PRACTICAL-1

Aim : Understand modules of compilation process with the help of program. (Preprocessor, Compiler, Assembler, Linker/Loader).

1. **Preprocessing** is the first pass of any compilation. It processes include-files, conditional compilation instructions and macros.

2. **Compilation** is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code.

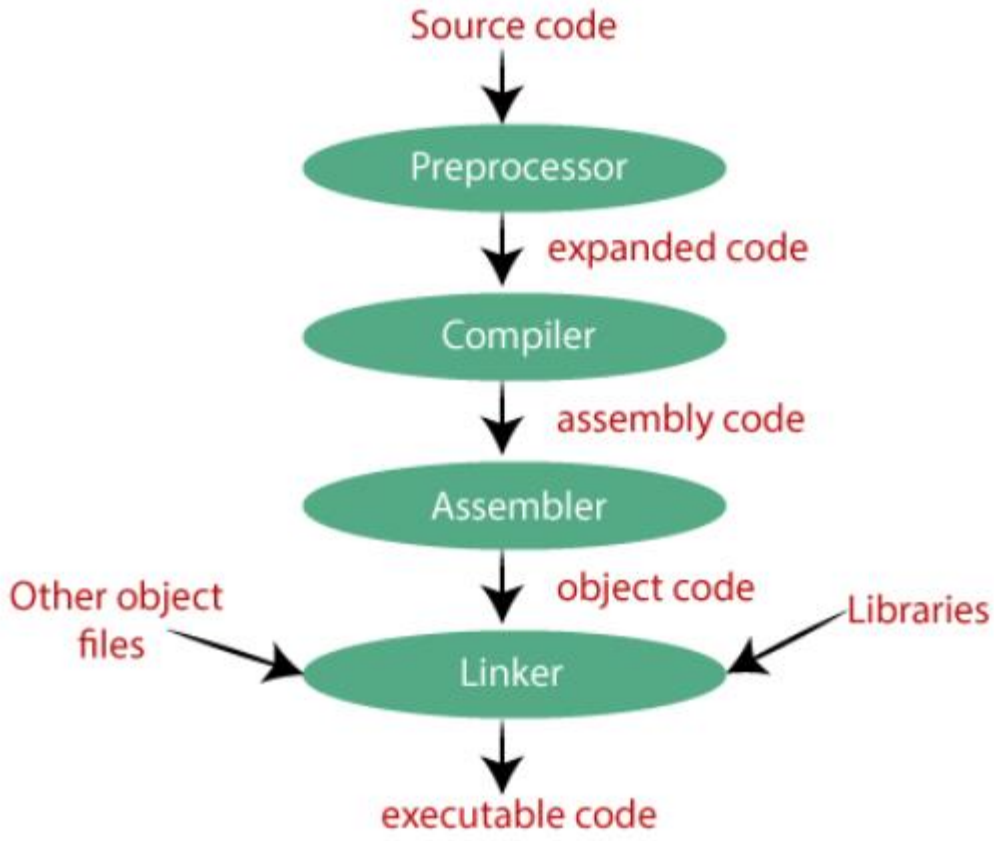
3. **Assembly** is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.

4. **Linking** is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single

The compilation is a process of converting the source code into object code.

The compilation process can be divided into four steps, i.e.,

- **Preprocessor**
- **Compiler**
- **Assembler**
- **Linker**



❖ **Preprocessor**

The C compilation begins with pre-processing of source file. Pre-processor is a small software that accepts C source file and performs below tasks.

- Remove comments from the source code.
- Macro expansion.
- Expansion of included header files.

In C , For Preprocessing Any C Code You Have to Write command,

Code: gcc -E cfile.c

It will Give Output in that preprocessor inserts content of header files to our source code file.Pre-processor generated file is larger than the original source file.

For storing o/p of Pre-Procesing ,

Output

```
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 840 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));



extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 868 "/usr/include/stdio.h" 3 4

# 2 "cfile.c" 2



# 4 "cfile.c"
void main()
{
 int a=10,b=20,c;
 c=a+10 +b;
 printf("c=%d\n",c);
 printf("Hello World");
}
~/IoT1$
```

- **Compiler**

In next phase of C compilation the compiler comes in action. It accepts temporary pre-processed <file-name>.i file generated by the pre-processor and performs following tasks.

- Check C program for syntax errors.
- Translate the file into intermediate code i.e. in assembly language.
- Optionally optimize the translated code for better performance.

For Compilation You have to write,

Code: gcc -S cfile.c
cat cfile.s

or to store compilation into other file ,

Output:

```
~/IoT1$ gcc -S cfile.c
~/IoT1$ cat cfile.s
        .file   "cfile.c"
        .text
        .section        .rodata
.LC0:
        .string "c=%d\n"
.LC1:
        .string "Hello World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $10, -12(%rbp)
        movl    $20, -8(%rbp)
        movl    -12(%rbp), %eax
        leal    10(%rax), %edx
        movl    -8(%rbp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
~/IoT1$
```

Code: gcc -S -o f1.asm cfile.c
cat f1.asm

After compiling it generates an intermediate code in assembly language as <file-name.s> file. It is assembly version of our source code.

Output:

```
~/IoT1$ gcc -S -o f1.asm cfile.c
~/IoT1$ cat f1.asm
        .file   "cfile.c"
        .text
        .section        .rodata
.LC0:
        .string "c=%d\n"
.LC1:
        .string "Hello World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $10, -12(%rbp)
        movl    $20, -8(%rbp)
        movl    -12(%rbp), %eax
        leal    10(%rax), %edx
        movl    -8(%rbp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
```

❖ Optimization

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

- Optimized code has faster execution speed.
- Optimized code utilizes the memory efficiently.
- Optimized code gives better performance.

For optimize code you have to write ,

Code: gcc -S -O -o f1.asm cfile.c
cat f1.asm

It will generate one optimize file whose name is we have mentioned in command . Content of this optimize file and <file-name.s> will be same but optimizer do some code optimization for faster execution . It is not mendatory that optimizer optimizes code each time but whenever it is required to do then optimizer will definitely do that .

Output:

```
~/IoT1$ gcc -S -O -o f1.asm cfile.c
~/IoT1$ cat f1.asm
        .file   "cfile.c"
        .text
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "c=%d\n"
.LC1:
        .string "Hello World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB23:
        .cfi_startproc
        subq    $8, %rsp
        .cfi_def_cfa_offset 16
        movl    $40, %edx
        leaq    .LC0(%rip), %rsi
        movl    $1, %edi
        movl    $0, %eax
        call    __printf_chk@PLT
        leaq    .LC1(%rip), %rsi
        movl    $1, %edi
        movl    $0, %eax
        call    __printf_chk@PLT
        addq    $8, %rsp
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE23:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
```

- Assembly

Moving on to the next phase of compilation. Assembler accepts the compiled source code (compilation.s) and translates to low level machine code. After successful assembling it generates <file-name.o> (in Linux) or <file-name.obj> (in Windows) file known as object file. In our case it generates the compilation.o file.

For generating Assembly code you have to write ,

Code: gcc -C cfile.s
cat a.out

It will generate .obj or .o file . This file is encoded in low level machine language and cannot be viewed using text editors. However, if you still open this in notepad, it look like.

Output:

- ❖ Linker/Loader

Finally, the linker comes in action and performs the final task of compilation process. It accepts the intermediate file <file-name.o> or <file-name.obj> generated by the assembler.

It links all the function calls with their original definition. Which means the function printf() gets linked to its original definition.

Linker generates the final executable file (.exe in windows).

For generating linker\loader file you have to type ,

- ➢ gcc -o name pr1.c
- ➢ ./name

Code: gcc -C cfile.s -o j1
cat j1

Output:

Code: ./j1

Output:

```
~/IoT1$ ./j1
c=40
Hello World~/IoT1$
```