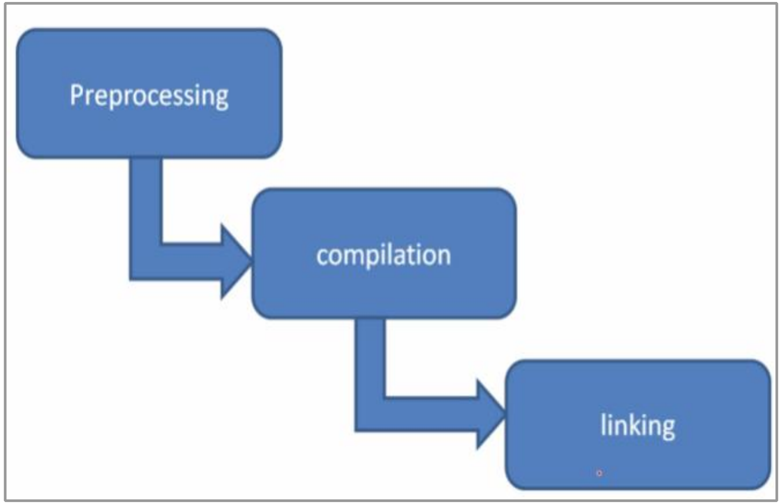
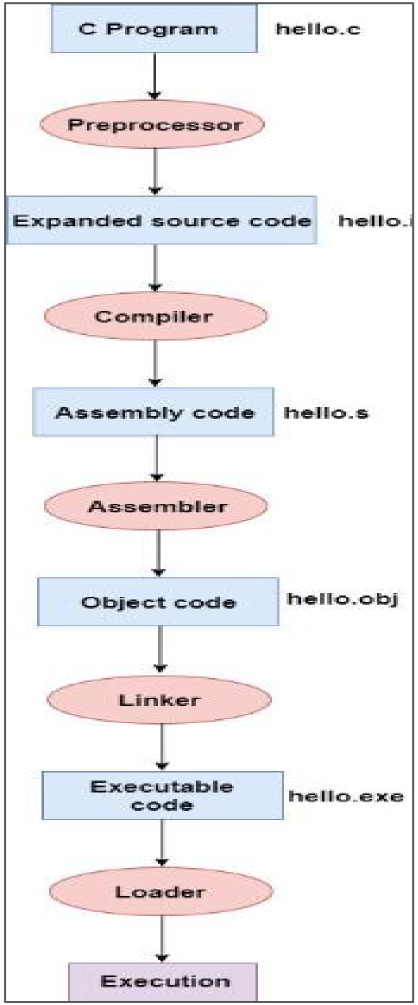
C Build Process

by

C.Mahesh

Overview of Build process:





GNU toolchain

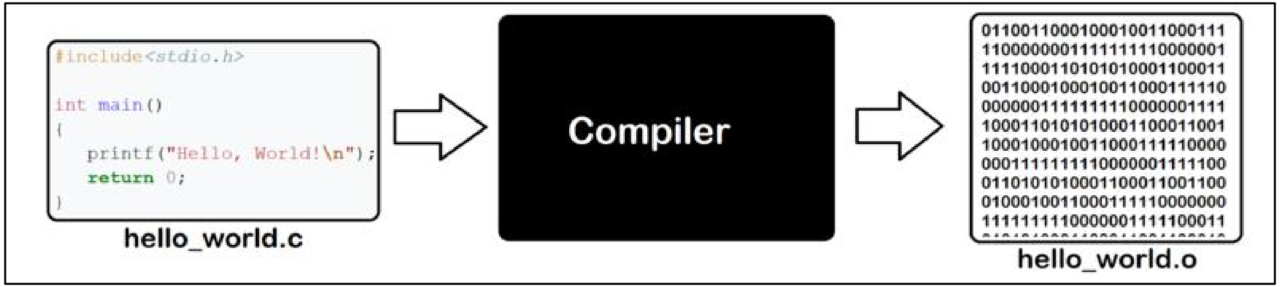
The **GNU toolchain** is a broad collection of programming tools is a broad collection of programming tools produced by the GNU Project.

Major components of GNU toolchain are:

- GNU make: an automation tool for compilation and build
- GNU Compiler Collection (GCC): a suite of compilers for several programming languages
- GNU C Library (glibc): core C library including headers, libraries, and dynamic loader
- GNU Binutils: a suite of tools including linker, assembler and other tools
- GNU m4: an m4 macro processor
- GNU Debugger (GDB): a code debugging tool

GCC(GNU Compiler Collection):

What is compiler ?



```
#include<stdio.h>

int main()
{
   printf("Hello, World!\n");
   return 0;
}
```

```
0110011000100010011000111
1100000001111111110000001
1111000110101010001100011
0011000100010011000111110
0000001111111110000001111
1000110101010001100011001
1000100010011000111110000
0001111111110000001111100
0110101010001100011001100
0100010011000111110000000
1111111110000001111100011
```

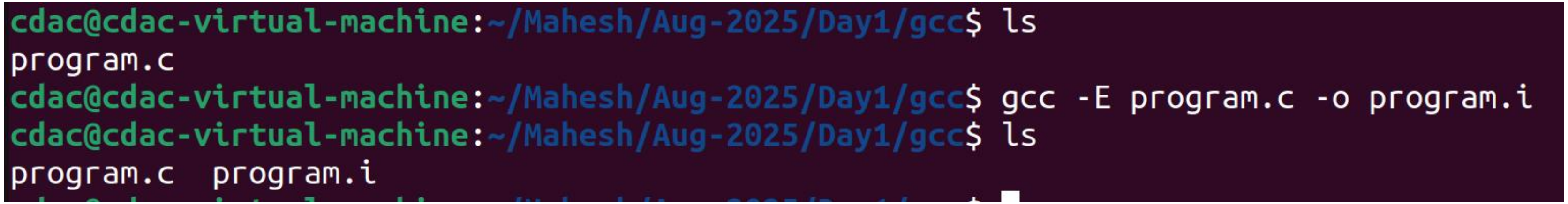
A compiler is system software which converts programming language code (human-readable source code)into binary format(machine-readable code or an intermediate code)in single steps.

GCC Compiler:

- GCC stands for GNU Compiler Collection. It is an integrated distribution of compilers, written in C language and developed by GNU Project, that supports a lot of programming languages: C, C++, Objective-C, Fortran, Go and more.
- The process of transforming source code into an executable involves the following stages: preprocessing, compilation, assembly, and linking.
- In Linux, there is a command called **gcc** that generates an **executable file** from a **.c file**

GCC Compilation steps

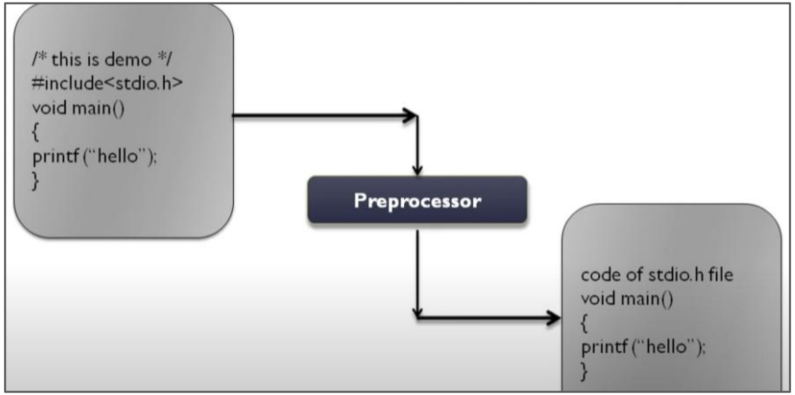
1.Preprocessing(-E):gcc -E hello.c -o hello.i



```
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ ls
program.c
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ gcc -E program.c -o program.i
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ ls
program.c  program.i
```

- Comments removed (// This is a comment is gone).
- Macros expanded (PI replaced with 3.14).
- Headers expanded
- (#include <stdio.h> → thousands of lines of library code).

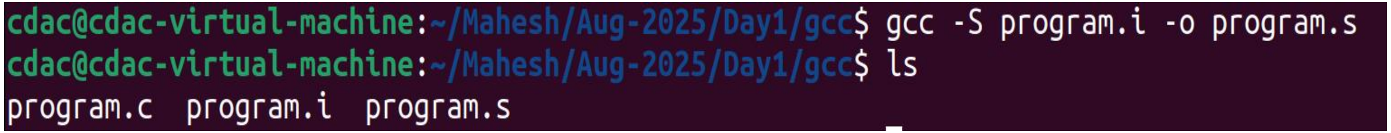
File generated. hello.i(expanded source code)



```
/* this is demo */
#include<stdio.h>
void main()
{
printf ("hello");
}
```

```
code of stdio.h file
void main()
{
printf ("hello");
}
```

2.Compilation(-S):gcc -S hello.i -o hello.s



```
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ gcc -S program.i -o program.s
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ ls
program.c  program.i  program.s
```

The compiler takes the output file of the preprocessor and generates an assembly code. In Linux, the command line that allows to generate that file is **gcc with the -S** option:

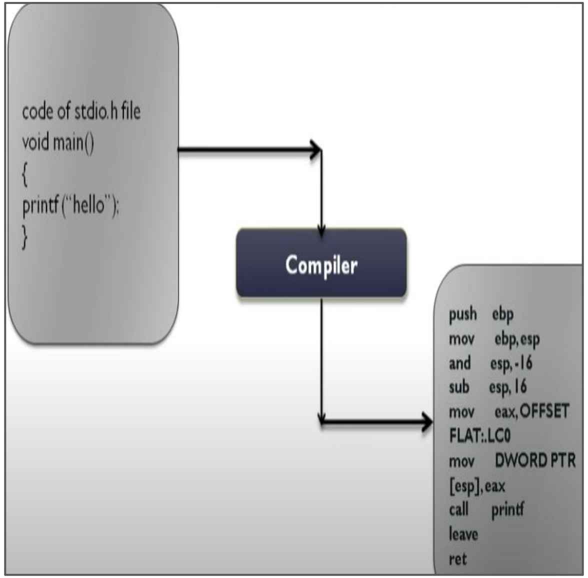
Description:

The -S option instructs GCC to stop after compilation (generating assembly code).

What happens:

Translates C code into **assembly code** (human-readable, CPU instructions).

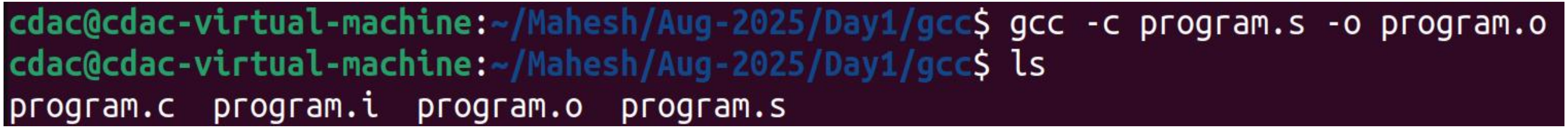
Analogy: Now recipe is written in **local language of the chef (CPU)**.



```
code of stdio.h file
void main()
{
printf("hello");
}
```

```
push   ebp
mov    ebp,esp
and    esp,-16
sub    esp,16
mov    eax,OFFSET
FLAT:.LC0
mov    DWORD PTR
[esp],eax
call   printf
leave
ret
```

3. Assemble(-c):gcc -c hello.s -o hello.o

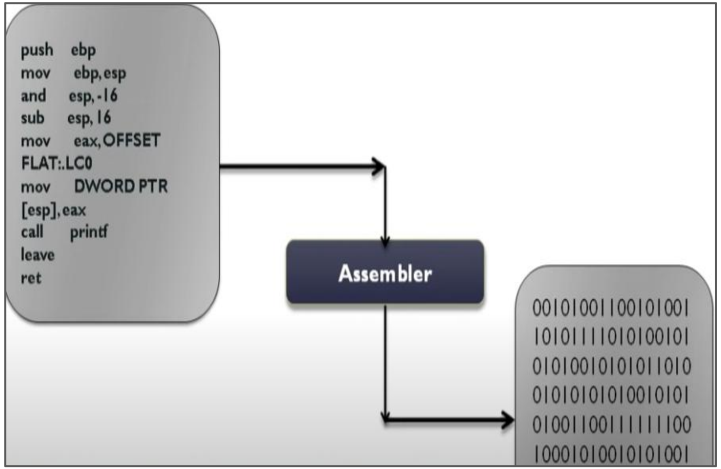


```
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ gcc -c program.s -o program.o
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ ls
program.c  program.i  program.o  program.s
```

This component of GCC takes the assembly code & converts it into a machine code(object code). It means, this is an object file with low-level language.

When compiling a C program, an **object file** (.o) is an **intermediate file** that contains compiled machine code but is not yet executable. Object files are later **linked** to create a final executable.

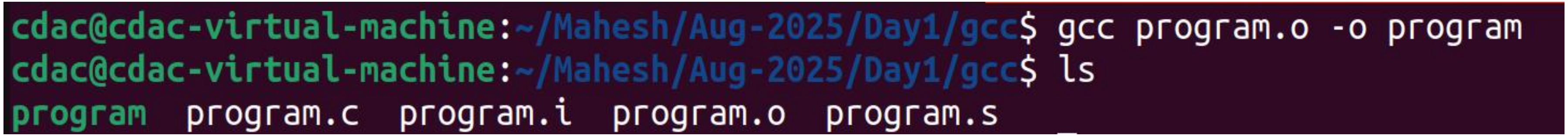
Analogy: Ingredients are **chopped & pre-cooked**, but meal is not served yet.



```
push  ebp
mov   ebp,esp
and   esp,-16
sub   esp,16
mov   eax,OFFSET
FLAT:.LC0
mov   DWORD PTR
[esp],eax
call  printf
leave
ret
```

```
00101001100101001
10101111010100101
01010010101011010
01010101010010101
01001100111111100
10001010010101001
```

4.Linking:gcc hello.o -o hello



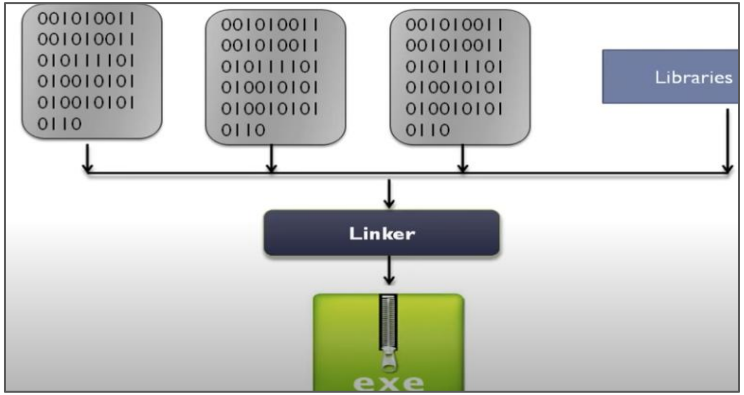
```
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ gcc program.o -o program
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ ls
program  program.c  program.i  program.o  program.s
```

The linker merges the codes into a single one. Also, this component of GCC merges library function codes declared with the codes developed.

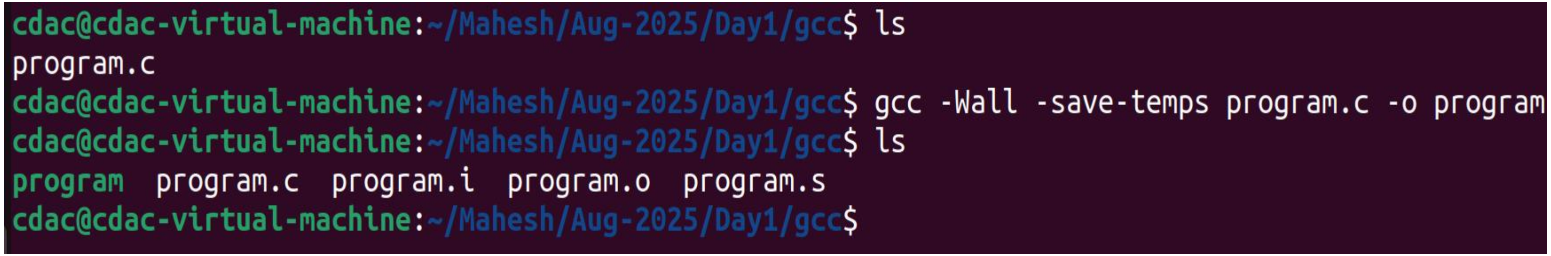
In this step, the executable is already generated (main).

Now you **serve the complete dish** by combining your cooking with ready-made sauces (libraries).

Run the program: **./hello**



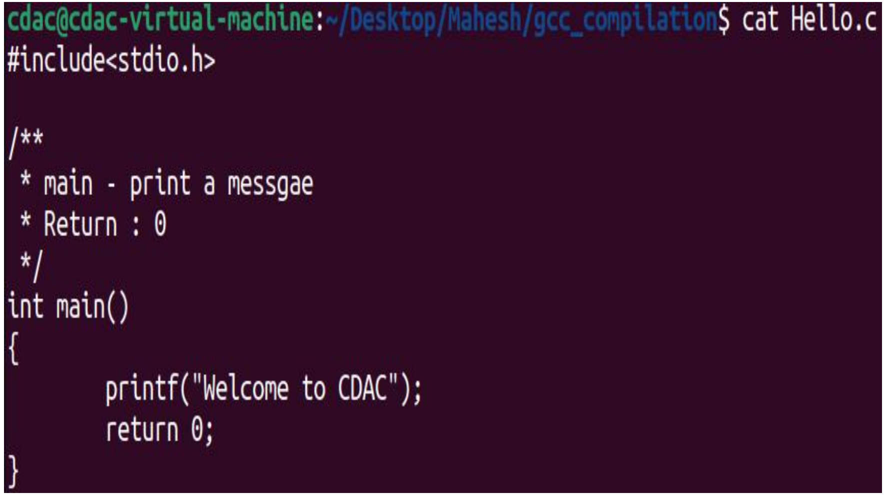
All:



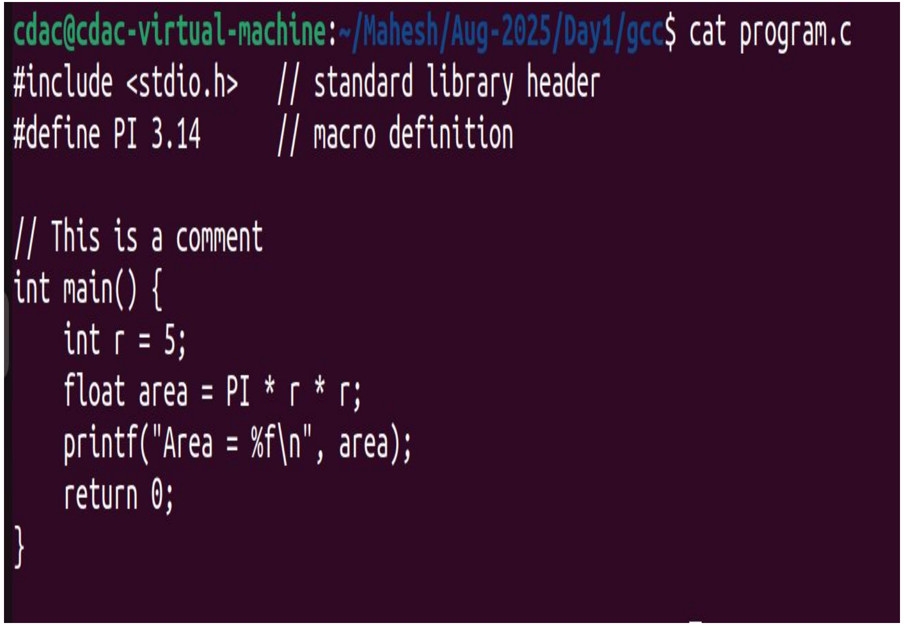
```
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ ls
program.c
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ gcc -Wall -save-temps program.c -o program
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ ls
program  program.c  program.i  program.o  program.s
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$
```

- -Wall: This option stands for "enable all warnings." It instructs the compiler to generate warning messages for a wide range of potential issues in the source code. Using -Wall is a good practice to catch common programming errors.
- -save-temps: This option tells the compiler to keep intermediate files generated during the compilation process.

Run:



```
cdac@cdac-virtual-machine:~/Desktop/Mahesh/gcc_compilation$ cat Hello.c
#include<stdio.h>

/**
 * main - print a messgae
 * Return : 0
 */
int main()
{
        printf("Welcome to CDAC");
        return 0;
}
```



```
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ cat program.c
#include <stdio.h>   // standard library header
#define PI 3.14      // macro definition

// This is a comment
int main() {
    int r = 5;
    float area = PI * r * r;
    printf("Area = %f\n", area);
    return 0;
}
```

Compile with GCC:

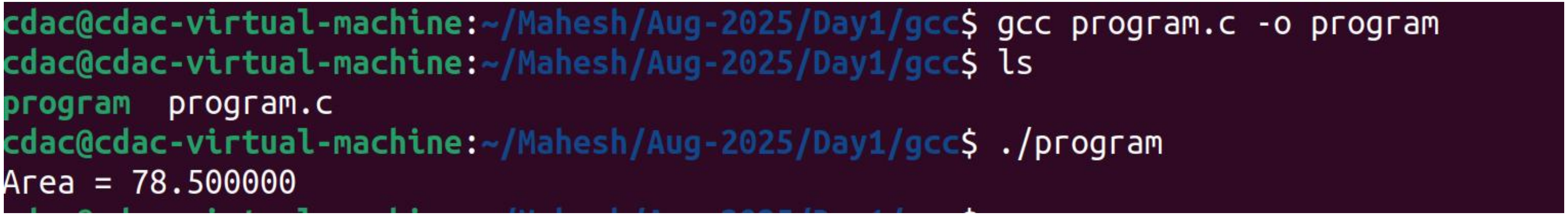
- **gcc hello.c -o hello**
- This command tells GCC to compile the hello.c source code and create an executable named hello (-o option specifies the output file).

-o: output file name.

Run the Executable:

- After the compilation is successful, run the executable:

```
./hello
```



```
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ gcc program.c -o program
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ ls
program  program.c
cdac@cdac-virtual-machine:~/Mahesh/Aug-2025/Day1/gcc$ ./program
Area = 78.500000
```

- It **automatically runs all 4 stages**:
    1. Preprocessing
    2. Compilation
    3. Assembling
    4. Linking

That means you don’t need to run -E, -S, -c separately unless you want to *see intermediate files*.

-o hello

-o = **output file name**.

By default, GCC creates an executable named a.out.

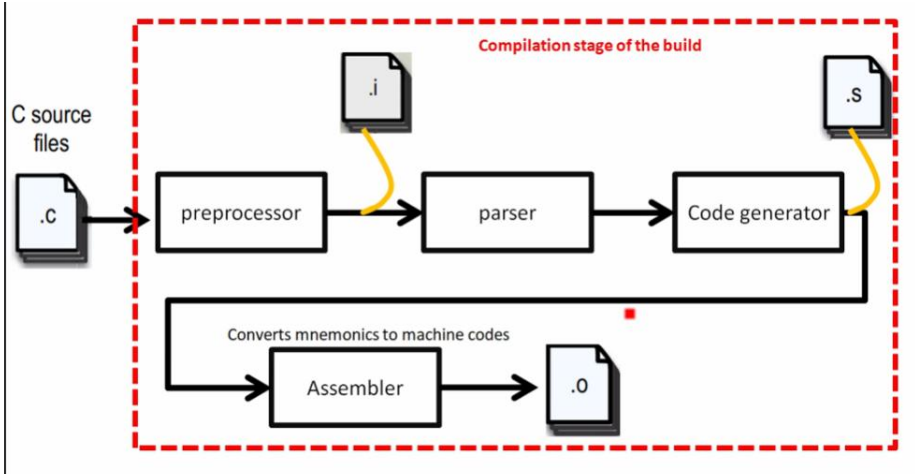
With -o hello, we **rename the final executable** to hello.

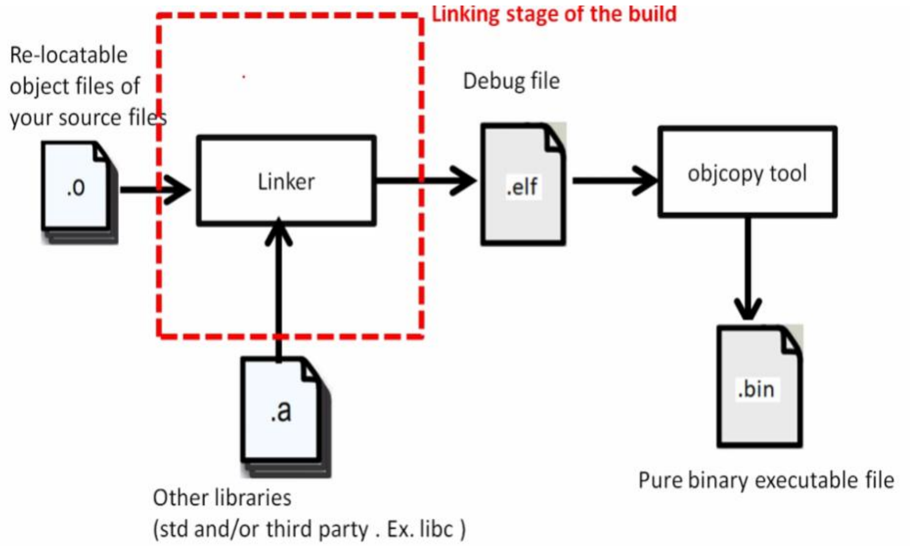
Note:“Compile my source program program.c through all stages and give me a final executable named hello.”

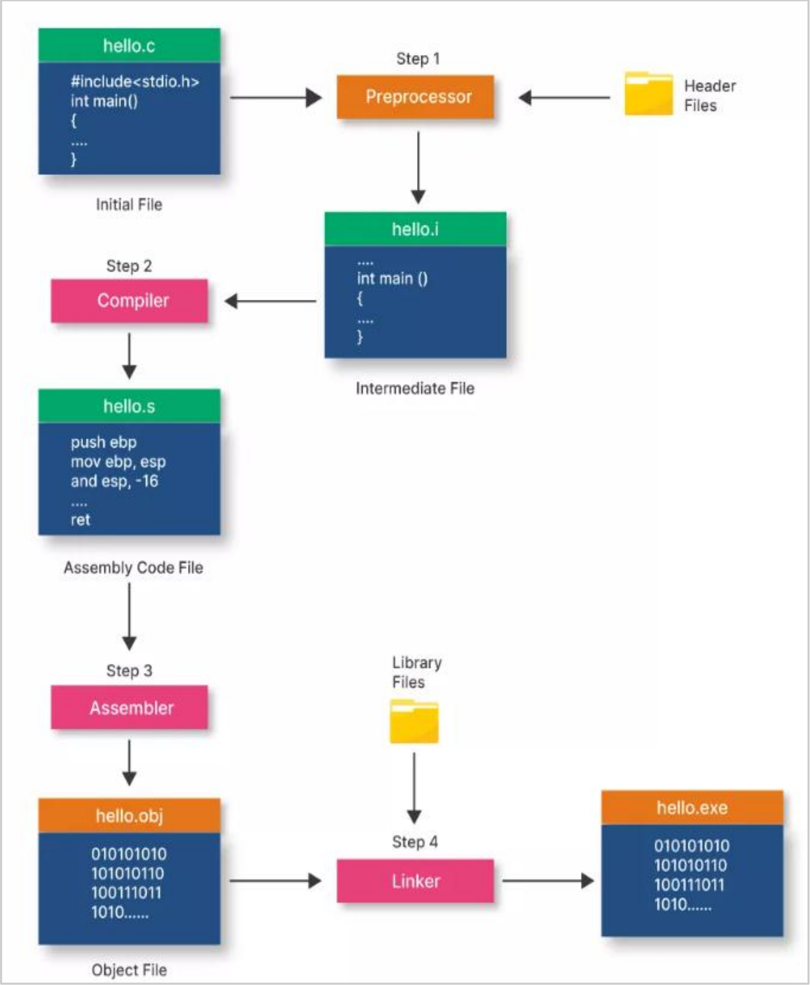
Run the program: **./hello**

./ tells the shell:

“Run this executable **from right here**, in this folder.”


Compilation stage of the build
C source files
.c
preprocessor
.i
parser
Code generator
.s
Converts mnemonics to machine codes
Assembler
.o


Linking stage of the build
Re-locatable object files of your source files
.o
Linker
Debug file
.elf
objcopy tool
.a
Other libraries
(std and/or third party . Ex. libc)
.bin
Pure binary executable file



hello.c

```
#include<stdio.h>
int main()
{
....
}
```

hello.i

```
....
int main ()
{
....
}
```

hello.s

```
push ebp
mov ebp, esp
and esp, -16
....
ret
```

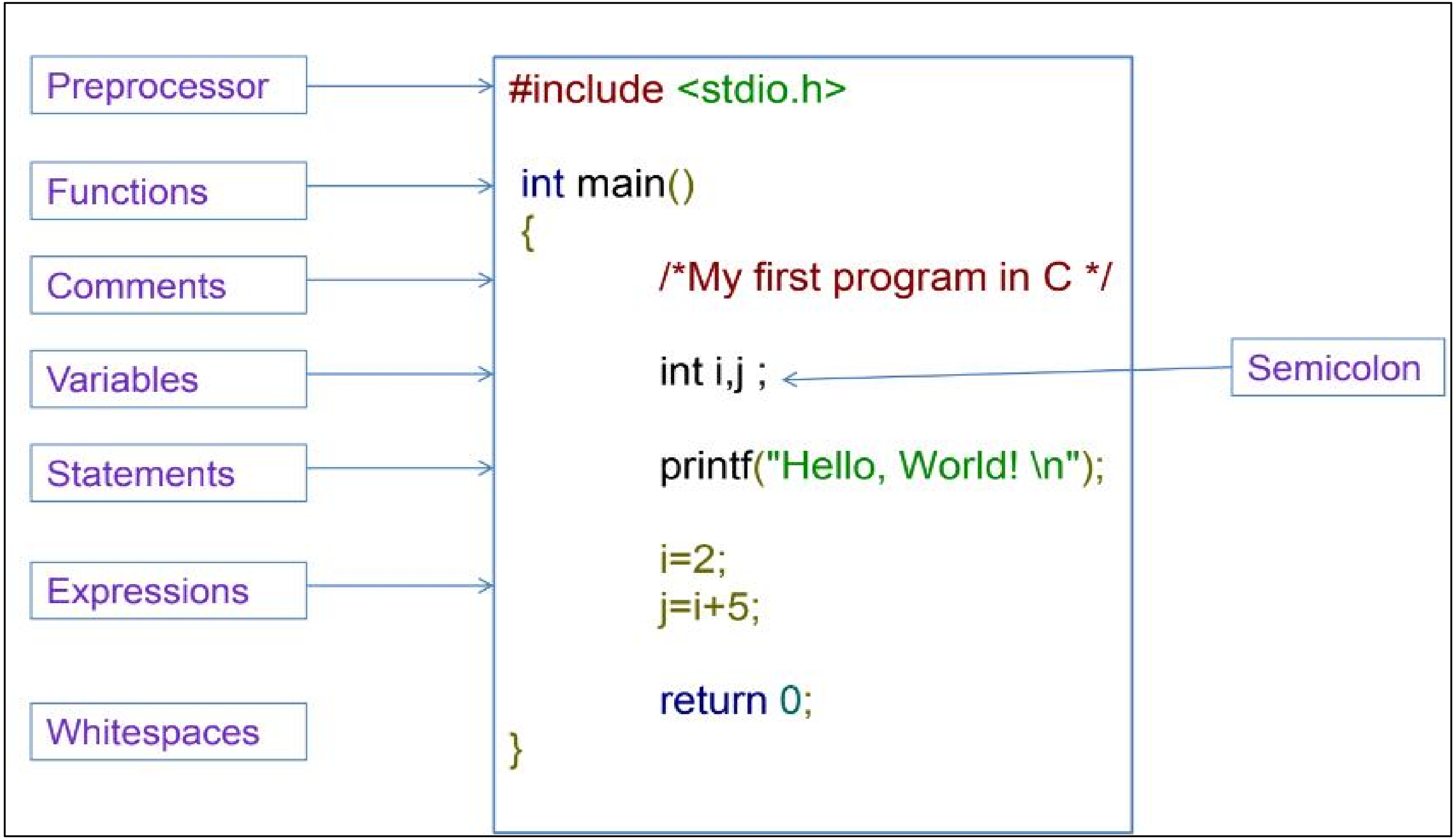
hello.obj

```
010101010
101010110
100111011
1010......
```

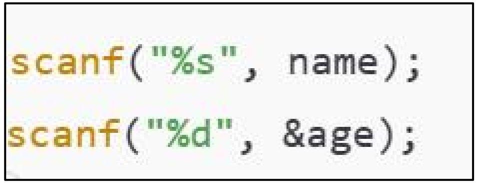
hello.exe

```
010101010
101010110
100111011
1010......
```

Components of a C Program



```
#include <stdio.h>

 int main()
 {
          /*My first program in C */

          int i,j ;

          printf("Hello, World! \n");

          i=2;
          j=i+5;

          return 0;
}
```



```
scanf("%s", name);
scanf("%d", &age);
```

Explanation:

1. `scanf("%s", name);` **(No &)**
   - `name` is declared as a **character array** (`char name[20]`).
   - In C, the name of an array itself acts as a pointer to the first element of the array.
   - `scanf("%s", name);` already receives the memory address of the first character of `name`, so **no & is needed**.
2. `scanf("%d", &age);` **(Uses &)**
   - `age` is an **integer variable** (`int age`).
   - `scanf("%d", &age);` needs the **memory address** of `age` so that it can store the input value at that location.
   - Since `age` is not an array, using `&age` ensures we pass its address to `scanf`.

General Rule:

- **For arrays (like `char name[]`), do NOT use & because the array name already acts as a pointer.**
- **For normal variables (like `int age`), use & to pass the memory address.**