C Refresher Module Assignment-0.2

Name - Rahul Khatoliya

Roll No. – 2019265

Compilation Process step by step (Command Line Options) :

1. **Preprocessing :**

   Command line argument : **$ gcc -E main.c -o main.i**

   Understanding the Meaning :

   Here **gcc** helps to invoke the GNU C compiler , **-E** enables us to stop and save the output file at preprocessed step (with an **.i extension** , meaning it's a preprocessed file) , Moreover **main.c** is our C program to be preprocessed and **-o** helps to name the resultant output file .

   Description of the Output File :

   1. It gets rid of all the comments in the source file of our program .
   2. it includes the code of the *header file(s)*, which is a file with extension .h which contains C function declarations and macro definitions .
   3. Removes all #define (which ever not used in the program) .

4. it replaces all of the *macros* (fragments of code which have been given a name) by their values .

2. **Compiling :**

Command line argument **: $ gcc -S main.i -o main.s**

Understanding the Meaning :

Here **gcc** helps to invoke the GNU C compiler , **-S** is used to stop and save the process at intermediate compilation step (with a **.s extension** , meaning it's an Assembly level instructions) , Moreover **main.c** is our C program to be and **-o** helps to name the resultant output file .

Description of the Output File :

1. File is purely based on Assembly language and contains Assembly commands .
2. It will produce different Language designs according to different system architecture . for eg:- (x86/64) etc .
3. Output file is the Resultant (Intermediate Representation) generated by compiler using the earlier preprocessed file .

3. **Assembly :**

   Command line argument : **$ gcc -c main.s -o main.o**

   Understanding the Meaning :

   Here **gcc** helps to invoke the GNU C compiler, **-c** is used to generate object file from the earlier formed assembly .s file and itself is a machine level instructions (with a **.o extension**) , Moreover **main.c** is our C program and **-o** helps to name the resultant output file . This is the next step of compiling, in this step the **main.s** (assembly level instructions) is used as the input file and converted to the **main.o** using the Assembler which is an object file (machine level instructions).

   Description of the Output File :

   1. The assembler takes the IR code and transforms it into object code, that is code in machine language (i.e. binary).
   2. Generated file is not human readable .
   3. It has .o extension .
   4. It is further linked , in the linking process .

4. **Linking :**

Command line argument : **$ gcc main.o -o main**

Understanding the Meaning :

Here **gcc** helps to invoke the GNU C compiler **, -o** instruct the C compiler that save the output file as main (Executable file with **.exe extension**) , Moreover **main.c** is our C program . Linking is the last stage of the compilation before producing the executable file. Linker is used to link the functions and its definitions. As linker know the address/location that where the the definitions of the function is written in the code. Also linker add some extra required code to the program like it add some standard code which is used to return the values in the program . Dynamic linking is done by the GCC by default.

Description of the Output File :

1. Can be execute using $ ./ main (executable)
2. All functions and their addresses are linked with their memories .
3. OS can easily load these types of files .
4. Some start and end code might be inserted .

Asm Command line options Step wise :

1. **Process of Conversion of .asm file to .o file :**

   Nasm : The Netwide Assembler is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit and 64-bit programs. NASM is considered to be one of the most popular assemblers for Linux.

   Command line Argument : **$ nasm -f elf64 add.asm -o add.o**

   Here **-f elf64** specifies the format to 64 bit based architecture , Also **add.asm** is the file where we have written the assembly language code , Moreover **-o** signifies the output file to be named as the file name written next to it , which is **add.o** in our case. This is the object file produced as the result of linking the assembly code individually .

2. **Linking Step for generating final Executable file :**

   Command line Argument : **$ gcc -no-pie main.o add.o -o a.out**

   Here **gcc** helps invoking the GNU C compiler , **-no-pie** ensures that produced executable is not position independent , also **main.o** and **add.o** are the resultant object file produced from the C program and Assembly program respectively , Atlast **-o** helps to save and name the final executable , i.e a.out in our case .

When an assembly source file is assembled by an assembler, and a C source file is compiled by a C compiler, those two object files can be linked together by a linker to form the final executable. The beauty of this approach is that the assembly files can be written using any syntax and assembler that the programmer is comfortable with. Also, if a change needs to be made in the assembly code, all of that code exists in a separate file, that the programmer can easily access. The only disadvanges of mixing assembly and C in this way are that a)both the assembler and the compiler need to be run, and b) those files need to be manually linked together by the programmer. These extra steps are comparatively easy, although it does mean that the programmer needs to learn the command-line syntax of the compiler, the assembler, and the linker.

Understanding the Process / Routine (Inside Main) :

Inorder to call a function present in the assembly file , we first have to Declare the extern prototyping inside our .C file which will tell the compiler initially the result type and parameter type for the same. In our case after taking user input of the two numbers , we invoke the function ‘Add’ which is defined in the .asm file and will eventually linked to Form the final executable file. Also , main() has its own _start label , so we don’t have to manually write that in the file again .

Understanding the Process / Routine (Assembly)

The Code inside the assembly in register friendly , and by default the arguments passed by the .C program are stored in rdi and rsi register , from where we get the final addition result . Here we can write comments starting from ‘ ; ’ inorder for good practices , different set of registers for e.g : rcx, rdx, rbx etc are used inorder to perform out arithmetic successfully. Moreover , the assembly code involves system calls such as write , exit etc , which have predefined id’s, addresses , and a specific number based on which it specifies STDOUT, STDIN etc.