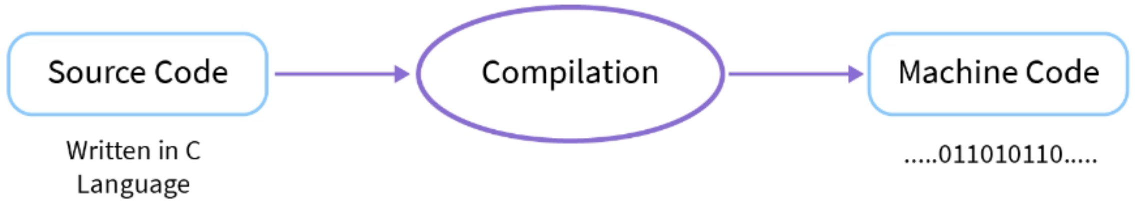
Compilation Process in C

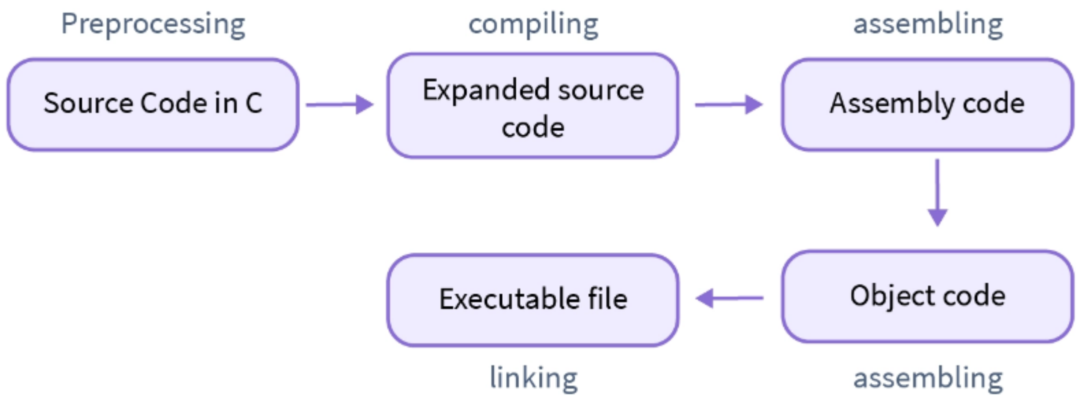
What is a Compilation?

The compilation process in C is converting an understandable human code into a Machine understandable code and checking the syntax and semantics of the code to determine any syntax errors or warnings present in our C program. Suppose we want to execute our C Program written in an IDE (Integrated Development Environment). In that case, it must go through several phases of compilation (translation) to become an executable file that a machine can understand.



Compilation process in C involves four steps:

- **Preprocessing**
- **Compiling**
- **Assembling**
- **Linking**



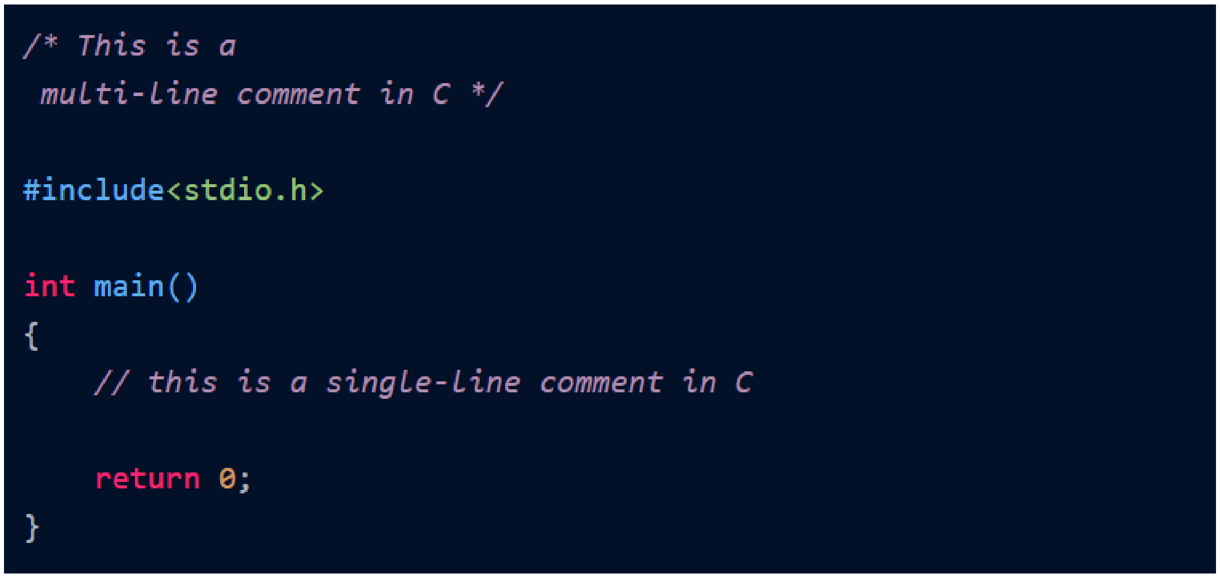
The Compilation Process in C

a. Pre-Processing

Pre-processing is the first step in the compilation process in C performed using the *pre-processor tool* . All the statements starting with the # symbol in a C program is processed by the pre-processor, and it converts our program file into an intermediate file with no # statements. Under following pre-processing tasks are performed :

i. Comments Removal

Comments in a C Program are used to give a general idea about a particular statement or part of code, comments are the part of code that is removed during the compilation process by the pre-processor as they are not of particular use for the machine. The comments in the below program will be removed from the program when the pre-processing phase is complete.



```c
/* This is a
 multi-line comment in C */

#include<stdio.h>

int main()
{
    // this is a single-line comment in C

    return 0;
}
```

ii. Macros Expansion

Macros are some constant values or expressions defined using the **#define** directives in C Language. A macro call leads to the macro expansion. The pre-processor creates an intermediate file where some pre-written assembly level instructions replace the defined expressions or constants (basically matching tokens). To differentiate between the original instructions and the assembly instructions resulting from the macros expansion, a '+' sign is added to every macros expanded statement.

iii. File inclusion

File inclusion in C language is the addition of another *file* containing some pre-written code into our C Program during the pre-processing. It is done using the **#include** directive. File inclusion during pre-processing causes the entire content of **filename** to be added to the source code, replacing the **#include<filename>** directive, creating a new intermediate file.

Example:

If we have to use basic input/output functions like printf() and scanf() in our C program, we have to include a pre-defined **standard input output header file** i.e. **stdio.h**.

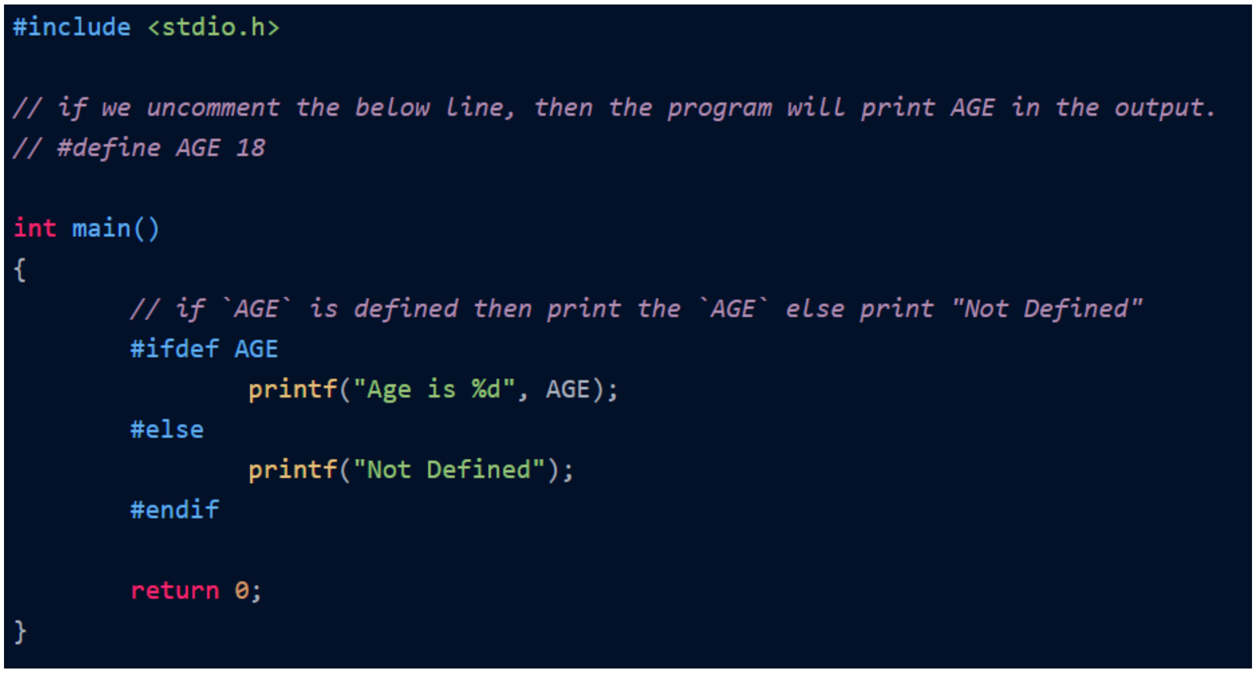
```
#include <stdio.h>
```

iv. Conditional Compilation

Conditional compilation is running or avoiding a block of code after checking if a **macro** is defined or not (a constant value or an expression defined using #define). The preprocessor replaces all the conditional compilation directives with some pre-defined assembly code and passes a newly expanded file to the compiler. Conditional compilation can be performed using commands like #ifdef, #endif, #ifndef, #if, #else and #elif in a C Program.

Example :

Printing the AGE macro, if AGE macro is defined, else printing Not Defined and ending the conditional compilation block with an #endif directive.



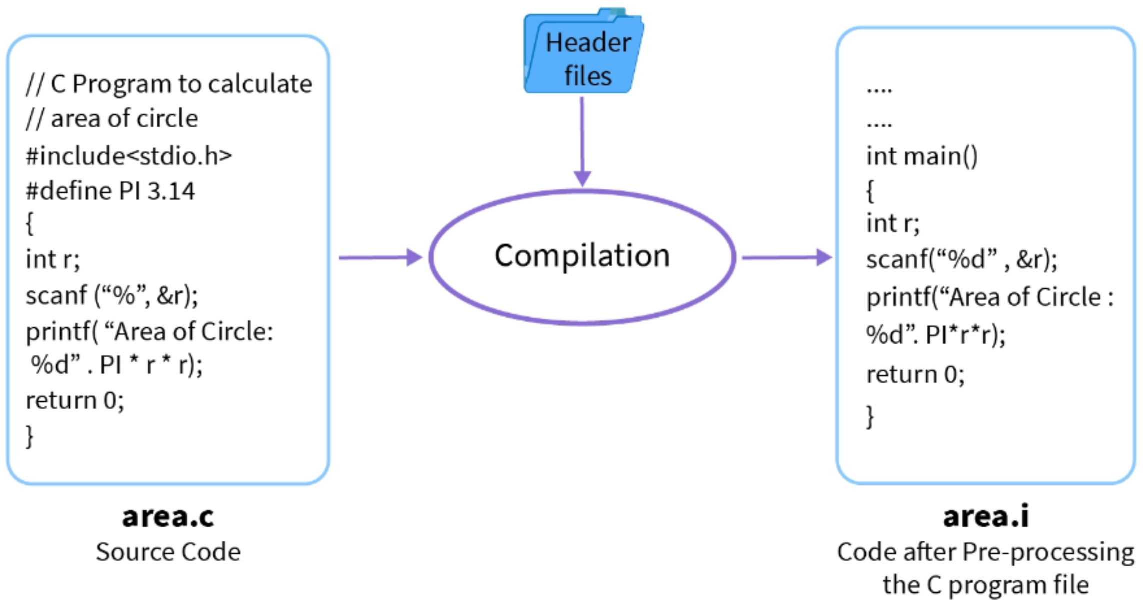
```
#include <stdio.h>

// if we uncomment the below line, then the program will print AGE in the output.
// #define AGE 18

int main()
{
	// if `AGE` is defined then print the `AGE` else print "Not Defined"
	#ifdef AGE
		printf("Age is %d", AGE);
	#else
		printf("Not Defined");
	#endif

	return 0;
}
```

How a pre-processor converts our source code file into an intermediate file.



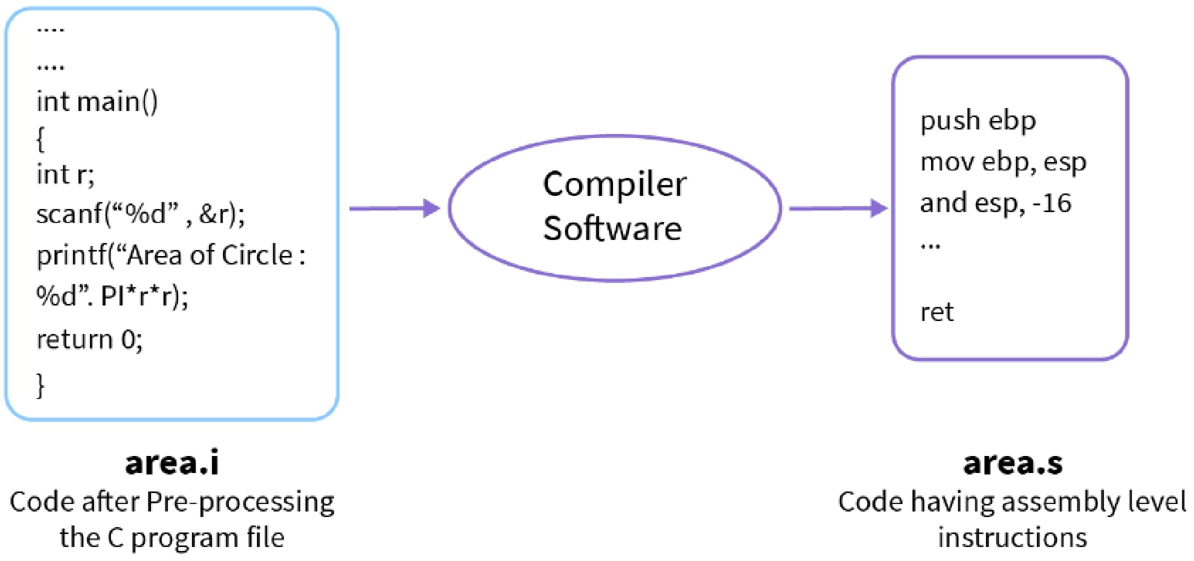
```
// C Program to calculate
// area of circle
#include<stdio.h>
#define PI 3.14
{
int r;
scanf ("%", &r);
printf( "Area of Circle:
 %d" . PI * r * r);
return 0;
}
```

```
....
....
int main()
{
int r;
scanf("%d" , &r);
printf("Area of Circle :
%d". PI*r*r);
return 0;
}
```

b. Compiling

Compiling phase in C uses an inbuilt *compiler software* to convert the intermediate (.i) file into an **Assembly file** (.s) having assembly level instructions (low-level code). To boost the performance of the program C compiler translates the intermediate file to make an assembly file.

The below image shows an example of how the compiling phase works.



Compiler Software
area.i
Code after Pre-processing the C program file
area.s
Code having assembly level instructions

```
....
....
int main()
{
int r;
scanf("%d" , &r);
printf("Area of Circle :
%d". PI*r*r);
return 0;
}
```

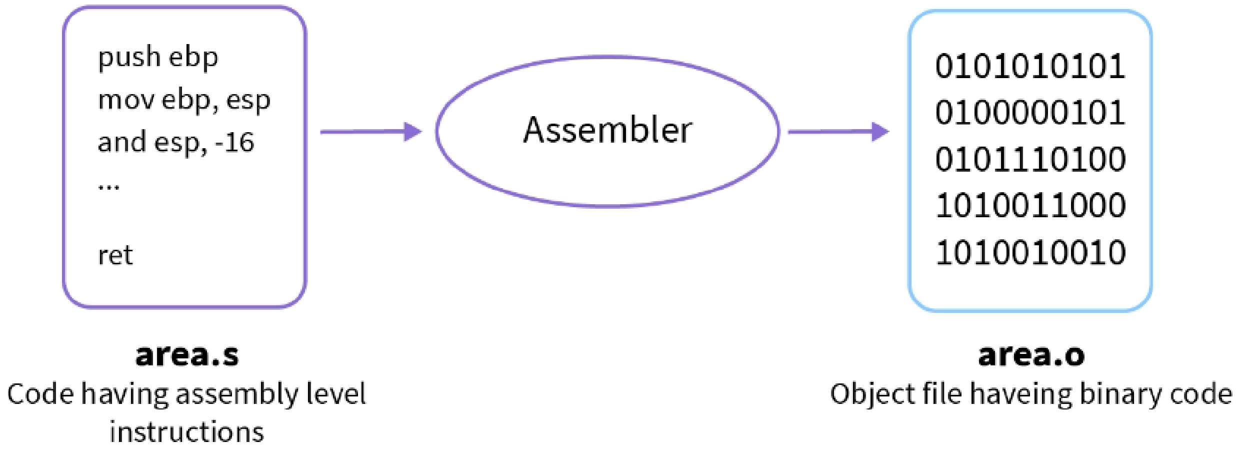
```
push ebp
mov ebp, esp
and esp, -16
...

ret
```

c. Assembling

Assembly level code (.s file) is converted into a machine-understandable code (in binary/hexadecimal form) using an *assembler*. Assembler is a pre-written program that translates assembly code into machine code. It takes basic instructions from an assembly code file and converts them into binary/hexadecimal code specific to the machine type known as the object code.

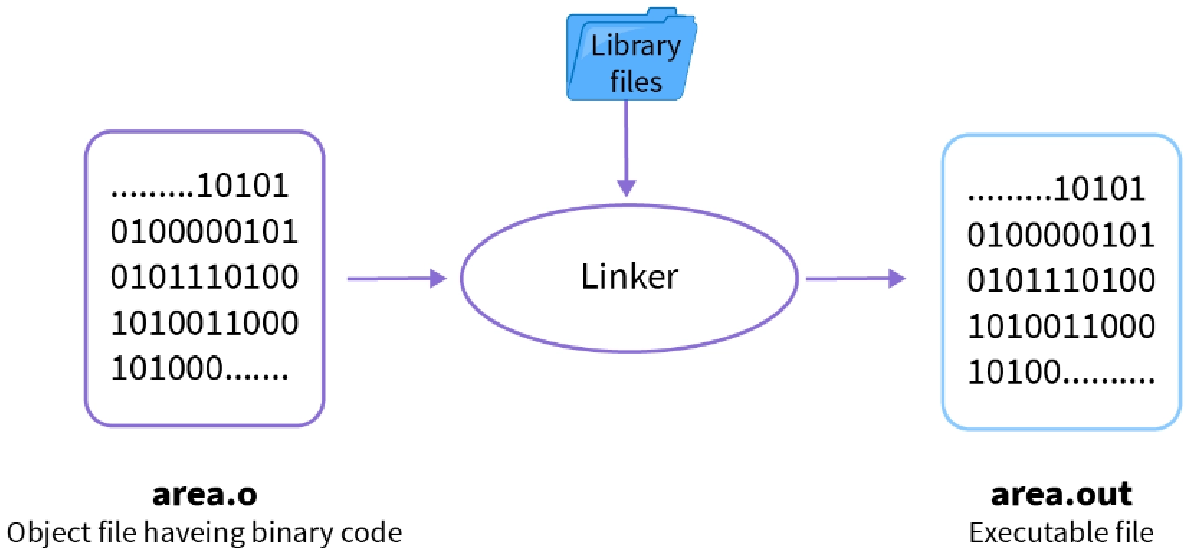
The below image shows an example of how the assembly phase works. An assembly **file area.s is translated to an object file area.o** having the same name but a different extension.



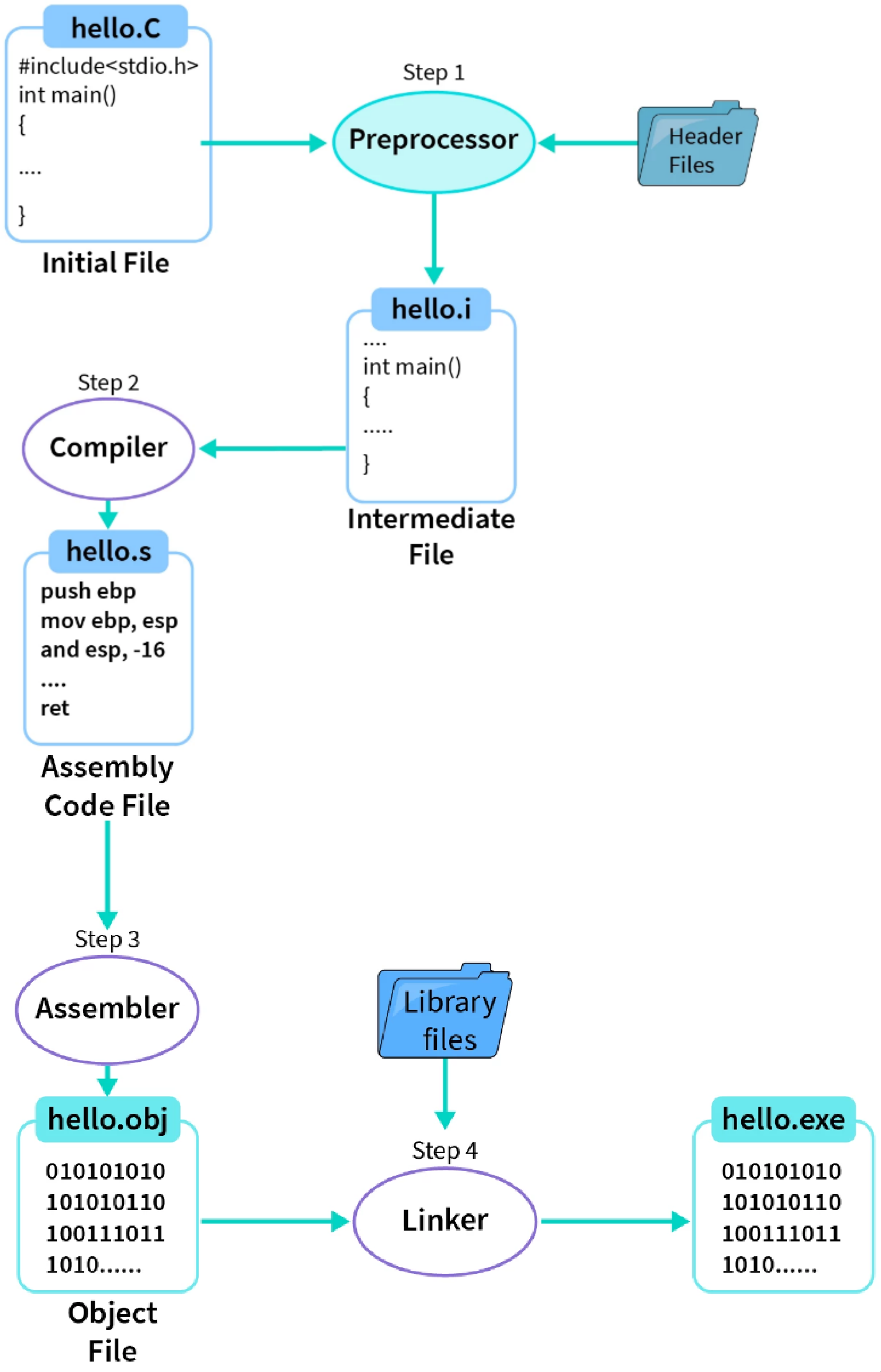
d. Linking

Linking is a process of including the library files into our program. *Library Files* are some predefined files that contain the definition of the functions in the machine language and these files have an extension of .lib. Some unknown statements are written in the object (.o/.obj) file that our operating system can't understand. The linking process generates an **executable file** with an extension of **.exe** in DOS and **.out** in UNIX OS.

The below image shows an example of how the linking phase works, and we have an object file having machine-level code, it is passed through the linker which links the library files with the object file to generate an executable file.



Flow Diagram of the Program



Ahmed Mostafa