

CS 240 Summer 2024

Lab 1: Getting Acquainted with C and Its System Environment

Dong Wang
Student ID: 00370-48599

Due: June 19, 2024

Problem 1

Size, Timestamp, and Protection Settings for `main.c`

```
-rw-r----- 1 wang6213 wang6213 185 Jun  6 21:08 main.c
```

The size of `main.c` is 185 bytes. The timestamp `Jun 6 21:08` indicates the last modification time of the file `main.c`. The owner had read (r) and write (w) permissions `rw-`, the group had read (r) permissions `r--`, and others had no permissions `---`.

Modification of Protection Bits for `main.c` and Key Difference

```
chmod 600 main.c  
ls -l main.c
```

After running the above commands, the protection bits for `main.c` were modified as follows:

```
-rw----- 1 wang6213 wang6213 185 Jun  6 21:08 main.c
```

Owner Protection Bits for `a.out`

```
-rwxr-x--- 1 wang6213 wang6213 15776 Jun 13 22:27 a.out
```

The owner had read (r), write (w) and execute (x) permissions `rwx`. The key difference in the protection bits between `main.c` and `a.out` is that `a.out` has execute permissions (x) for the owner and read/execute permissions for the group, whereas `main.c` only had read and write permissions for the owner and read permissions for the group before modification.

Modification to Match main.c

To match the owner protection bits of `a.out` to `main.c`, run the following command:

```
chmod 600 a.out
```

Since the execute permission was removed, the command results in:

```
-bash: ./a.out: Permission denied
```

Problem 2

Two ways of changing the name of the executable file generated by `gcc` are:

```
gcc -o my_program main.c
```

```
gcc main.c  
mv a.out my_program
```

Location of the gcc Executable File

The return of command `whereis gcc` might return something like:

```
gcc: /usr/bin/gcc /usr/lib/gcc /usr/share/gcc /usr/share/man/man1/gcc.1.gz
```

The true location of the `gcc` executable is `/usr/bin/gcc`. The other paths shown by `whereis` include:

- `/usr/lib/gcc`: Contains library files used by `gcc`.
- `/usr/share/gcc`: May contain shared resources and configuration files for `gcc`.
- `/usr/share/man/man1/gcc.1.gz`: The manual page for `gcc`.

The true executable file used when we run the `gcc` command is `/usr/bin/gcc`. This can be confirmed using the `which` command:

```
realpath $(which gcc)
```

This will return:

```
/usr/bin/x86_64-linux-gnu-gcc-11
```

Therefore, the true name of the `gcc` executable is `gcc`, and its full pathname is:

```
/usr/bin/x86_64-linux-gnu-gcc-11
```

Problem 3

Verification of `stdio.h`

Run the command:

```
ls -l /usr/include/stdio.h
```

The return is:

```
-rw-r--r-- 1 root root 31176 May  6 16:34 /usr/include/stdio.h
```

The first `-` means that the `stdio.h` is indeed a file. The size of `stdio.h` is 31176 bytes. The owner had read (r) and write (w) permissions `rw-`, the group had read (r) permissions `r--`, and others had read (r) permissions `r--`.

Verification of execution of `a.out`

The results is:

```
result of 6 plus 3 equals 9
```

Problem 4

Explanation of Changes

It works well too and return the result:

```
result of 6 plus 3 equals 9
```

Since the modified `main33.c` includes the local `prob4.h`, and `prob4.h` is actually a copy of the system's `stdio.h`, the program should be able to compile and run correctly.

Problem 5

Shell: The shell is a command-line interpreter that provides a user interface for accessing the operating system's services.

gcc (GNU Compiler Collection): Before running `a.out`, the source code (written in C or another language) must be compiled into machine code. This is done using a compiler like `gcc`. The `gcc` compiler translates the source code into an object file, which is a binary representation of the code.

Linker: The object file created by `gcc` is not yet a complete executable. The linker takes this object file and combines it with other necessary object files and libraries to produce a final executable file. This process resolves symbol references (e.g., functions and variables) and produces the `a.out` file.

a.out: This is the output file generated by the linker, named according to a historical convention for Unix executables. It is a binary file that contains machine code, ready to be executed by the system's hardware.

Loader: When the shell receives the command `a.out`, it invokes the loader. The loader is a part of the operating system responsible for loading the executable file into

memory. It sets up the execution environment, including memory allocation and addressing, and prepares the program to run.

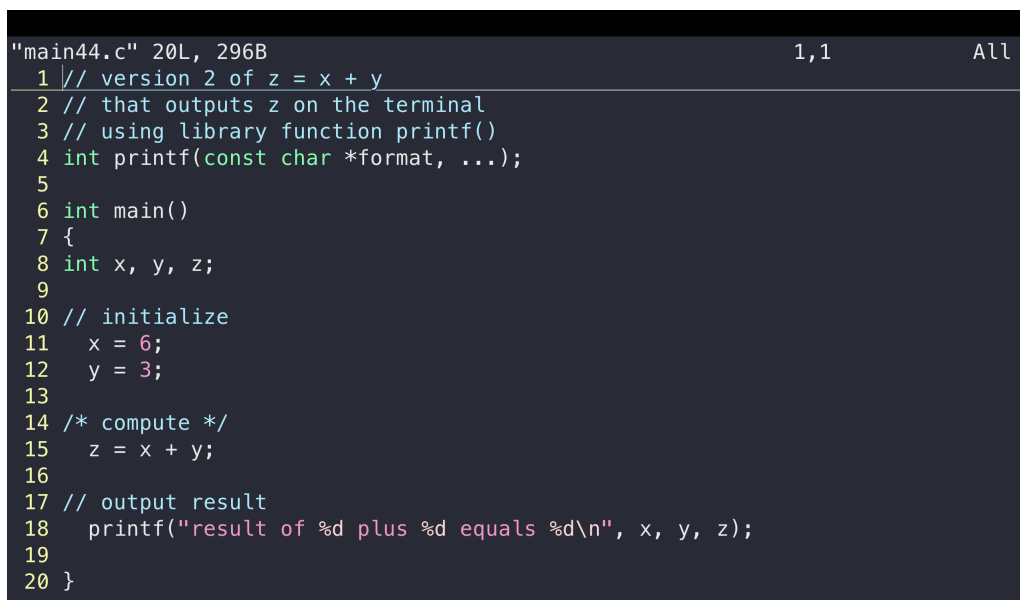
In summary, the shell acts as the interface for executing commands, the loader is responsible for loading the executable into memory and resolving dependencies, the linker helps resolve external references from shared libraries, GCC compiles the C source code into the executable `a.out`, and the CPU executes the machine code instructions contained in `a.out`.

Problem 6

Explanation and Verification

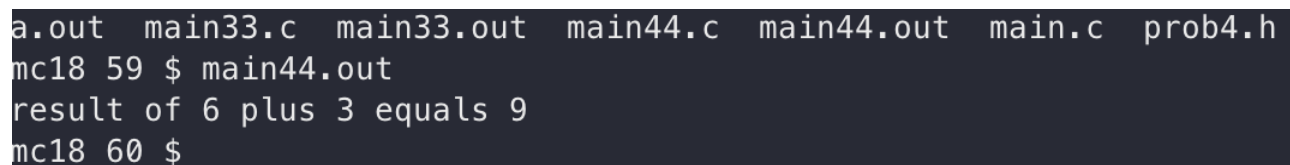
Manually including the function prototype for `printf()` without including the entire `stdio.h` header file can work. The function prototype for `printf()` is:

```
int printf(const char *format, ...)
```



```
"main44.c" 20L, 296B 1,1 All
1 // version 2 of z = x + y
2 // that outputs z on the terminal
3 // using library function printf()
4 int printf(const char *format, ...);
5
6 int main()
7 {
8     int x, y, z;
9
10 // initialize
11     x = 6;
12     y = 3;
13
14 /* compute */
15     z = x + y;
16
17 // output result
18     printf("result of %d plus %d equals %d\n", x, y, z);
19
20 }
```

Figure 1: `main44.c`



```
a.out main33.c main33.out main44.c main44.out main.c prob4.h
mc18 59 $ gcc main44.c
result of 6 plus 3 equals 9
mc18 60 $ ./main44.out
```

Figure 2: compiles and executes correctly

Problem 7

The `main()` function typically returns 0 to indicate successful execution. Other return values can indicate different types of abnormal terminations, depending on the value returned.

Return Value Experiments

- **Return Value: 0**
 - Outcome: The program runs successfully, exit status is 0
- **Return Value: 1**
 - Outcome: The program runs, exit status is 1
- **Return Value: -1**
 - Outcome: The program runs, exit status is 255
- **Return Value: 127**
 - Outcome: The program runs, exit status is 127
- **Return Value: 255**
 - Outcome: The program runs, exit status is 255
- **Return Value: -255**
 - Outcome: The program runs, exit status is 1
- **Return Value: 2**
 - Outcome: The program runs, exit status is 2
- **Return Value: -2**
 - Outcome: The program runs, exit status is 254

Problem 8

When using `scanf()` in C to read input, it's crucial to pass the addresses of the variables where the input values will be stored. This is achieved using the ampersand `&` operator in front of the variable names. If we write `scanf("%d %d", x, y)` instead of `scanf("%d %d", &x, &y)`, gcc will respond with warnings rather than errors during compilation.

This warning indicates that `scanf()` expects pointers (addresses of variables) but receives integers instead.

```

mc18 65 $ gcc main.c
main.c: In function 'main':
main.c:14:11: warning: format '%d' expects argument of type 'int *', but argumen
t 2 has type 'int' [-Wformat=]
    14 |     scanf("%d %d", x, y);
        |           ^~      ~
        |           |       |
        |       int *   int
main.c:14:14: warning: format '%d' expects argument of type 'int *', but argumen
t 3 has type 'int' [-Wformat=]
    14 |     scanf("%d %d", x, y);
        |           ^~      ~
        |           |       |
        |       int *   int
mc18 66 $

```

Figure 3: problem 8

Compilation Warnings vs. Errors

- **Compilation Warning:** This indicates a potential issue in the code that may lead to runtime problems. However, `gcc` will still generate an executable file (`a.out`). Warnings do not halt the compilation process.
- **Compilation Error:** This indicates a serious issue that violates the syntax or semantic rules of the language, preventing `gcc` from generating an executable. The compilation process stops, and no `a.out` file is produced.

Omitting the ampersands may result in "Segmentation fault" when we try to compile `main.c`. This is because `scanf()` tries to write the input values to the memory addresses specified by its arguments. Without the ampersand, `scanf()` interprets the integer values of `x` (6) and `y` (3) as memory addresses instead of pointers, leading to undefined behavior and a segmentation fault.

Problem 9

Passing Arguments to `add2()`

When passing the two arguments to `add2()`, we do not use ampersands (i.e., we invoke `add2(x, y)` instead of `add2(&x, &y)`). The ampersand operator is used to pass the address of a variable, which is not necessary in this context since `add2()` expects the values themselves, not their addresses.

If we mistakenly use ampersands, like `add2(&x, &y)`, the code will result in a compilation error. This is because the function `add2()` is defined to take arguments of type `float`, not `float*`.

Testing the Code

To test the effect of this change, make a copy of `v5/main.c` as `v5/main55.c` and compile it using `gcc`.

```

mc18 55 $ cp ./main.c ./main55.c
mc18 56 $ vim main55.c
mc18 57 $ gcc -o main55.out main55.c
main55.c: In function 'main':
main55.c:17:12: error: incompatible type for argument 1 of 'add2'
   17 |     z = add2(&x, &y);
      |            ^~
      |            |
      |            float *
main55.c:7:12: note: expected 'float' but argument is of type 'float *'
    7 | float add2(float, float);
      |            ^~
main55.c:17:16: error: incompatible type for argument 2 of 'add2'
   17 |     z = add2(&x, &y);
      |                ^~
      |                |
      |                float *
main55.c:7:19: note: expected 'float' but argument is of type 'float *'
    7 | float add2(float, float);
      |               ^~

```

Figure 4: main44.c

This indicates a compilation bug due to the incorrect passing of arguments. Therefore, using ampersands in this context results in a compilation error rather than a run-time bug.

Problem 10

What was the purpose of the coding change from v5/ to v6/?

The purpose of the coding change from v5/ to v6/ was to enhance the modularity of the code by separating the addition operation into a distinct function, `add2()`. This makes the design more organized and easier to manage, especially for more complex tasks that benefit from being divided into smaller, reusable functions.

Do we need to name the new file `add2.c`? If not, why name the file `add2.c`?

We do not strictly need to name the new file `add2.c`. The file is named `add2.c` for clarity and organization, indicating that it contains the implementation of the `add2()` function. This naming convention helps maintain a well-organized codebase, making it easier to identify and locate specific functionalities within the project. And also in the file `main.c`, the calling function is `add2()`, which is a corresponding relationship.

Assuming all files containing C code end with suffix `.c` are contained in the current directory, what shortcut courtesy of our shell can be used to reduce typing effort when invoking `gcc`?

To reduce typing effort when invoking `gcc`, we can use a wildcard character. The shell shortcut for compiling all C files in the current directory with `gcc` is:

```
gcc -o output_name *.c
```

Here, `*.c` matches all files with the `.c` suffix in the current directory, and `-o output_name` specifies the name of the output executable file. This shortcut simplifies the compilation

process by allowing us to compile all C source files at once without having to list each file individually.

Problem 11

Code Implementation

The `maximus3()` function is implemented as follows:

```
1 #include <stdio.h>
2
3 int maximus3(int a, int b, int c)
4 {
5     int max = a;
6     if (b > max) {
7         max = b;
8     }
9     if (c > max) {
10        max = c;
11    }
12    // return the max integer
13    return max;
14 }
```

Figure 5: `maximus3.c`

The `main.c` file includes a test function to verify the correctness of `maximus3()`:

```
mc18 131 $ vim main.c
mc18 132 $ gcc -o main.out maximus3.c main.c
mc18 133 $ main.out
maximus3(1, 2, 3) = 3
maximus3(3, 2, 1) = 3
maximus3(-1, -2, -3) = -1
maximus3(0, 0, 0) = 0
maximus3(5, 5, 5) = 5
maximus3(1, 10, 5) = 10
Type in 3 integers: 3 8 9
The maximus is 9
```

Figure 6: `main.c`

Exhaustive Testing

To exhaustively determine the correctness of `maximus3()` by brute force, we need to consider all possible combinations of three 32-bit integers. The range of a 32-bit signed integer is from $-2,147,483,648$ to $2,147,483,647$, giving us 2^{32} distinct values.

Since we need to test all combinations of three such integers, the total number of tests required would be:

$$(2^{32})^3 = 2^{96}$$

This equals approximately 7.92×10^{28} tests. This number is impractically large for brute force testing.

Bonus Problem

Code for `raisetopower` Function

```
1 #include <stdio.h>
2
3 int raisetopower(int base, int exponent){
4     //Edge case
5     if (base == 0) return 0;
6     if (exponent == 0) return 1;
7
8     int result = 1;
9     for (int i = 0; i < exponent; i++) {
10         result *= base;
11     }
12     return result;
13 }
```

Figure 7: `raisetopower.c`

Code for main

```
"main.c" 17L, 516B                                1,1
1 #include <stdio.h>
2
3 int raisetopower(int a, int b);
4
5 int main() {
6     // Test cases
7     printf("2^3 = %d\n", raisetopower(2, 3)); // Expected output: 8
8     printf("5^0 = %d\n", raisetopower(5, 0)); // Expected output: 1
9     printf("0^5 = %d\n", raisetopower(0, 5)); // Expected output: 0
10    printf("3^4 = %d\n", raisetopower(3, 4)); // Expected output: 81
11    printf("Try to type two integers: ");
12
13    int x, y;
14    scanf("%d %d", &x, &y);
15    printf("%d^%d = %d\n", x, y, raisetopower(x, y));
16    return 0;
17 }
```

Figure 8: main.c

Compiling and Testing the Code

```
mc18 115 $ vim main.c
mc18 116 $ gcc -o main.out raisetopower.c main.c
mc18 117 $ main.out
2^3 = 8
5^0 = 1
0^5 = 0
3^4 = 81
Try to type two integers: 3 3
3^3 = 27
```

Figure 9: bonusresult.c

Code Explanation

- **Special Cases Handling:**

- If **base** is 0, the result is 0 regardless of the exponent.
- If **exponent** is 0, the result is 1 regardless of the base (except when base is 0, which is handled before).

- **Loop for Power Calculation:**

- The **for** loop multiplies the **result** by **base** for **exponent** times to compute the power.

- **Test Cases:**

- Several test cases are included in the **main** function to verify that the function works correctly for different scenarios.