Table of Contents

What is Programming?

A simple answer would be, “Programming is the act of instructing computers to carry out certain tasks.” It is often referred to as **coding**

So then, what is a **computer program**?

A program is a sequence of instructions that tell a computer how to do a task. When a computer follows the instructions in a program, we say it executes the program. You can think of it like a recipe that tells you how to make a peanut butter sandwich. ***In this model, you are the computer, making a sandwich is the task, and the recipe is the program that tells you how to execute the task.***

Activity: Come up with a sequence of instructions to tell someone how to make a peanut butter sandwich. Don't leave any steps out or put them in the wrong order.

A computer in the definition above is any device that is capable of processing code. These could be smartphones, ATMs, the Raspberry Pi, and Servers to name a few.

Some More Non-Technical Examples

First, there are patterns in our everyday lives. The universe operates in a somewhat predictable way; For example — day and night, seasons, sunrise and sunset. People go through routines such as rising in the morning, going to school or to work.

We get instructions from other people such as our superiors at work. How we cook certain recipes can be explained in finite steps.

Second, every time we use smart devices, some code is running in the background. Moving a mouse pointer from one part of your computer screen to the other may seem like a simple task, but in reality, so many

lines of code just ran. An act as simple as typing letters into Google Docs leads to lines of code being executed in the background. It's all code everywhere.

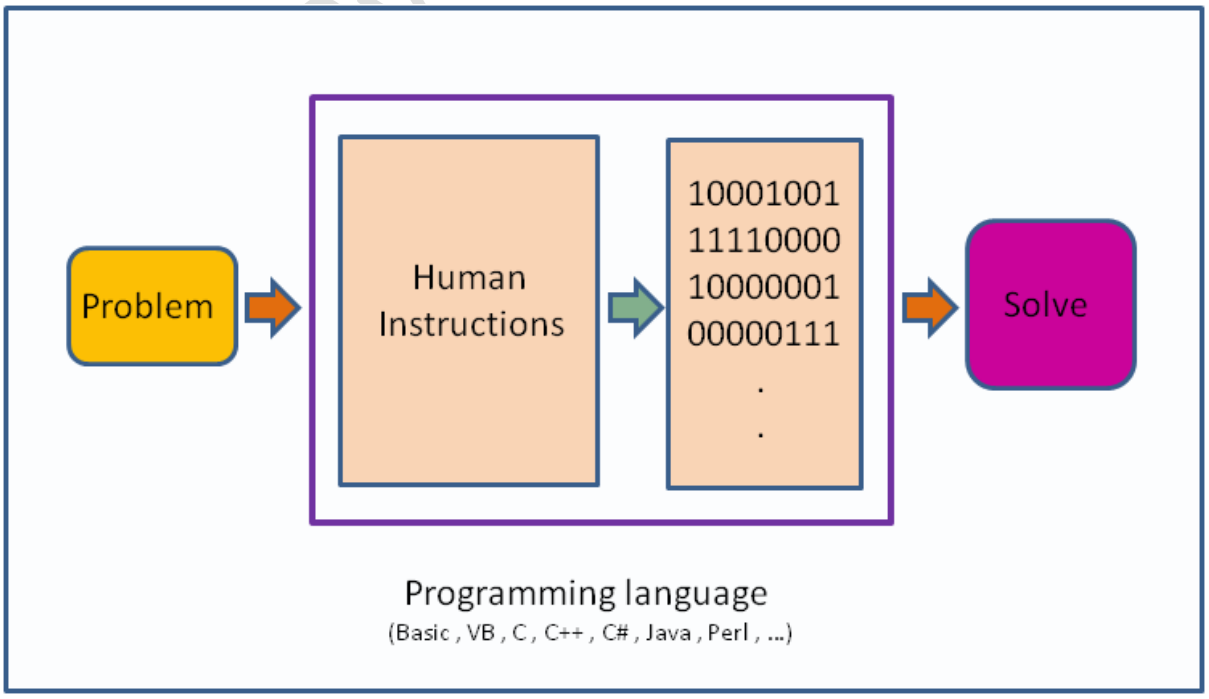
The Natural Language of the Computer

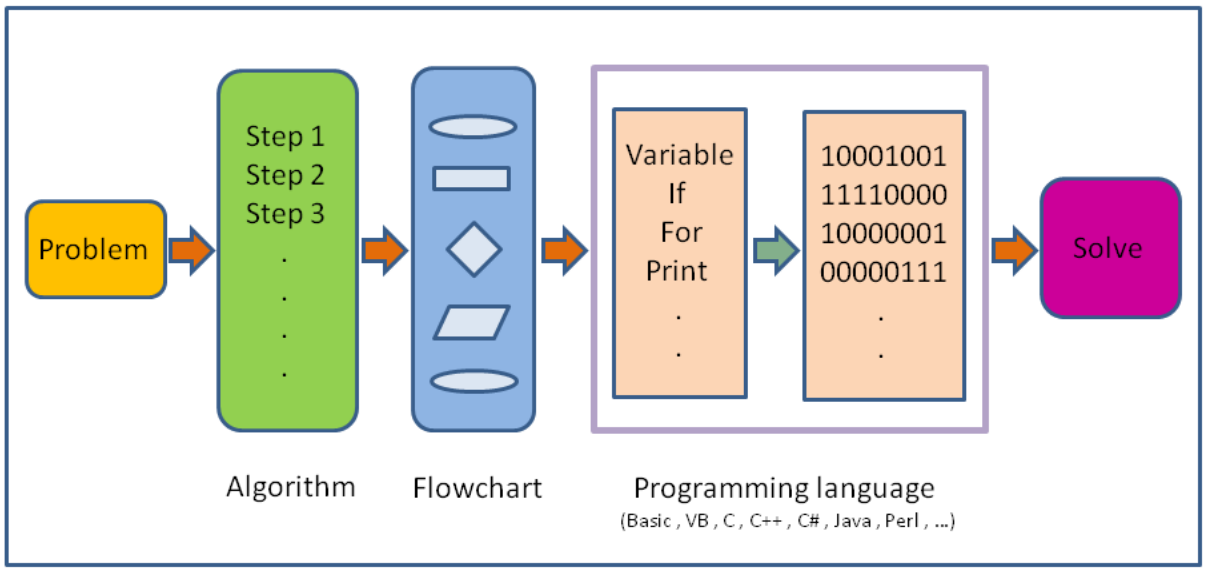
Machines have their natural language like humans do. Computers do not understand the human language. The natural language of computers is the binary code — 1 and 0. These represent two states: **on (1)** and **off (0)**.

That is the natural language of electronic equipment. It would be hectic for us as humans to communicate with the computer in binary.

Of course, computers don't understand recipes written on paper. Everything that a computer does is implemented in this most basic of all numbering systems—binary.

If you really wanted to tell a computer what to do directly, you'd have to talk to it in binary, giving it coded sequences of 1s and 0s that tell it which instructions to execute. However, this is nearly impossible. In practice, we use a programming language.





What is a Programing Language?

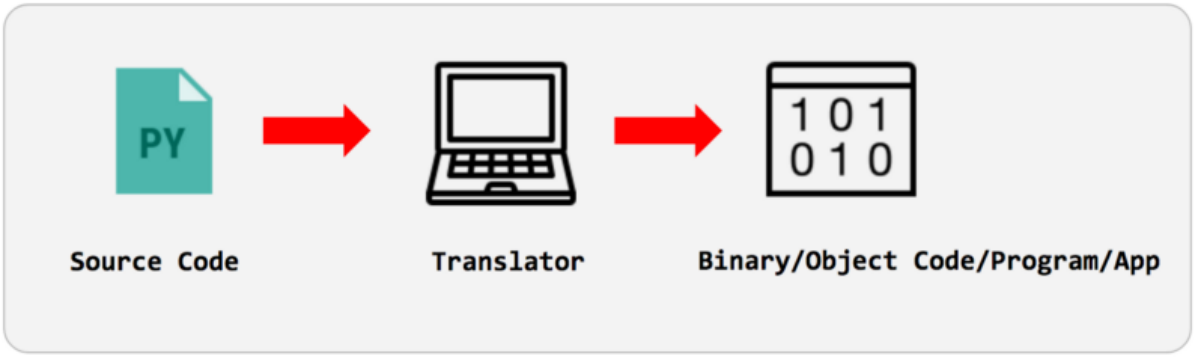
To communicate with machines who speak binary, we do so in a language that's closer to our own natural language. Such as English, French, Swahili, or Arabic. Programming languages are close to our natural languages. But they are more structured and must be thoroughly learned.

They could be high-level or low-level languages. High-level programming languages are farther away from the machine language than low-level languages. This "farther away" is usually called an **abstraction**.

The computer needs a way to understand our human language. To do this, we'll need a translator.

What are Translators

Translators have the responsibility of converting your source code to the machine language. This is also known as **binary**. Remember ones and zeros. We may refer to the binaries as **Object Code** , the Program or a common word today: **App**.



Translators can be any of:

- Interpreters
- Compilers
- A hybrid of Interpreters and Compilers
- Assemblers

Interpreters

Some languages are interpreted. The translator processes the source code line by line and runs every line in the final program or app. This means that interpreted source code starts running until it encounters an error. Then the interpreter stops to report such errors. Python is a good example of an interpreted programming language

Compilers

Compilers function differently. They convert the source code in its entirety via a compilation process to binary. The binary is then executed. If there were errors in the source code, they are detected during the compilation time and flagged. This interrupts the compilation process, and no binary is generated.

Interpreters translate line by line and execute the line before going on to the next line. Compilers translate all lines of a program to a file (binary) and execute the whole file.

Remember the definition of computer program? It's a sequence of instructions that is executed by a computer.

An executing program is usually called a process. Such programs use certain resources on the computer system or smartphone such as memory, disk space and the file system. An executing program can also be said to be **running**.

Compilation vs Interpretation

A compiled program has to be modified into machine code before it is used. The binary is then permanently stored. As an analogy, think of a novel that was written in one language and then translated into another. For example, the Harry Potter novels were written in British English, and were then subsequently translated into 67 other languages, including Hindi, Latvian, and Latin.

In much the same way, a computer program can be compiled (or "translated") into machine code, and it may potentially be compiled into different architectures (or "dialects") of machine code to suit different computers. Each translation will be a unique version of the program, in the same way that each translated book is a unique version of the original novel.

To take the analogy further, if I was fortunate enough to have written the first Harry Potter novel, it may be the case that I wouldn't understand the language into which it is translated. Thus, I could be given the Latvian version of the novel, and I could reasonably surmise that it tells the same story as the British English one, but I would be unable to read it. In the same sense, the version of my program that has been compiled into machine code might be impossible for me to read: it is said to be

"machine-readable", in that the computer can understand it, but it is far from easily readable for humans.

An interpreted program is stored in a human-readable form. When the program is executed, an interpreter modifies the human-readable content as it is run. This is analogous to the role that a human interpreter performs. For example, rather than translating the British English version of Harry Potter into Latvian and then providing the Latvian version to someone who understands the language, (as per compilation), we could hire a translator who knows both British English and Latvian.

The translator may choose to read each line from the British English novel, translate each line (one at a time) into Latvian, and, as each line is translated, relate it to the listener.

The computer interpreter performs the same function: it reads an instruction in one programming language, translates it into machine code, and then executes the machine code version. Once that instruction is out of the way it moves along to the next, performing exactly the same task, in much the same way that the interpreter of the Harry Potter novel would move on to the next line once the first has been related.

There are advantages for both types of software development. As a generalization, compiled programs are faster to run but slower to develop. Compiled programs often run faster because the computer only needs to execute the previously translated instructions. In interpreted languages, every time the program is run the computer also needs to translate each of the instructions. This translation causes a delay, slowing the execution of the program.

On the other hand, interpreted languages are often written in a smaller time frame, because the languages are simpler and the whole program does not need to be compiled each time a new feature is bug tested.

Hybrid Translators

A hybrid translator is a combination of the Interpreter and Compiler. A popular hybrid programming language is **Java**. Java first compiles your source code to an intermediate format known as the **Bytecode**.

The Bytecode is then interpreted and executed by a runtime engine also known as a Virtual machine. This enables the hybrid translators to run the bytecode on various operating systems.

Assemblers

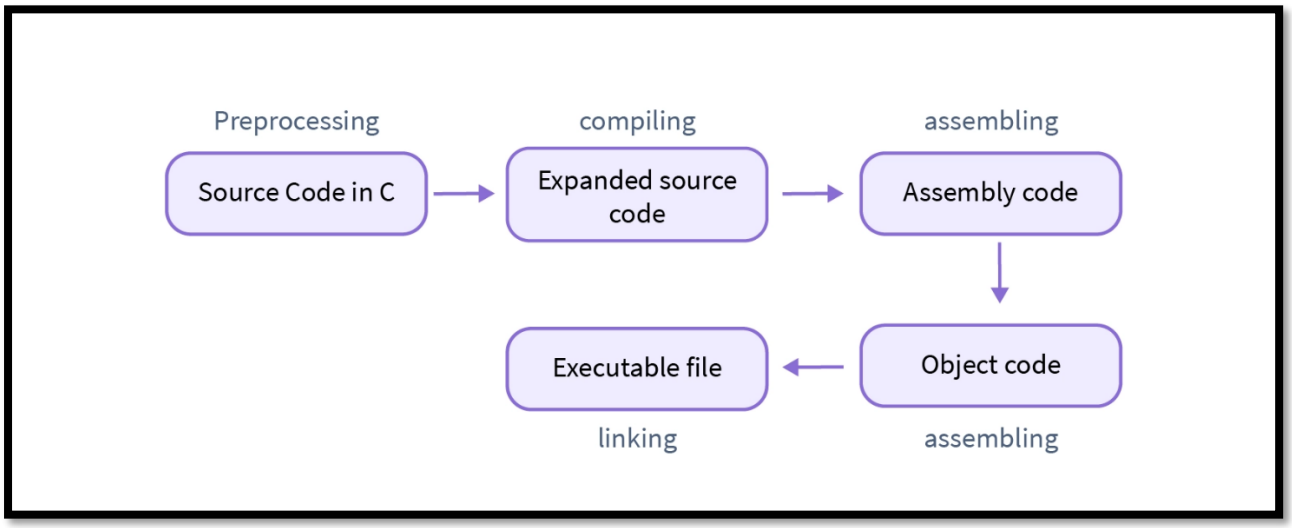
There's the Assembler as well for translating low-level Assembly language to binary.

Assembly language is easier to use than machine language. An assembler is useful for detecting programming errors. Programmers do not have the absolute address of data items. Assembly language encourage modular programming.

Compilation Process in C

Compilation process in C involves four steps:

1. Preprocessing
2. Compiling
3. Assembling
4. Linking



a. Pre-Processing

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

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

Macros Examples:

Defining a value

```
#define G 9.8
```

Defining an expression

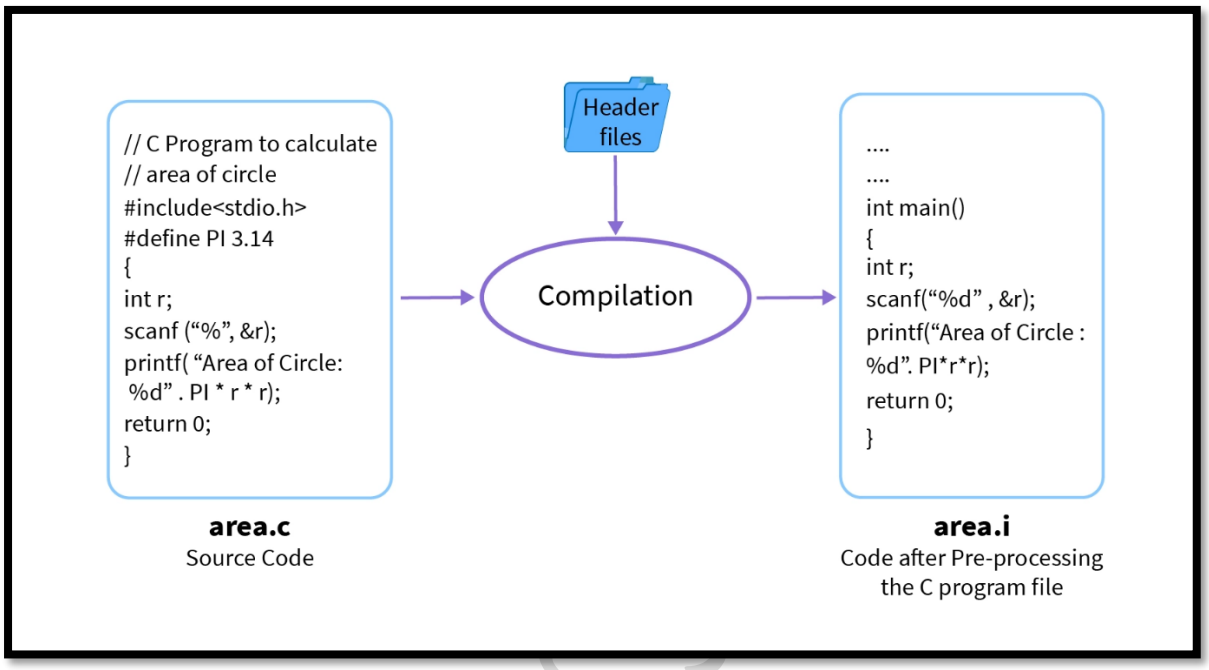
```
#define SUM(a,b) (a + b)
```

iii. File inclusion

File inclusion in C language is addition of an 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 in the source code replacing the #include<filename> directive and it creates 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>
```

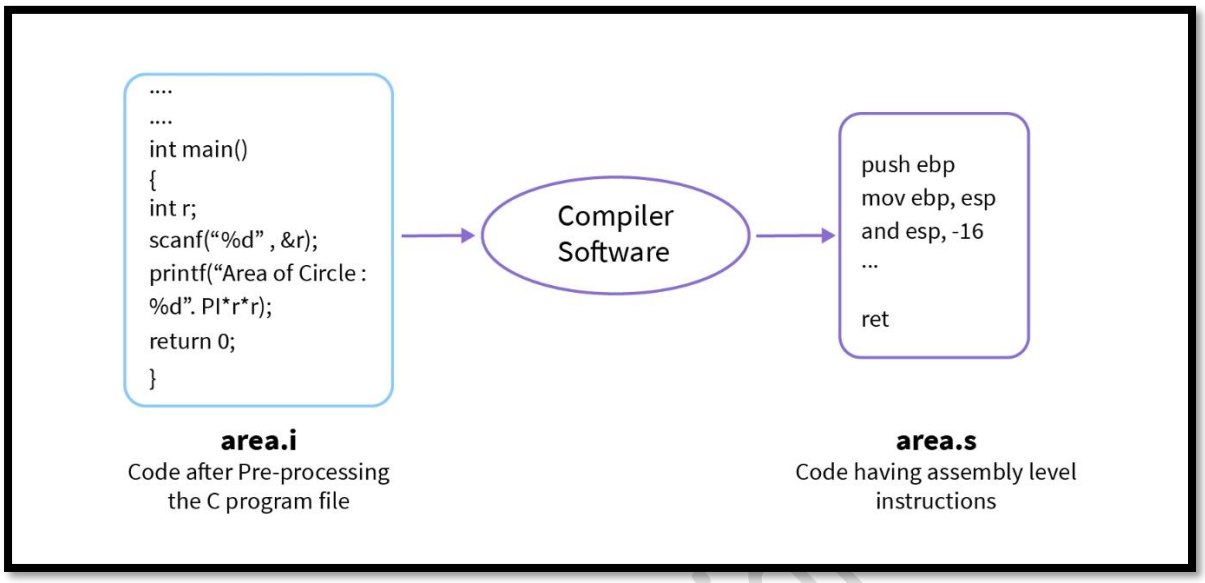


b. Compiling

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 compiler translates the intermediate file to make an assembly file.

Assembly code is a simple English type language and is used to write low level instructions (in micro-controller programs we use assembly language). The whole program code is parsed (syntax analysis) by the compiler software in one go and it tells us about any **syntax errors** or **warnings** present in the source code through the terminal window.

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

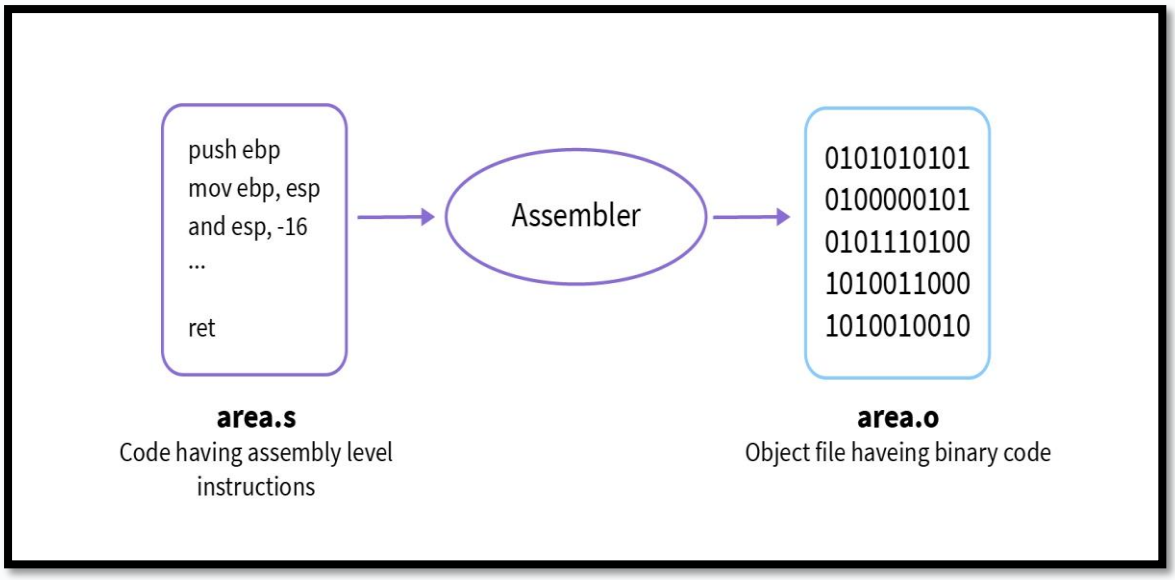


c. Assembling

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 assembly code file and converts them into binary/hexadecimal code specific to the machine type known as the object code.

The file generated has the same name as the assembly file and is known as an **object file** with an extension of .obj in DOS and .o in UNIX OS.

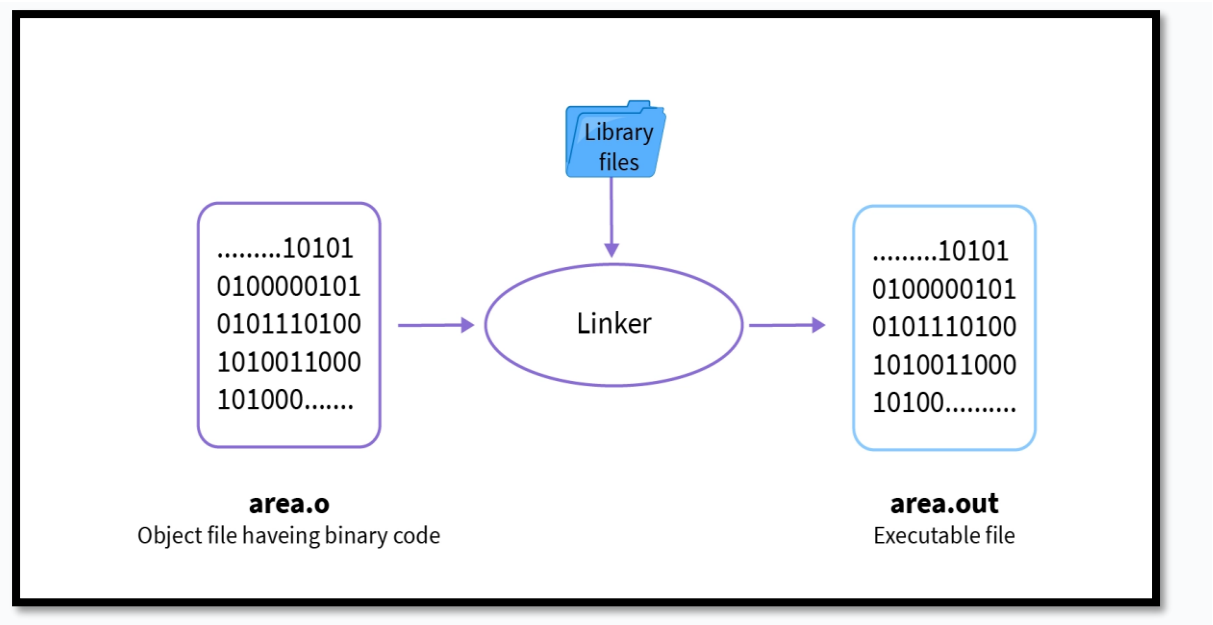
The below image shows an example of how the assembly phase works, an assembly file area.s is translated to an object file area.o having same name but different extension.



d. Linking

Linking is a process of including the library files into our program. *Library Files* are some predefined files that contains the definition of the functions in the machine language and these files have an extension of .lib. There are some unknown statements written in the object (.o/.obj) file that our operating system can't understand, you can understand this as a book having some words that you don't know, you will use a dictionary to find the meaning of those words, similarly we use *Library Files* to give meaning to some unknown statements from our object file. 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, 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.



Our First End to End Example

C program to display Hello World! on the output screen.

```c
// Simple Hello World program in C
#include<stdio.h>

int main()
{
    // printf() is a output function which prints
    // the passed string in the output console
    printf("Hello World!");

    return 0;
}
```

OUTPUT:

```
Hello World!
```

This tiny Hello World! program has to go through several steps of the compilation process to give us the output on the screen.

Explanation:

1. To compile the above code use this command in the terminal : **gcc hello.c -o hello**
2. First, the pre-processing of our C Program begins, comments are removed from the program, as there are no macros directives in this program so macro expansion doesn't happen, also we have included a **stdio.h** header file and during pre-processing, declarations of standard input/output functions like printf(), scanf() etc. is added in our C Program.
3. Now the during the compilation phase of our program, all the statements are converted into assembly level instructions using the compiler software.
4. Assembly level instructions for the above program (hello.s file) :

```
        .file   "hello.c"
        .def    ___main;        .scl    2;      .type   32;     .endef
        .section .rdata,"dr"
LC0:
        .ascii "Hello World!\0"
        .text
        .globl  _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
LFB12:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
LFE12:
        .ident  "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"
        .def    _printf;        .scl    2;      .type   32;     .endef
```

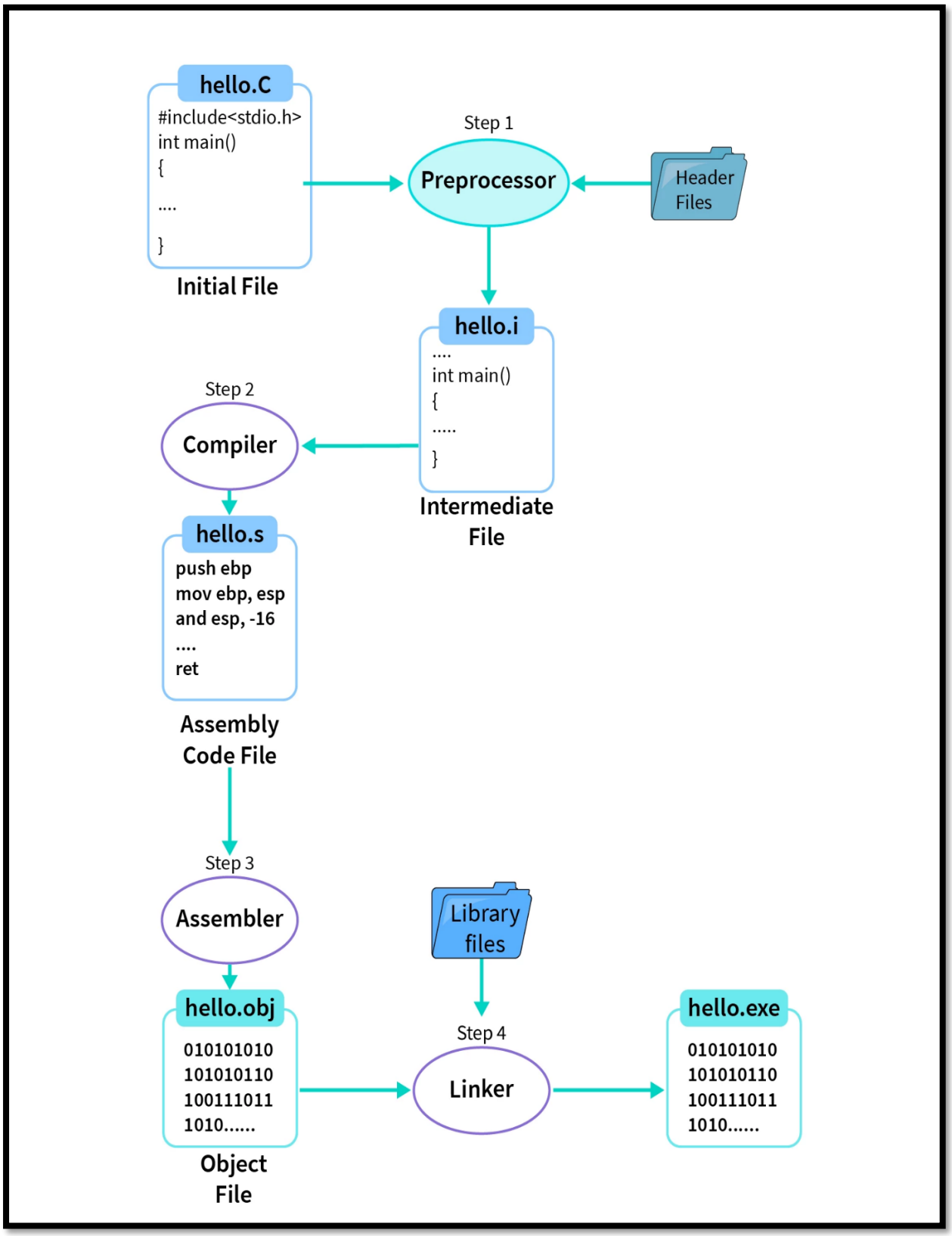
- You can get the above hello.s file using the command:

  g++ -S hello.c in the terminal.

- hello.s file is converted into binary code using the assembler program and generates an object file **hello.obj** in DOS and **hello.o** in UNIX OS.
- Now, the linker adds required definitions into the object file using the library files and generates an executable file **hello.exe** in DOS and **hello.out** in UNIX OS.
- When we run **hello.exe/hello.out**, we get a Hello World! output on the screen.

Flow Diagram of the Program

Let us look at the flow diagram of a program in the compilation process in C:



- We have a C Program file with an extension of **.c** i.e. **hello.c** file.
- **Step 1** is **preprocessing** of header files, all the statements starting with **#** (hash symbol) and comments are replaced/removed during the pre-processing with the help of a pre-processor. It generates an intermediate file with **.i** file extension i.e. a **hello.i** file.
- **Step 2** is **compilation** of **hello.i** file, compiler software translates the **hello.i** file to **hello.s** file having assembly level **instructions (low level code)**.
- **Step 3**, **assembly level** code instructions are converted into a machine understandable code (binary/hexadecimal form) by the assembler and the file generated is known as the object file with an extension of **.obj/.o** i.e. **hello.obj/hello.o** file.
- **Step 4**, **Linker** is used to link the library files with the object file to define the unknown statements. It generates an executable file with .exe/.out extension i.e. a **hello.exe/hello.out** file.
- Next, we can run the **hello.exe/hello.out** executable file to get the desired output on our output window i.e. **Hello World!.**

Conclusion

- Compilation process in C is also known as a process of converting Human Understandable Code (*C Program*) into a Machine Understandable Code (*Binary Code*))
- Compilation process in C involves four steps: pre-processing, compiling, assembling and linking.
- Preprocessor tool helps in comments removal, macros expansion, file inclusion and conditional compilation. These commands are executed in the first step of the compilation process.
- Compiler software helps in boosting the performance of the program and translates the intermediate file to an assembly file.
- Assembler helps in converting the assembly file into an object file containing machine level code.
- Linker is used for linking the library file with the object file, it is the final step in compilation to generate an executable file.