# EXPERIMENT 2

## Creating, Compiling and Executing C/C++ programs using gcc/g++ Compilers and Make File

**OBJECTIVE:**

- Learn the use of g++ and gcc compilers to compile and execute C++ and C programs
- To get familiarized with the working of Make File for C/C++ programs

**BACKGROUND:**

**Compiling C/C++ program using g++ and gcc:**

**For C++:**

Command: g++ *source_files…* -o *output_file*

**For C:**

Command: gcc *source_files… -o output_file*

Source files need not be cpp or c files. They can be preprocessed files, assembly files, or object files.

The whole compilation file works in the following way:

Cpp/C file(s) → Preprocessed file(s) → Assembly File(s) Generation → Object file(s) Generation → Final Executable

Every c/cpp file has its own preprocessed file, assembly file, and object file.

1. For running only the preprocessor, we use -E option.

2. For running the compilation process till assembly file generation, we use –S option.

3. For running the compilation process till object file creation, we use –c option.

4. If no option is specified, the whole compilation process till the generation of executable will run.

A file generated using any option can be used to create the final executable. For example, let’s suppose that we have two source files: math.cpp and main.cpp, and we create object files:

*g++ main.cpp –c –o main.o*

*g++ math.cpp –c –o math.o*

The object files created using above two commands can be used to generate the final executable.

*g++ main.o math.o –o my_executable*

The file named “my_executable” is the final exe file. There is specific extension for executable files in Linux.

**Command Line Arguments:**

Command line arguments are a way to pass data to the program. Command line arguments are passed to the main function. Suppose we want to pass two integer numbers to main function of an executable program called a.out. On the terminal write the following line:

*./a.out 1 22*

./a.out is the usual method of running an executable via the terminal. Here 1 and 22 are the numbers that we have passed as command line argument to the program. These arguments are passed to the main function. In order for the main function to be able to accept the arguments, we have to change the signature of main function as follows:

int main(int argc, char *arg[]);

- ➔ argc is the counter. It tells how many arguments have been passed.
- ➔ arg is the character pointer to our arguments.

argc in this case will not be equal to 2, but it will be equal to 3. This is because the name ./a.out is also passed as command line argument. At index 0 of arg, we have ./a.out; at index 1, we have 1; and at index 2, we have 22. Here 1 and 22 are in the form of character string, we have to convert them to integers by using a function atoi. Suppose we want to add the passed numbers and print the sum on the screen:

cout<< atoi(arg[1]) + atoi(arg[2]);

**Compiler Process:**

- ➢ Compiler Stage: All C++ language code in the .cpp file is converted into a lower-level language called Assembly language; making .s files.
- ➢ Assembler Stage: The assembly language code made by the previous stage is then converted into object code which are fragments of code which the computer understands directly. An object code file ends with .o.

- Linker Stage: The final stage in compiling a program involves linking the object code to code libraries which contain certain "built-in" functions, such as cout. This stage produces an executable program, which is named a.out by default.

**Makefiles:**

- Provide a way for separate compilation.
- Describe the dependencies among the project files.
- The *make* utility.

**Using makefiles:**

Naming:

- *makefile* or *Makefile* are standard
- other name can be also used

Running make:

*make*

*make –f filename* – if the name of your file is not “makefile” or “Makefile”

**Sample makefile:**

Makefiles main element is called a *rule*:

```
target : dependencies

TAB  commands      #shell commands
```

**Example:**

my_prog : eval.o main.o

g++ -o my_prog eval.o main.o          (-o to specify executable file name)

eval.o : eval.c eval.h

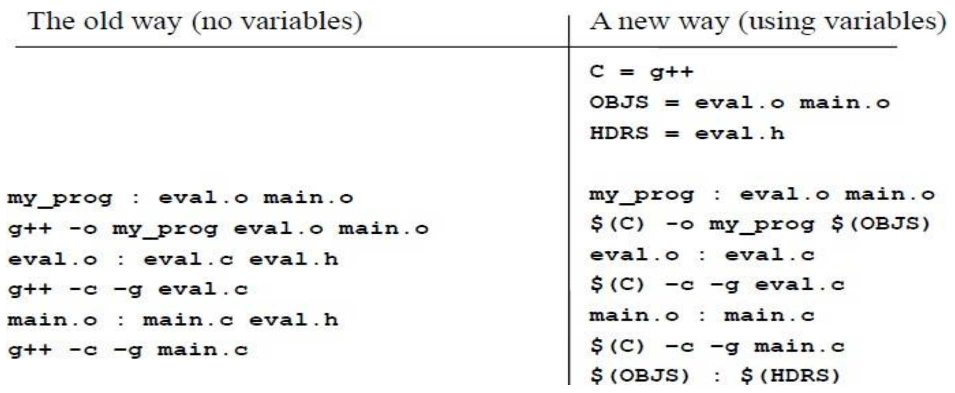
g++ -c eval.c                          (-c to compile only (no linking))

main.o : main.c eval.h

g++ -c main.c

**Variables:**

Figure 3.1 shows the use of variables while generating makefile.



| The old way (no variables) | A new way (using variables) |
|---|---|
| | `C = g++` |
| | `OBJS = eval.o main.o` |
| | `HDRS = eval.h` |
| | |
| `my_prog : eval.o main.o` | `my_prog : eval.o main.o` |
| `g++ -o my_prog eval.o main.o` | `$(C) -o my_prog $(OBJS)` |
| `eval.o : eval.c eval.h` | `eval.o : eval.c` |
| `g++ -c -g eval.c` | `$(C) -c -g eval.c` |
| `main.o : main.c eval.h` | `main.o : main.c` |
| `g++ -c -g main.c` | `$(C) -c -g main.c` |
| | `$(OBJS) : $(HDRS)` |

**Use of Variables for generating makefile**

**Figure 3.1**

**Automatic variables:**

Automatic variables are used to refer to specific part of rule components.

eval.o : eval.c eval.h

g++ -c eval.c

$@ - The name of the target of the rule (eval.o).

$< - The name of the first dependency (eval.c).

$^ - The names of all the dependencies (eval.c eval.h).

$? - The names of all dependencies that are newer than the target

make **options:**

-f filename - when the makefile name is not standard

-t - (touch) mark the targets as up to date

-q - (question) are the targets up to date, exits with 0 if true

-n - print the commands to execute but do not execute them

/ -t, -q, and -n, cannot be used together /

-s - silent mode

-k - keep going – compile all the prerequisites even if not able to link them !!

**Conditionals (directives):**

Possible conditionals are:

*if*  *ifeq*  *ifneq*  *ifdef*  *ifndef*

All of them should be closed with *endif*.

Complex conditionals may use *elif* and *else*.

**Example:**

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
libs=$(libs_for_gcc)            #no tabs at the beginning
else
libs=$(normal_libs)             #no tabs at the beginning
endif
```

**In-Lab Questions:**

**Question 1:** Write a C or C++ program that accepts a file name as command line argument and prints the file's contents on console. If the file does not exist, print some error on the screen.

**Question 2:** Write a C or C++ program that accepts a list of integers as command line arguments sorts the integers and print the sorted integers on the screen.

**Question 3:** Create the following classes in separate files (using .h and .cpp files)

Student, Teacher, Course.

A student has a list of courses that he is enrolled in.

A teacher has a list of courses that he is teaching.

A course has a list of students that are studying it, and a list of teachers that are teaching the course. Create some objects of all classes in main function and populate them with data.

Now compile all classes using makefile

**Post-Lab Questions:**

**Problem 1:** Write a C/C++ program that takes some integers as command line parameters, store them in an array and prints the sum and average of that array. Also note that you have to run the program for all possible error checks.

**Problem 2:** Write a C/C++ program that takes some integers in the form of series as command line parameters; store them in array than compute the missing element from that series and output that missing element to file.

**Problem 3:** Write a C/C++ program that reads file in which there are integers related to series and store them in array than compute the missing element from that series and output that missing element to file.

**Problem 4:** Create the following classes in separate files (using .h and .cpp files)

LetterCount, WordCount, LineCount.

LetterCount counts number of letters in a text file.

WordCount counts number of words in a text file. LineCount

counts number of lines in a text file.

Create some objects of all classes in main function and populate them with data. Now

compile all classes using makefile