

Lab # 4

C Programming Introduction

CS311L

Lab Engr. Eisha ter raazia Mir

Learning Outcomes

- Compile the C program in gcc compiler
- The compilation process
- Malloc() and free()

Install the C Compiler:

Ubuntu Linux typically comes with the GNU Compiler Collection (GCC) installed by default. However, if it's not installed on your system, you can install it via the terminal using the following command:

- **sudo apt-get update**
- **sudo apt-get install build-essential**

Introduction to C

- C compiler in linux, ***gcc***
- Extension ***.c***
- Format to compile a C program by ***gcc*** compiler is given by:

gcc first.c -o first

- The file is executed as:

./first

C programming

```
#include <stdio.h>  
int main() {  
    printf("Hello, World!\n");  
    return 0;  
}
```

How Does the Program works

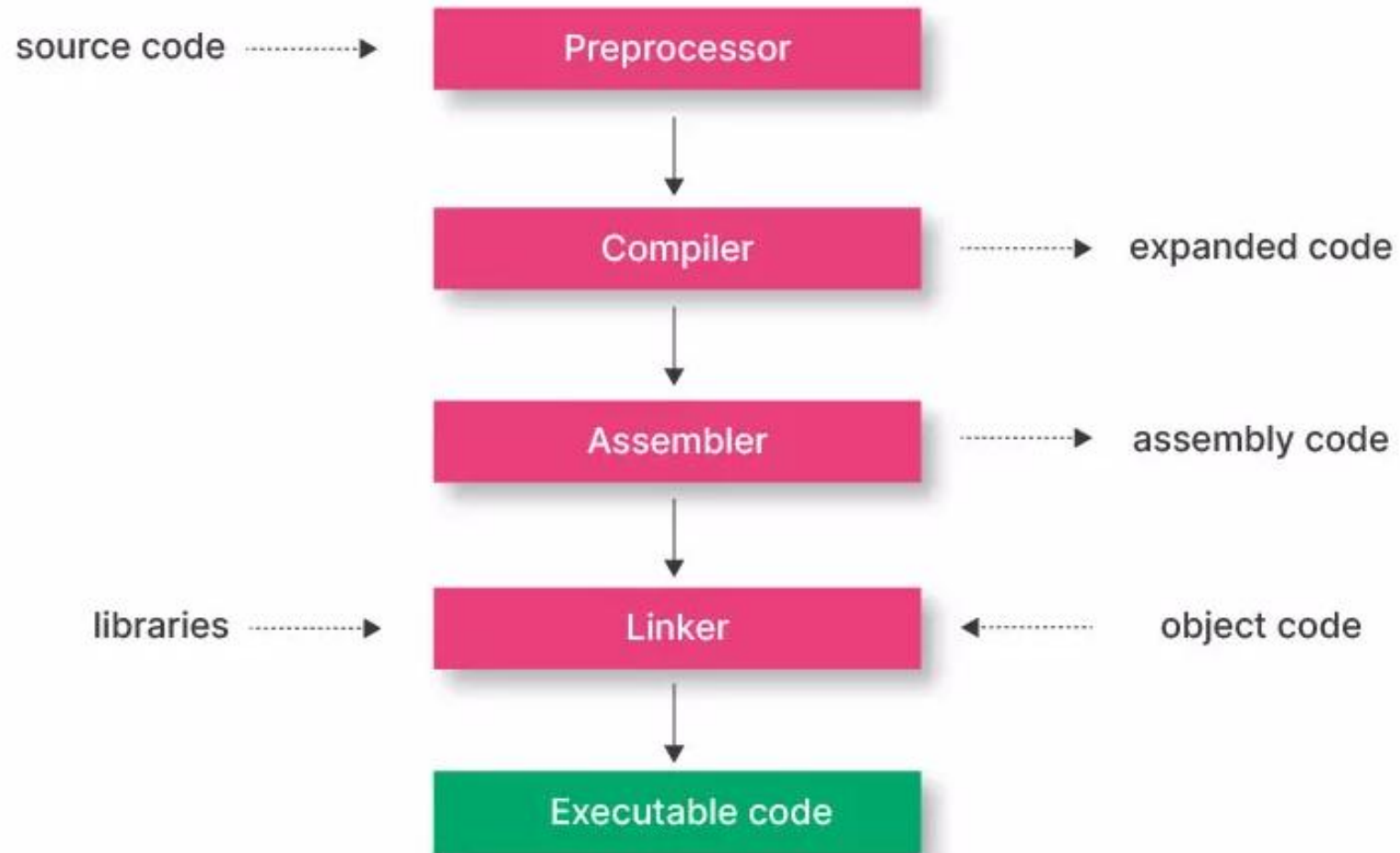
- Must contain the **main()** function
- The code execution begins with **main()** function
- The **printf()** is to send formatted output to the screen
- To use **printf()** include **stdio.h** header file using the **#include <stdio.h>** statement
- The **return 0** "Exit status"

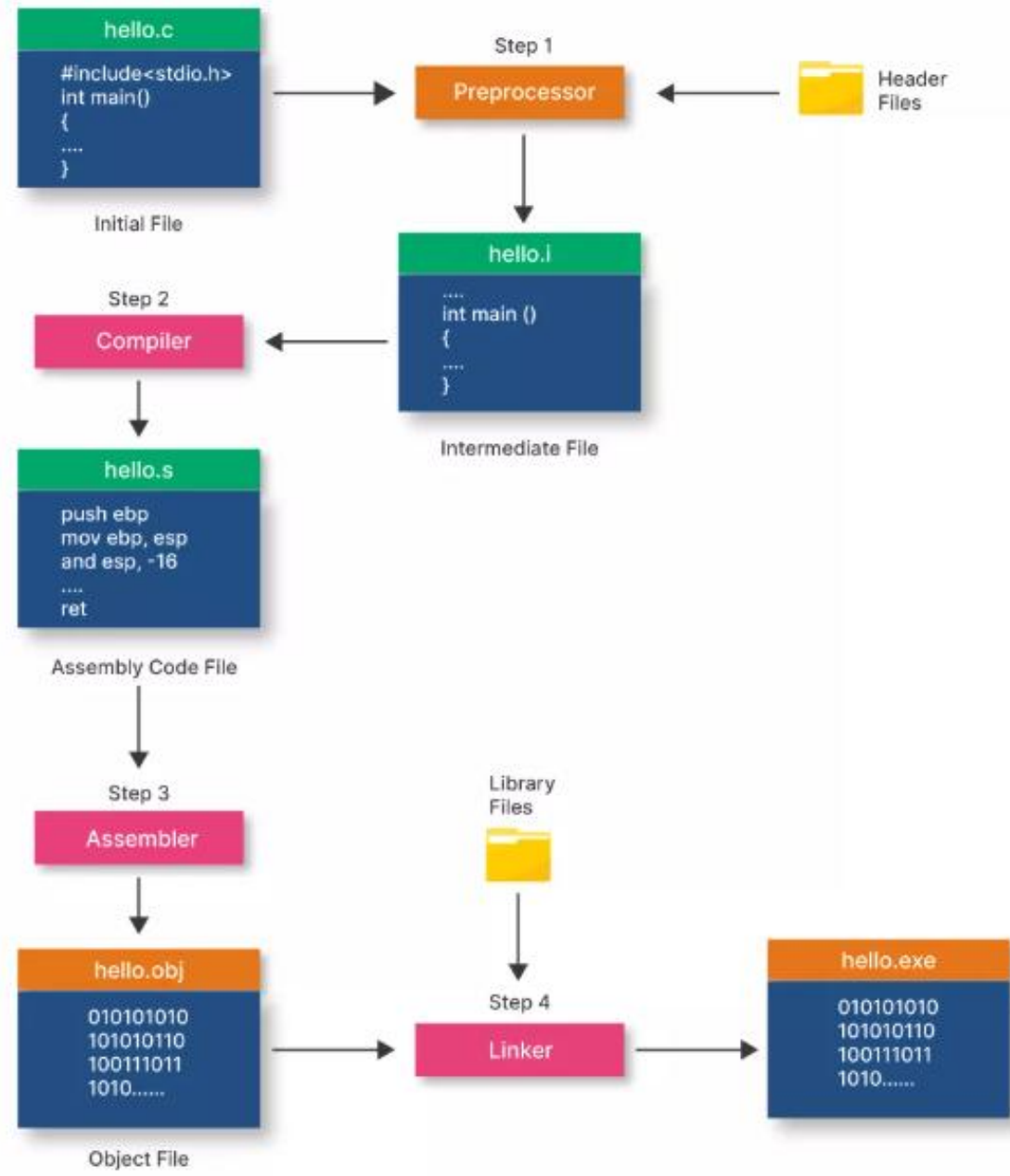
Compilation process:

- Compilation refers to the process by which human-readable source code written in any programming language is transformed into machine-executable binary code.
- In essence, compilation in C language is a means by which we take our high-level code, written in the C programming language, and translate it into low-level machine-readable code that a computer's central processing unit (CPU) can directly execute.

Step by Step Compilation using gcc

1. **Preprocessing:** Handles directives like `#include` and `#define`, expanding macros, and including header files to prepare the source code for compilation.
2. **Compilation:** Translates the preprocessed code into assembly language, checking for syntax errors and optimizing the code.
3. **Assembly:** Converts assembly code into machine code, producing object files with a `.obj` or `.o` extension.
4. **Linking:** Combines object files and libraries into a single executable, resolving references to functions and variables.





Step by Step Compilation

- The pre-processor:
 - **gcc -E 1.c -o 1.i**
- The Compilation:
 - **gcc -S 1.i -o 1.s**
- The Assembler:
 - **gcc -c 1.s -o 1.o**
- Linker:
 - **gcc 1.o -o output**

Input and Output

➤ Input:

- ***scanf()***

➤ Output:

- ***printf()***

Format Specifiers

1. In C, when we want to print or read data using `printf()` or `scanf()`, the computer needs to know what kind of data we are dealing with an integer, a character, a floating-point number, etc.
2. In C, `printf` and `scanf` don't know the types of your variables. They only see a list of values stored at different memory addresses.
3. To interpret those values correctly, they rely on the format specifiers you give (`%d`, `%f`, `%c`, etc.).
4. Without them, the function wouldn't know how many bytes to read or how to interpret the data.

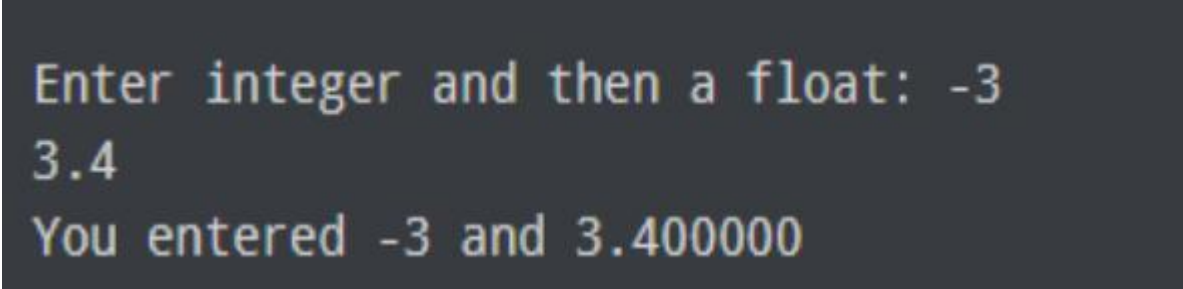
Data Type	Format Specifier
<code>int</code>	<code>%d</code>
<code>char</code>	<code>%C</code>
<code>float</code>	<code>%f</code>
<code>double</code>	<code>%lf</code>
<code>short int</code>	<code>%hd</code>
<code>unsigned int</code>	<code>%u</code>
<code>long int</code>	<code>%li</code>
<code>long long int</code>	<code>%lli</code>
<code>unsigned long int</code>	<code>%lu</code>
<code>unsigned long long int</code>	<code>%llu</code>
<code>signed char</code>	<code>%r</code>
<code>unsigned char</code>	<code>%C</code>
<code>long double</code>	<code>%Lf</code>

```
#include <stdio.h>

int main() {
    int a;
    float b;
    printf("Enter an integer and then a float: ");
    // Taking multiple inputs
    scanf("%d %f", &a, &b);
    printf("You entered %d and %f", a, b);
    return 0;
}
```

Note:

- scanf needs & (address) so it knows where in memory to store the input value.
- printf doesn't need & because it only reads and displays the value, not change it.

A screenshot of a terminal window with a dark background. It shows the execution of a C program. The first line is the prompt "Enter integer and then a float:" followed by the user input "-3" on the same line and "3.4" on the next line. The second line shows the program's output: "You entered -3 and 3.400000".

```
Enter integer and then a float: -3
3.4
You entered -3 and 3.400000
```

malloc() and free() functions

- **malloc()** used to dynamically allocate memory at runtime (when you don't know in advance how much memory you'll need).
- Example: creating arrays whose size depends on user input.
- **free()** used to release that allocated memory back to the system when you're done, so it doesn't cause memory leaks.

Syntax of malloc() and free()

Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

Syntax of free()

```
free(ptr);
```



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

int main() {
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) malloc(n * sizeof(int));

    // if memory cannot be allocated
    if(ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);

    // deallocating the memory
    free(ptr);

    return 0;
}
```

Output

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```



Tasks

1. Write a C code which **asks** from **user** to enter his/her **name** and **age** then it prints ***Hello, Name(Eisha), you are age(22) years old***
2. Write a code in C which **asks** from **user** to enter **two numbers** and then perform ***all the basic operations*** on them (+, -, *)
3. Create a C program named task3 that prompts the user to enter their GPA and the number of courses they are enrolled in, displaying these values with two decimal places using printf. Then, prompt the user to enter their name, allocate dynamic memory to store the entered name, and display it using printf.