Creation and Compilation of Simple Program in OS Linux

Work with gcc (g++) compiler

gcc is command line compiler. gcc is C language compiler. For C++ language g++ compiler is used.

Parameters for compiler call are following:

```
g++ [flags] source_code.cc [-o executable_file]
```

If ○ **executable_file** were not defined, compiler would create file **a.out.**

Flags could be following:

-g	**compile with debugging information**
-c	**don't ling, just compile to object files .o**
-Wall	**display all possible warning messages**

Examples:

```
g++ -g -Wall test.c++ -o test
gcc -c test.c
```

If compilation succeeded, in current directory would appear executable file (with execution permission: marked with green colour and * near file name).

To run this file from command line, enter

```
./lab1
```

You cannot use just **lab1**, because current directory by default isn't present in path which is used during program start according to security issues.

2.2 Work with gdb debugger

To debug program using gdb you should:

1. compile program with debugging information (key -g)

```
g++ -g lab1.cc -o lab1
```

2. run debugger with the program name for parameter

```
gdb lab1
```

After that debugging can be started. During debugging following commands can be used:

b (break) function-name or b line-number	**set breakpoint**
b min	**set breakpoint to the first operator of min() function**
b main	**set breakpoint to the first operator of main() function. To the beginning of program execution.**
b 10	**set breakpoint to the line number 10**
d (delete) the same parameters	**remove breakpoint**
r (run)	**start program for executing till first breakpoint under debugger control**
n (next)	**step to the next operator without entering functions (step over)**
s (step)	**step to the next operator with entering into functions (step into)**
c (continue)	**continue till next breakpoint**
l (list)	**display source code of program near current operator (10 lines)**
p variable_name	**show value of variable**
disp variable_name	**show variable on each program step**
undisp n	**hide variable number n**
bt (backtrack)	**display list of functions, which called current one**
bt full	**the same, including values of local variables**
i (info)	**display information about program which includes:**
i b	**list of breakpoints**
i lo	**values of all local variables**
q (quit)	**quit debugger**

Debugging session example:

```
gdb lab1
(gdb) b 1        // breakpoint in the beginning
(gdb) r          // till breakpoint
(gdb) disp k     // display k on each step
(gdb) n          // step
(gdb) l          // where are we?
(gdb) n          // step
(gdb) c          // execution till end
(gdb) q          // exit
```

2.3 Work with make command

make is used for program projects control. Program project is several source, header and similar files, needed for program product compilation.

After start, make reads special file called `makefile` (which name usually makefile). In makefile all connections between project files and commands, needed for compilation are described. Make defines from makefile which commands should be executed and runs them. Recompilation occurs only for those parts of project, which changed comparing to last make run.

If current directory (project directory) has makefile, then to recompile everything needed, you should just run command without parameters:

```
make
```

Makefile consists of rules collection. Syntax of rules is following:

```
target : files, from which target is dependent
command
```

Target can be:

1. file name, which must be created as a result of command execution (executable or object file);
2. name which is not name of file, which can be defined explicitly during make run for correspondent program execution.

Let's consider following example: program consists from two source files a1.cc and a2.cc, each of them includes header file my.h. Executed file lab1 can be compiled by compiling source files to a1.o and a2.o and linking them.

As target we use name of the executable file `lab1`, and rule for linking `lab1` would look like.

```
lab1 : a1.o a2.o
    g++ -o lab1 a1.o a2.o
```

Now every time when one of the object files would change, appropriate command from this rule would be run.

Then we need rules for a1.o and a2.o creation. Each .o file can be compiled from correspondent .cc file, and, except that, must be recompiled if `my.h` file is changed. For each file can be provided commands:

```
a1.o : a1.cc my.h
           g++ -c a1.cc -o a1.o
```

make allows to make rules easier using implicit rules. Implicit rules are stored within make and define dependencies by default. One of the implicit rules is that object file .o depends on file with the same name and extension .cc and compilation goes through command

```
g++ -c ******.cc -o ******.o
```

So our rules for object files become easier:

```
a1.o : my.h # everything else by default
```

So all makefile would be:

```
lab1 : a1.o a2.o
    g++ -o lab1 a1.o a2.o
a1.o : my.h
a2.o : my.h
```

As we can see, we have list of object file repeated two times. For not to type such lists for several times (and they can be long), we can use inner variables of make. After definition of such variable, it can be used in any place of makefile.

```
objects = a1.o a2.o          # variable definition
lab1 : $(objects)            # variable using
    g++ -o lab1 $(objects)
```

As an example of target, which is not file, let's examine target `clean`, which is used for cleaning directory from compilation results. To show, that this target is not file name (and to prevent conflicts with file named clean, if such appears in current directory), target must be described explicitly in such way:

```
.PHONY : clean
```

Rule for this target would be:

```
clean :
    rm -f lab1 $(objects)
```

Key -f for rm defines, that rm would not return mistake even if files are missing in the directory.

For execution such target, it must be defined explicitly as make parameter:

```
make clean
```

Final makefile would be:

```
objects = a1.o a2.o
.PHONY : clean
lab1 : $(objects)
    g++ -o lab1 $(objects)
a1.o : my.h
a2.o : my.h
clean :
    rm -f lab1 $(objects)
```

If we need to link project, consisting from several executable files (for example lab1a and lab1b) than we just can set common target, for example all, depending of targets lab1a and lab1b:

```
all : lab1a lab1b
lab1a : $(objects_a)
    g++ -o lab1a $(objects_a)
lab1b : $(objects_b)
    g++ -o lab1b $(objects_b)
```

If in current directory is file makefile then xemacs automatically on `compile` command would run make instead of compiler.

Task for the individual work:

Develop and debug program project (two or more source files, connected) in Linux environment using ***vim, emac or nano editor***, **gcc/g++** compiler, project manager **make** and debugger **gdb**. Program project must implement collection of interacting units, using standard input-output.

During work presentation, student must explain project development, and used tools of Linux environment.

Create a program to find the values of the expressions **A** and **B** for the given values of the source data **x**, **y**, **z**. The functions for finding **A** and **B** must be described in various files. You can get the input data (according your variant) from your teacher.

Useful links:

1) https://cets.seas.upenn.edu/answers/gcc.html
2) https://www.opennet.ru/docs/RUS/linux_parallel/node251.html
3) https://www.maketecheasier.com/debug-program-using-gdb-linux/
4) https://2buntu.com/articles/1035/how-to-write-a-makefile/
5) https://stackoverflow.com/questions/33280898/how-to-run-makefile-in-ubuntu
6) http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
7) https://habr.com/ru/post/155201/