used to cycle through them.

```
function name {
    commands
    commands
}

name arg1 arg2 arg3
```

```
name() {
    commands
    commands
}

name arg1 arg2 arg3
```

Function Arguments

- In the example shown we test the use of passing arguments.
- In the first example we see the use of **$#**, **$@** and **$1**.
- In the second example we use **shift** to cycle through the arguments.
- <u>**Note:**</u> each time we call **shift** we discard the first argument from the list one space to the left.
- Using shift is destructive so if you use this method you must store each value you are interested.

1. gareth@brutha: ~/Lec05 (ssh)

```bash
#!/bin/bash

# Arguments can be accessed using the same parameters as
# the Bash script itself, i.e. $1 represents the first
# argument passed to the function
function test_args {
        echo "I was called with $# arguments"
        echo "The arguments were $@"
        echo "The first argument was $1"
        echo
}

# Functions are called in the same way as commands
# Arguments are listed after the functin name
test_args a b c d

# We can access all the arguments by using a shift function
# shift effectively removes the first element and "shifts"
# the other elements left.
function shift_args {
        until [ -z $1 ]
        do
                echo "There are $# args - $@"
                echo $1
                shift
        done
}

shift_args a b c d
```

2. gareth@brutha: ~/Lec05 (ssh)

```
gareth@brutha:~/Lec05$ ./simple4.sh
I was called with 4 arguments
The arguments were a b c d
The first argument was a

There are 4 args - a b c d
a
There are 3 args - b c d
b
There are 2 args - c d
c
There are 1 args - d
d
gareth@brutha:~/Lec05$
```

Global & Local Variables

- In Bash all variables are global within the running script.
- This means variables declared in functions are available outside of the functions calls.
- If this is not desirable behaviour Bash provides the **local** keyword that allows us to restrict the scope of a variable.
- **<u>Note:</u>** Variables declared outside of a function are also available within a Bash function.

1. gareth@brutha: ~/Lec05 (ssh)

```bash
#!/bin/bash

# By default all variables declared in a Bash script have file
# scope, or in other words are declared a GLOBAL Variables
GLOBALVAR="This is a GLOBALVAR"

# We can specify a variable should be local to a function by
# using the local keyword
function variables {

        MYGLOBALVAR="This is also a GLOBALVAR"
        local MYVAR="This is a local VAR"

        echo "Function: ${GLOBALVAR}"
        echo "Function: ${MYGLOBALVAR}"
        echo "Function: ${MYVAR}"
}


echo "Main before: ${GLOBALVAR}"
echo "Main before: ${MYGLOBALVAR}"
echo "Main before: ${MYVAR}"

variables

echo "Main after: ${GLOBALVAR}"
echo "Main after: ${MYGLOBALVAR}"
echo "Main after: ${MYVAR}"

exit 0
```

2. gareth@brutha: ~/Lec05 (ssh)

```
gareth@brutha:~/Lec05$ ./simple5.sh
Main before: This is a GLOBALVAR
Main before:
Main before:
Function: This is a GLOBALVAR
Function: This is also a GLOBALVAR
Function: This is a local VAR
Main after: This is a GLOBALVAR
Main after: This is also a GLOBALVAR
Main after:
gareth@brutha:~/Lec05$
```

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Brian Kernighan

Good Practice

- ALWAYS use comments, you should include at least a single line comment telling what the script does.
- Use white space to separate ideas, it will make it easier to understand each section of code.
- Use Quoted Variables, i.e. **“$MYVAR”**, this will get around problems with Bash “word splitting”.
- Always use exit statements, and test sub-commands return codes. Bash has a tendency to fail quietly.

2. gareth@brutha: ~/Lec05 (ssh)

```bash
#!/bin/bash

# A single or multi-line comment telling the person
# reading the code what the script is supposed to do.

function usage {
        echo "This script prints out a given file line by line"
        echo "It requires a single argument whcih is the file"
        echo "to read."
        echo
        echo "simple6.sh <filename>"
}

# Test to make sure that an argument has been supplied.
if [ -z "$1" ]; then
        usage
        exit 1
fi

# Loop over the file and print each line
# Prepend each line with the filename.
while read LINE; do
        echo "$1: ${LINE}"
done < $1

exit 0

```

3. gareth@brutha: ~/Lec05 (ssh)

```
gareth@brutha:~/Lec05$ ./simple6.sh
This script prints out a given file line by line
It requires a single argument whcih is the file
to read.

simple6.sh <filename>
gareth@brutha:~/Lec05$ ./simple6.sh test_file
test_file: 10 rations of food
test_file: 2 blue potions
test_file: 1 scrol entitiles "exyc ahks"
gareth@brutha:~/Lec05$
```

Debugging

- A Bash script can be just as complicated as a piece of C code.
- Luckily there are some tools we can use to help debug scripts:
  - **set -x** - shows a simple trace of the script executing.
  - **set -n** - stops any commands from running, good for looking at syntax errors.
  - **set -v** - prints out the shell input line as the interpreter runs. Even more verbose that **set -x**

https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html

2. gareth@brutha: ~/Lec05 (ssh)

```
#!/bin/bash

# Debugging support:
export PS4='+$BASH_SOURCE:$LINENO:$FUNCNAME: '
set -x

# A single or multi-line comment telling the person
# reading the code what the script is supposed to do.

function usage {
        echo "This script prints out a given file line by line"
        echo "It requires a single argument whcih is the file"
        echo "to read."
        echo
        echo "simple6.sh <filename>"
}

# Test to make sure that an argument has been supplied.
if [ -z "$1" ]; then
        usage
        exit 1
fi

# Loop over the file and print each line
# Prepend each line with the filename.
while read LINE; do
        echo "$1: ${LINE}"
done < $1

exit 0
```

3. gareth@brutha: ~/Lec05 (ssh)

```
gareth@brutha:~/Lec05$ ./simple6.sh test_file
+./simple6.sh:19:: '[' -z test_file ']'
+./simple6.sh:26:: read LINE
+./simple6.sh:27:: echo 'test_file: 10 rations of food'
test_file: 10 rations of food
+./simple6.sh:26:: read LINE
+./simple6.sh:27:: echo 'test_file: 2 blue potions'
test_file: 2 blue potions
+./simple6.sh:26:: read LINE
+./simple6.sh:27:: echo 'test_file: 1 scrol entitiles "exyc ahks"'
test_file: 1 scrol entitiles "exyc ahks"
+./simple6.sh:26:: read LINE
+./simple6.sh:30:: exit 0
gareth@brutha:~/Lec05$
```

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation

Compiling

- Unlike a shell script, a C program needs to be compiled.
- Compilation means taking the human-readable source code and translating it into a set of instructions for a machine and operating system.
- Unlike a shell script, a compiled program runs independently of other programs.It does not require a shell to interpret it (c.f. python, ruby & perl).
- So far in lectures and labs you've seen something like:
    - **gcc -o loop loop.c**

2. gareth@brutha: ~/Lec05/C (ssh)

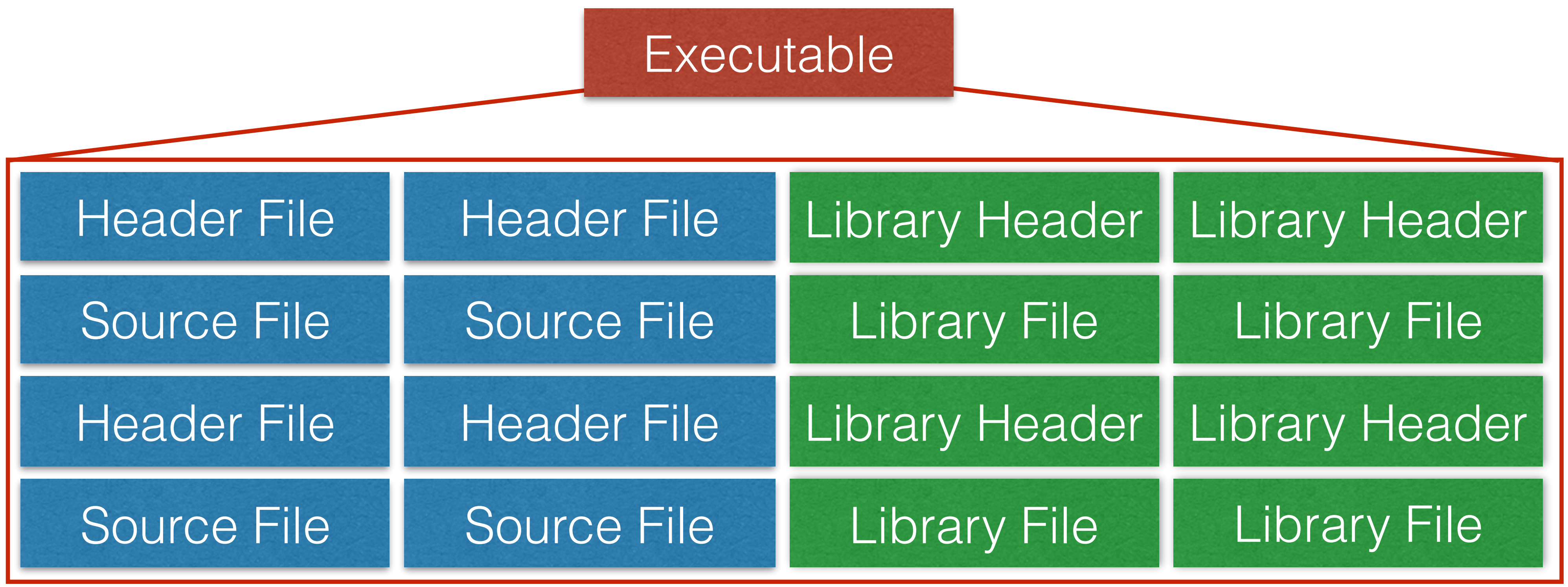
```c
#include <stdio.h>

/*
 * A small program that reads arguments from stdin,
 * loops over the arguments and prints them out one
 * by one.
 *
 */

/* Main */
int main(int argc, char *argv[]) {
    int i,sum,average;

/* Ensure we actually have some arguments */
    if (1 == argc) {
       printf("Arghhhhh\n");
       return 1;
    }

    /* Loop over each Argument and print to screen */
    for (i=1; i < argc; i++) {
        printf("Arg %d - %s\n",i,argv[i]);
    }
    return 0;
}
```

3. gareth@brutha: ~/Lec05/C (ssh)

```
gareth@brutha:~/Lec05/C$ gcc -o loop loop.c
gareth@brutha:~/Lec05/C$ ./loop
Arghhhhh
gareth@brutha:~/Lec05/C$ ./loop a b c d
Arg 1 - a
Arg 2 - b
Arg 3 - c
Arg 4 - d
gareth@brutha:~/Lec05/C$
```

Components of a C program

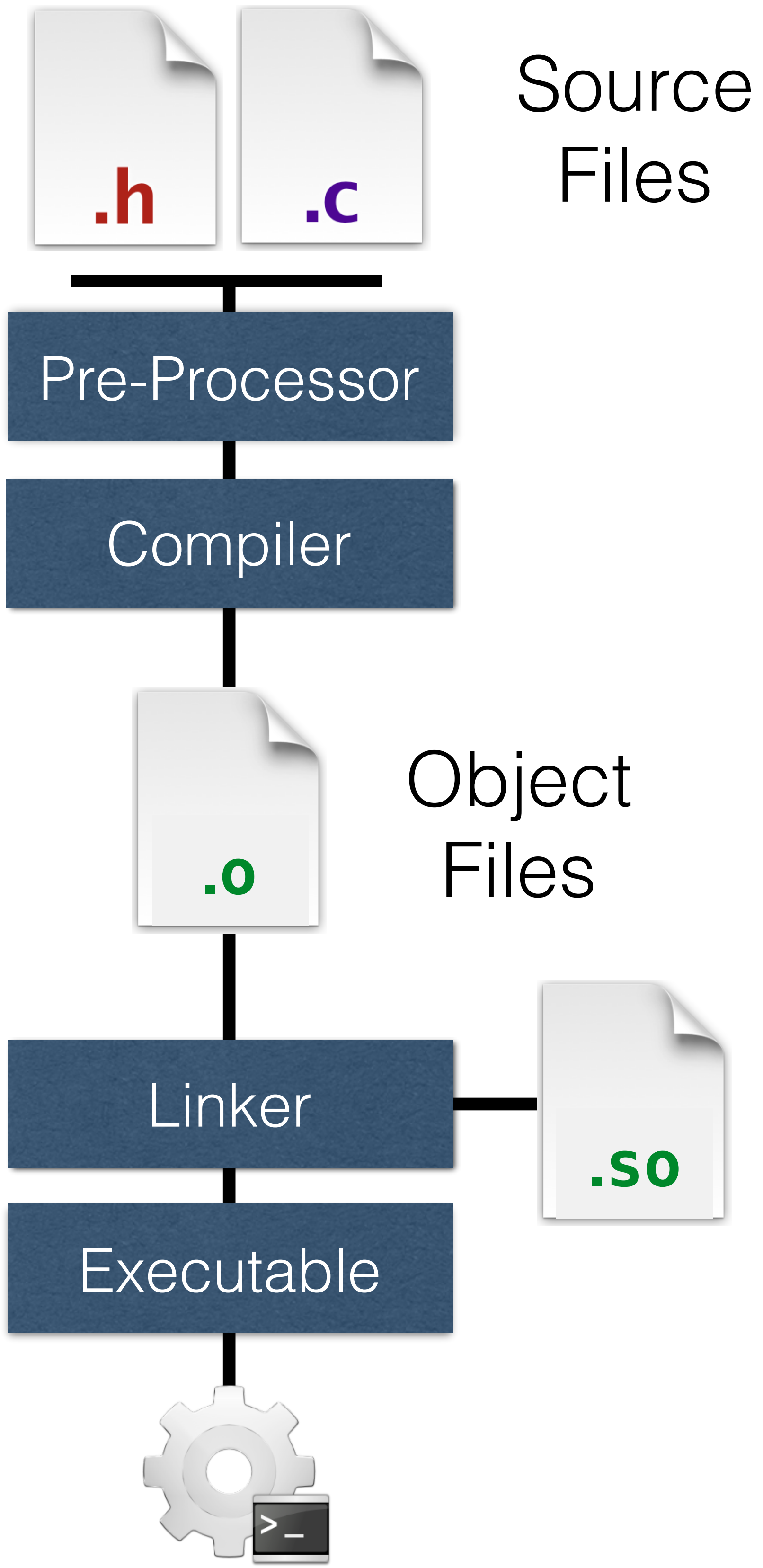
Header File	Definitions & Prototypes (***.h**)
Source File	Our functions, etc (***.c**)
Library Header	Library Definitions & Prototypes (***.h**)
Library File	Pre-compiled Library Functions (***.a**, ***.so**)



An executable is usually created from a number of source and header files (created by us and from libraries).

Compilation

- Compilation is actual three separate stages:
- **Pre-processor** - takes source and header files and combines the two, while expanding all #defines etc (see C lectures).
- **Compilation** - take the preprocessed C files and convert them to machine code, label each function with a name (referred to as a symbol) and create an object file.
- **Linking** - take all the object files, and match the symbols to actual machine code fragments. Place all fragments in the executable and replace the names with addresses.



GNU Compiler Collection

- So far you've been using gcc to compile your code.
- A more complete example of compiling a binary would use the following flags:
  - `-l` :: Library Name
  - `-L` :: Path to search for library code
  - `-I` :: Path to search for header files.
  - `-o` :: Executable name
- In this course you'll likely only by using standard system libraries which means `-L` and `-I` can be ignored as **gcc** will look in all the standard locations (including `.`) by default.
- `-c` can be used to only do pre-processing and compilation, and generates an object file.

```
gcc -l libraries
    -L path
    -I path
    -o name
    source.c
```

```
gcc -l mylib
    -L ./obj/
    -I ./include/
    -o myexe
    source1.c
    source2.c
```

Example

- We've created a small program that creates some random data and writes it to screen.

- Three source files:

    - **main.c** - creates an array of integers, calls a function to generate random data and prints it to the screen.

    - **util.h** - includes **time.h** and **stdlib.h** and has a function prototype.

    - **util.c** - has a function that takes an array and fills it with a series of random numbers.

2. gareth@brutha: ~/Lec05/C/multi (ssh)

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

void generateRandomData(int data[], int len);
```

util.h 1,1 All

```c
#include "util.h"

void generateRandomData(int data[], int len) {

	int i;

	srand(time(NULL));
	for(i=0; i<len; i++) {
		data[i] = rand() % 100;
	}
}
```

util.c 1,1 All

```c
#include <stdio.h>
#include "util.h"

#define SIZE_OF_DATA 10

int main(void) {
	int data[SIZE_OF_DATA];

	generateRandomData(data, SIZE_OF_DATA);

	int i;
	for(i=0; i<SIZE_OF_DATA; i++){
		printf("%d: %d\n",i,data[i]);
	}
	return 0;
}
```

main.c [+] 17,1 All

-- INSERT --

Example

- Compile **util.c** into and object file:
  - **gcc -c util.c**
- Look at the contents of the object file with **nm**:
  - **nm util.o**
  - U means a symbol is unknown, while T means a symbol exist (in the Text/Code Segment)
- Now with **main.c**:
  - **gcc -c main.c**
  - **nm main.o**
- Create the executable **myprog**:
  - **gcc -o myprog main.o util.o**

```
2. gareth@brutha: ~/Lec05/C/multi (ssh)
gareth@brutha:~/Lec05/C/multi$ ls
main.c  util.c  util.h
gareth@brutha:~/Lec05/C/multi$ gcc -c util.c
gareth@brutha:~/Lec05/C/multi$ ls
main.c  util.c  util.h  util.o
gareth@brutha:~/Lec05/C/multi$ nm util.o
0000000000000000 T generateRandomData
                 U rand
                 U srand
                 U time
gareth@brutha:~/Lec05/C/multi$ gcc -c main.c
gareth@brutha:~/Lec05/C/multi$ ls
main.c  main.o  util.c  util.h  util.o
gareth@brutha:~/Lec05/C/multi$ nm main.o
                 U generateRandomData
0000000000000000 T main
                 U printf
gareth@brutha:~/Lec05/C/multi$ gcc -o myprog main.o util.o
gareth@brutha:~/Lec05/C/multi$ ls
main.c  main.o  myprog  util.c  util.h  util.o
gareth@brutha:~/Lec05/C/multi$ ./myprog
0: 8
1: 58
2: 76
3: 99
4: 80
5: 47
6: 82
7: 73
8: 0
9: 76
gareth@brutha:~/Lec05/C/multi$
```

Libraries (Dynamic and Static)

- We need to specify external libraries using the `-l` flag. This will search you **LD_LIBRARY_PATH** env variable or path specified by `-L`
- Linking to libraries can be done in two ways.
- By default **gcc** will link to a library dynamically. This means the library functions are not added in at compile time.
- Instead library functions are loaded when the program starts/runs.
- You can see what library's a program will load dynamically by using **ldd**.
- We can force all libraries functions to be included at compile time by using the **-static** keyword.

```
2. gareth@brutha: ~/Lec05/C/multi (ssh)
gareth@brutha:~/Lec05/C/multi$ clear
gareth@brutha:~/Lec05/C/multi$ ldd myprog
        linux-vdso.so.1 =>  (0x00007fff37800000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fcbbd3c0000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fcbbd7b0000)
gareth@brutha:~/Lec05/C/multi$ gcc -o myprogstatic -static *.o
gareth@brutha:~/Lec05/C/multi$ ls -ltrh
total 900K
-rw-rw-r-- 1 gareth gareth   86 Feb  9 13:29 util.h
-rw-rw-r-- 1 gareth gareth  238 Feb  9 14:07 main.c
-rw-rw-r-- 1 gareth gareth  152 Feb  9 14:07 util.c
-rw-rw-r-- 1 gareth gareth 1.6K Feb  9 14:07 util.o
-rw-rw-r-- 1 gareth gareth 1.6K Feb  9 14:07 main.o
-rwxrwxr-x 1 gareth gareth 8.5K Feb  9 14:49 myprog
-rwxrwxr-x 1 gareth gareth 865K Feb 10 08:42 myprogstatic
gareth@brutha:~/Lec05/C/multi$ ldd myprogstatic
        not a dynamic executable
gareth@brutha:~/Lec05/C/multi$ 
```

- Input (User & File)
- Functions
- Good Practice
- An Aside - Compilation