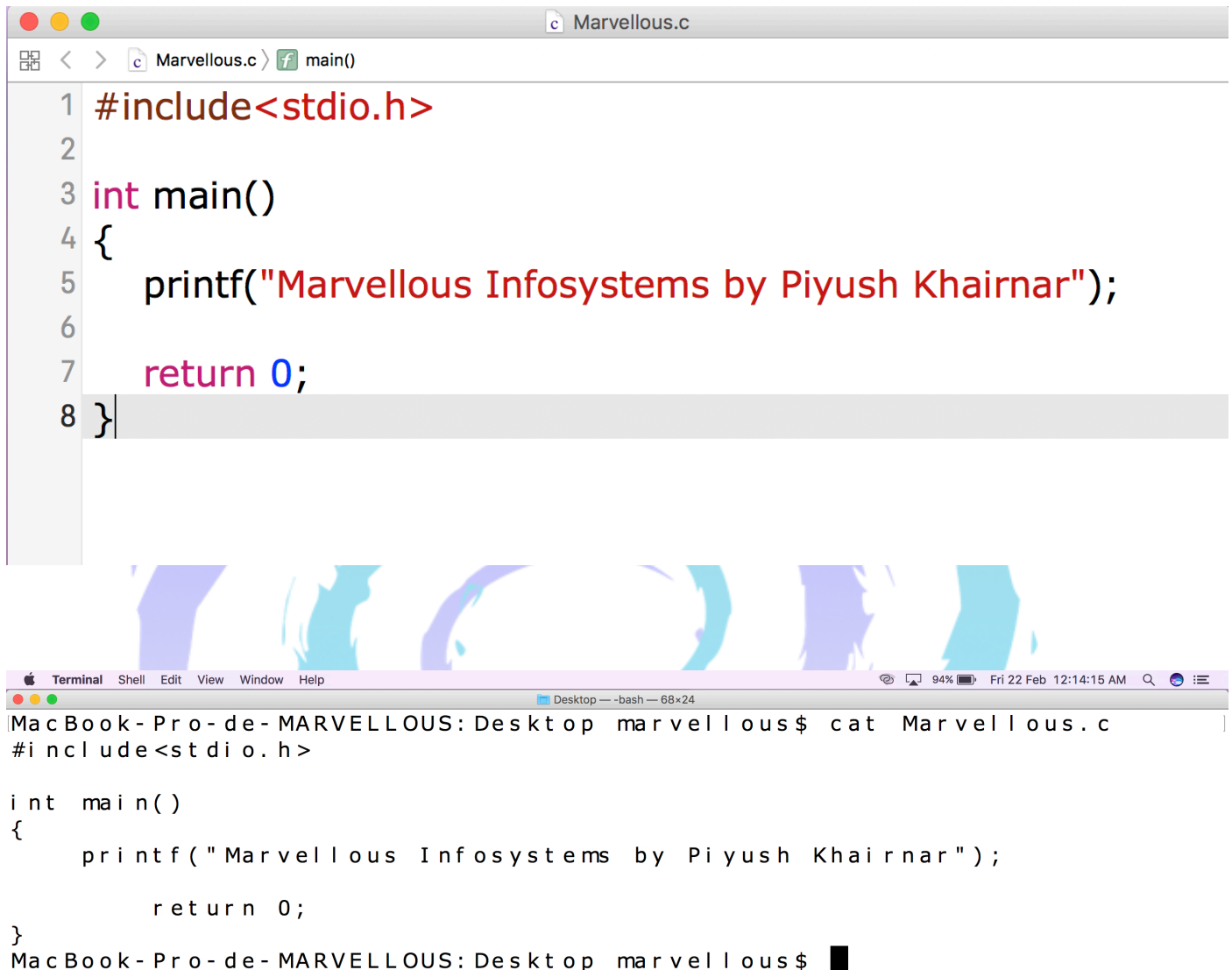


Build process of C Program using GCC toolchain

By using any editor (gedit, kwrite) write below C program that we want to refer to understand different phases of toolchain



```
1 #include<stdio.h>
2
3 int main()
4 {
5     printf("Marvellous Infosystems by Piyush Khairnar");
6
7     return 0;
8 }
```

```
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ cat Marvellous.c
#include<stdio.h>

int main()
{
    printf("Marvellous Infosystems by Piyush Khairnar");

    return 0;
}
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$
```

1.Preprocessor

During compilation of a C program the compilation is started off with preprocessing the directives (e.g., #include and #define).

The preprocessor (cpp - c preprocessor) is a separate program in reality, but it is invoked automatically by the compiler.

Preprocessor performs following tasks as

- File inclusion
- Macro expansion
- Conditional compilation
- Comment removal
- Extra Whitespace removal

For example, the #include <stdio.h> command in line 1 of Marvellous.c tells the preprocessor to read the contents of the system header file stdio.h and insert it directly into the program text.

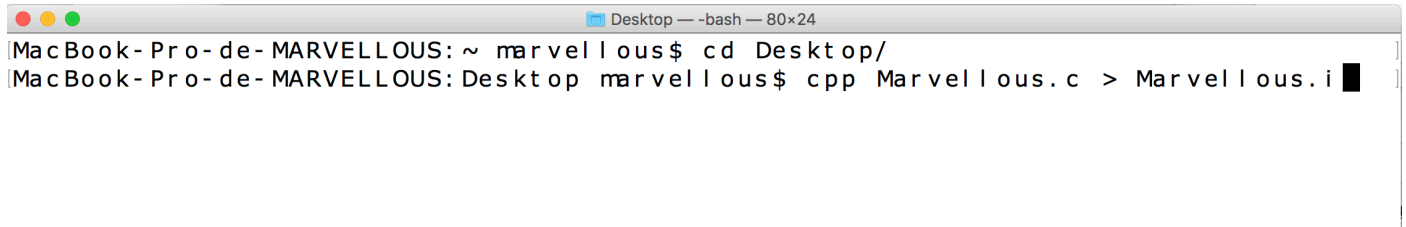
The result is another file typically with the .i suffix.

In practice, the preprocessed file is not saved to disk unless the -save-temps option is used.

This is the first stage of compilation process where preprocessor directives (macros and header files are most common) are expanded.

To perform this step gcc executes the following command internally.

[root@host ~]# cpp Marvellous.c > Marvellous.i

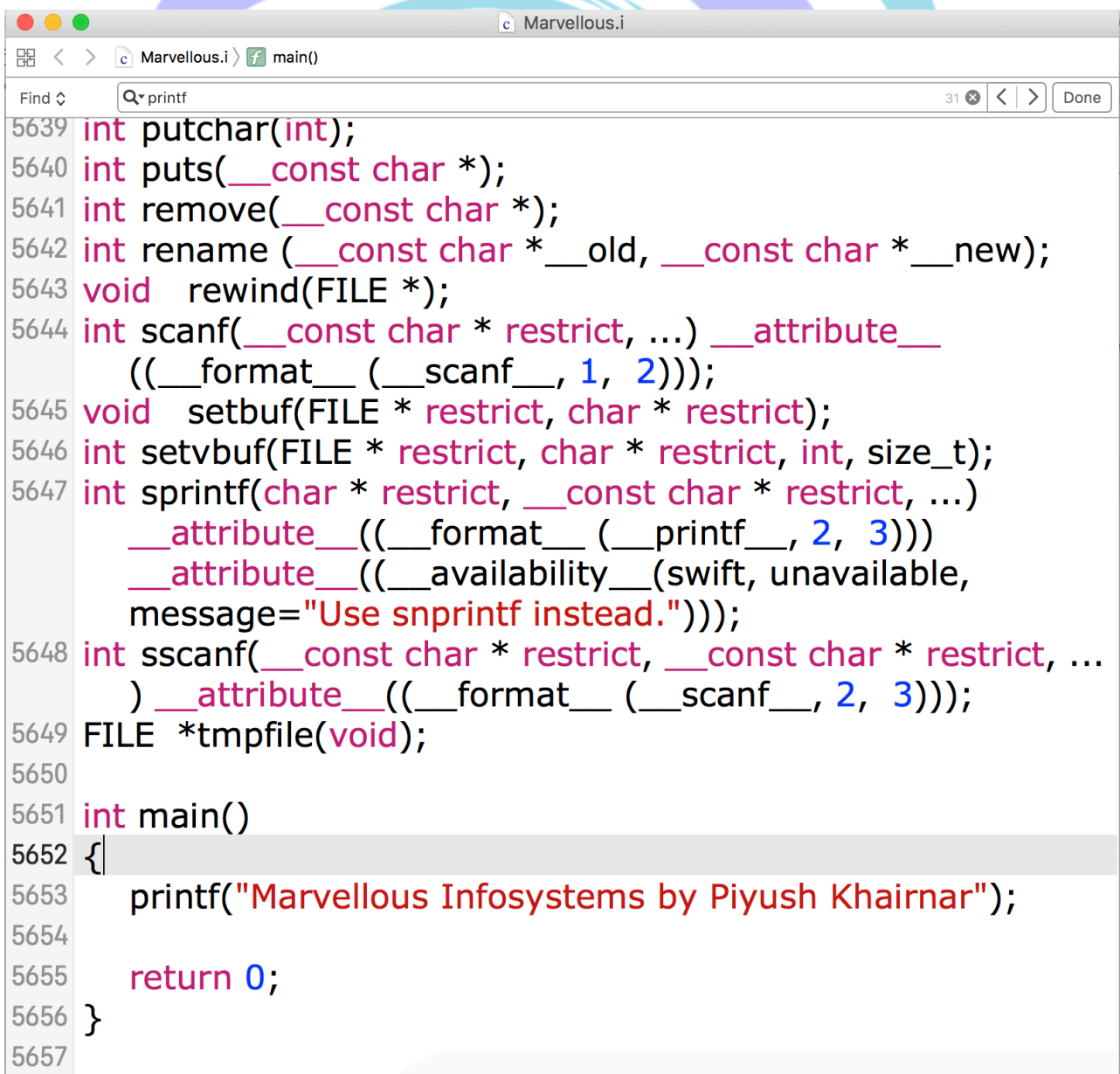


```

Desktop — -bash — 80x24
MacBook-Pro-de-MARVELLOUS: ~ marvellous$ cd Desktop/
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ cpp Marvellous.c > Marvellous.i
  
```

The result is a file Marvellous.i that contains the source code with all macros expanded.

If we execute the above command in isolation then the file Marvellous.i will be saved to disk and we can see its content by any editor.



```

Marvellous.i
main()
Find printf
31 Done
5639 int putchar(int);
5640 int puts(__const char *);
5641 int remove(__const char *);
5642 int rename (__const char *__old, __const char *__new);
5643 void rewind(FILE *);
5644 int scanf(__const char * restrict, ...) __attribute__
    ((__format__ (__scanf__, 1, 2)));
5645 void setbuf(FILE * restrict, char * restrict);
5646 int setvbuf(FILE * restrict, char * restrict, int, size_t);
5647 int sprintf(char * restrict, __const char * restrict, ...)
    __attribute__((__format__ (__printf__, 2, 3)))
    __attribute__((__availability__(swift, unavailable,
    message="Use snprintf instead.")));
5648 int sscanf(__const char * restrict, __const char * restrict, ...
    ) __attribute__((__format__ (__scanf__, 2, 3)));
5649 FILE *tmpfile(void);
5650
5651 int main()
5652 {
5653     printf("Marvellous Infosystems by Piyush Khairnar");
5654
5655     return 0;
5656 }
5657
  
```

2. Compiler

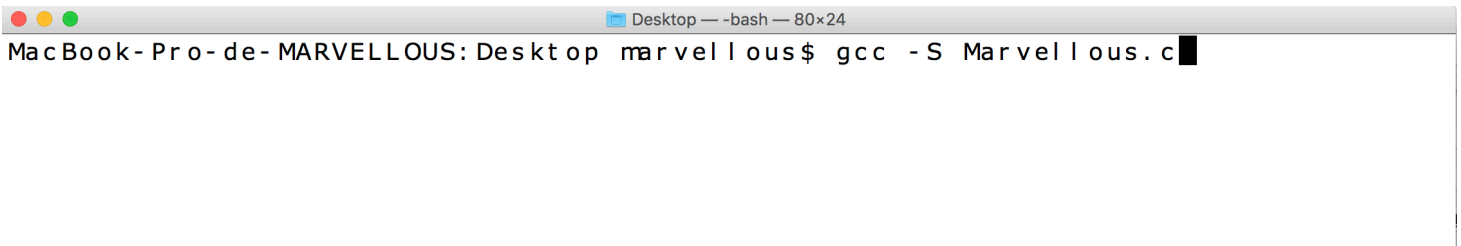
In this phase compilation proper takes place.

Compiler converts human understandable code to Machine dependent code ie inn assembly language.

The compiler translates Marvellous.i into Marvellous.s. File Marvellous.s contains assembly code.

We can explicitly tell gcc to translate Marvellous.i to Marvellous.s by executing the following command.

```
[root@host ~]# gcc -S Marvellous.s
```



```
Desktop — -bash — 80x24
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ gcc -S Marvellous.c
```

The command line option -S tells the compiler to convert the preprocessed code to assembly language without creating an object file.

After having created Marvellous.s we can see the content of this file.

While looking at assembly code we may note that the assembly code contains a call to the external function printf.

3. Assembler

Here, the assembler (as) translates Marvellous.s into machine language instructions, and generates an object file Marvellous.o.

We can invoke the assembler at our own by executing the following command.

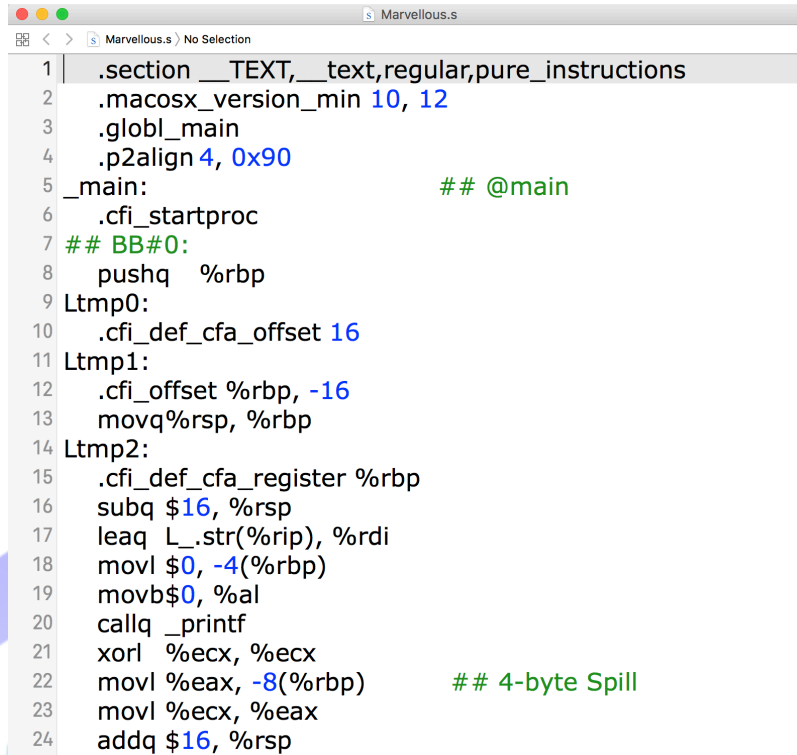
```
[root@host ~]# as Marvellous.s -o Marvellous.o
```



```
Desktop — -bash — 80x24
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ as Marvellous.s -o Marvellous.o
```

The above command will generate Marvellous.o as it is specified with -o option.

And, the resulting file contains the machine instructions for the our program, with an undefined reference to printf.



```

1 | .section __TEXT,__text,regular,pure_instructions
2 | .macosx_version_min 10, 12
3 | .globl _main
4 | .p2align 4, 0x90
5 | _main:                                ## @main
6 | .cfi_startproc
7 | ## BB#0:
8 | pushq %rbp
9 | Ltmp0:
10 | .cfi_def_cfa_offset 16
11 | Ltmp1:
12 | .cfi_offset %rbp, -16
13 | movq %rsp, %rbp
14 | Ltmp2:
15 | .cfi_def_cfa_register %rbp
16 | subq $16, %rsp
17 | leaq L_.str(%rip), %rdi
18 | movl $0, -4(%rbp)
19 | movb $0, %al
20 | callq _printf
21 | xorl %ecx, %ecx
22 | movl %eax, -8(%rbp)                ## 4-byte Spill
23 | movl %ecx, %eax
24 | addq $16, %rsp

```

4. Linker

This is the final stage in compilation of our program.

This phase links object files to produce final executable file.

Linker links our .o file with other .o files.

An executable file requires many external resources (system functions, C run-time libraries etc.). Regarding our program we have noticed that it calls the printf function to print the '**Marvellous Infosystems by Piyush Khairnar**' message on console.

This function is contained in a separate pre compiled object file printf.o, which must somehow be merged with our Marvellous.o file.

The linker (ld) performs this task for we.

Eventually, the resulting file Marvellous is produced, which is an executable.

This is now ready to be loaded into memory and executed by the system.

The actual link command executed by linker is rather complicated.

```
[root@host ~]# ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 /usr/lib64/crt1.o /usr/lib64/crti.o /usr/lib64/crtn.o Marvellous.o /usr/lib/gcc/x86_64-redhat-linux/4.1.2/crtbegin.o -L /usr/lib/gcc/x86_64-redhat-linux/4.1.2/ -lgcc -lgcc_eh -lc -lgcc -lgcc_eh /usr/lib/gcc/x86_64-redhat-linux/4.1.2/crtend.o -o Marvellous
```

5. Loader :

As above executable file is in hard disk but for execution purpose it should be loaded in RAM.

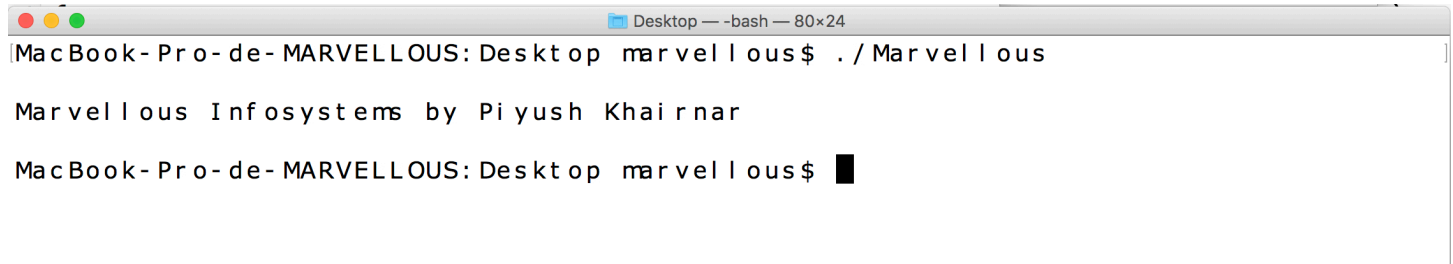
Loader is an Operating Dependent entity which loads our executable file from Hard disk into Ram.

Belo command invokes the loader.

[root@host ~]# ./Marvellous

Output:

Marvellous Infosystems by Piyush Khairnar



```
Desktop — -bash — 80x24
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ ./Marvellous

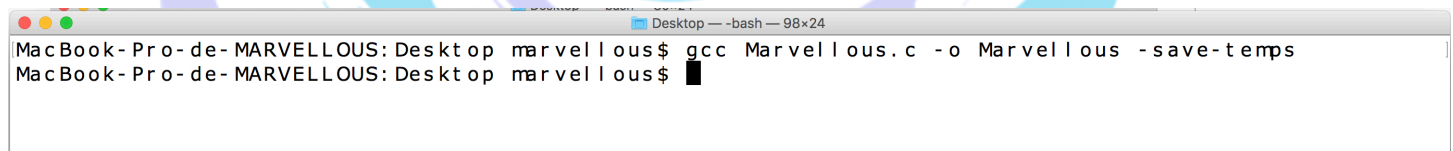
Marvellous Infosystems by Piyush Khairnar

MacBook-Pro-de-MARVELLOUS: Desktop marvellous$
```

For us, there is no need to type the complex ld command directly - the entire linking process is handled transparently by gcc when invoked, as follows.

[root@host ~]# gcc Marvellous.c -o Marvellous

To get the intermediate files which are created in build process by using -save-temps option.



```
Desktop — -bash — 98x24
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$ gcc Marvellous.c -o Marvellous -save-temps
MacBook-Pro-de-MARVELLOUS: Desktop marvellous$
```