CL2006 - Operating Systems Fall 2025
LAB # 2

Please note that all labs topics including pre-lab, in-lab and post-lab exercises are part of the theory and lab syllabus. These topics will be part of your Midterms and Final Exams of lab and theory.

Objectives:

1. Learn Compiler Commands and process.
2. Learn how to use command line arguments
3. Learn to create and use Makefile
4. Learn compiling linux kernel form source code

Lab Tasks:

1. Compiling C/C++ Programs using g++ and gcc:
    a. Commands for compiling C++ (g++) and C (gcc) programs.
    b. Overview of the compilation process: Options for running only the preprocessor, compilation process assembly file generation till object file creation.
    c. Example of creating object files and generating the final executable.

2. Command Line Arguments:
    a. Passing data to the program through command line arguments.
    b. Modifying the main function to accept command line arguments (int main(int argc, char *argv[])).
    c. Handling command line arguments and converting them to integers.

3. Compiler Process:
    a. Stages in the compiler process: Compiler Stage, Assembler Stage, Linker Stage.
    b. Compilation from C++ language code to Assembly language to object code.
    c. Linking object code to code libraries to produce the executable program.

4. Makefiles:
    a. Purpose of Makefiles: Separate compilation, describe project file dependencies.
    b. Overview of the make utility.
    c. Introduction to rules in makefiles, Example makefile structure with a rule.
    d. Rule elements: Target, Dependencies, and Commands.
    e. Automatic variables ($@, $<, $^, $?) and their role in rules.
    f. Common Example

5. Compiling Linux Kernel from sources

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 executablewill 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 objectfiles:

```
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-levellanguage called Assembly language; making .s files.
- Assembler Stage: The assembly language code made by the previous stage is thenconverted 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 codeto 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.

Naming:

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

Running make:

```
$make
or
$make -f filename
```

Where filename is the name of your file is not "makefile" or "Makefile"

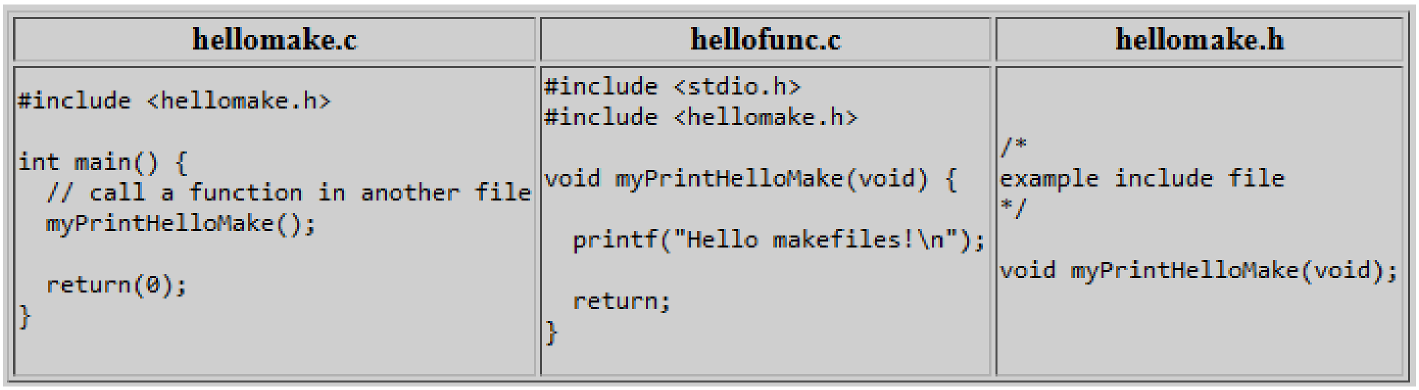
Automatic variables:

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
```

Makefile Example (Taken from https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/)

Let's start off with the following three files, hellomake.c, hellofunc.c, and hellomake.h, which would represent a typical main program, some functional code in a separate file, and an include file, respectively.



hellomake.c

```
#include <hellomake.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

hellofunc.c

```
#include <stdio.h>
#include <hellomake.h>

void myPrintHelloMake(void) {

  printf("Hello makefiles!\n");

  return;
}
```

hellomake.h

```
/*
example include file
*/

void myPrintHelloMake(void);
```

Normally, you would compile this collection of code by executing the following command:

```
gcc -o hellomake hellomake.c
hellofunc.c -I
```

This compiles the two .c files and names the executable hellomake. The -I is included so that gccwill look in the current directory (.) for the include file hellomake.h.

Without a makefile, the typical approach to the test/modify/debug cycle is to use the up arrow in a terminal to go back to your last compile command so you don't have to type it each time, especially once you've added a few more .c files to the mix.

Unfortunately, this approach to compilation has two downfalls. First, if you lose the compile command or switch computers then you have to retype it from scratch, which is inefficient at best. Second, if you are only making changes to one .c file, recompiling all of them every time is also time-consuming and inefficient. So, it's time to see what we can do with a makefile.

The simplest makefile you could create would look something like:

Makefile 1

```
hellomake: hellomake.c hellofunc.c
      gcc -o hellomake hellomake.c
      hellofunc.c -I
```

If you put this rule into a file called Makefile or makefile and then type make on the command line. It will execute the compile command as you have written it in the makefile. Note that make with no arguments executes the first rule in the file.

Furthermore, by putting the list of files on which the command depends on the first line after the : make knows that the rule hellomake needs to be executed if any of those files change. Immediately, you have solved problem #1 and can avoid using the up arrow repeatedly, looking for your last compile command. However, the system is still not being efficient in terms of compiling only the latest changes.

One very important thing to note is that there is a tab before the gcc command in the makefile. There mustbe a tab at the beginning of any command, and make will not be happy if it's not there.

In order to be a bit more efficient, let's try the following:

Makefile 2

```
CC=gcc
CFLAGS=I

hellomake: hellomake.o hellofunc.o
      $(CC) -o hellomake hellomake.o
hellofunc.o
```

So now we've defined some constants CC and CFLAGS. It turns out these are special constants that communicate to make how we want to compile the files hellomake.c and hellofunc.c. In particular, the macro CC is the C compiler to use, and CFLAGS is the list of flags to pass to the compilation command. By putting the object files--hellomake.o and hellofunc.o--in the dependency list and in the rule, make knows it must first compile the .c versions individually, and then build the executable hellomake.

Using this form of makefile is sufficient for most small-scale projects. However, there is one missing: dependency on the include files. If you were to make a change to hellomake.h, for example, make would not recompile the .c files, even though they needed to be. In order to fix this, we need to tell makethat all .c files depend on certain .h files. We can do this by writing a simple rule and adding it to the makefile.

Makefile 3

```
CC=gcc
CFLAGS=I.
DEPS = hellomake.h

%.o: %.c $(DEPS)
   $(CC) -c -o $@ $< $(CFLAGS)

hellomake: hellomake.o hellofunc.o
   $(CC) -o hellomake hellomake.o
hellofunc.o
```

This addition first creates the macro DEPS, which isthe set of .h files on which the .c files depend. Then we define a rule that applies to all files ending in the .o suffix. The rule says that the .o file depends uponthe .c version of the file and the .h files included in the DEPS macro. The rule then says that to generatethe .o file, make needs to compile the .c file using the compiler defined in the CC macro. The -c flag says to generate the object file, the -o $@ says to put the output of the compilation in the file named on the left side of the :, the $< is the first item in thedependencies list, and the CFLAGS macro is defined as above.

As a final simplification, let's use the special macros $@ and $^, which are the left and right sides of the:, respectively, to make the overall compilation rulemore general. In the example below, all of the include files should be listed as part of the macro DEPS, and all of the object files should be listed aspart of the macro OBJ.

Makefile 4

```
CC=gcc
CFLAGS=-I
DEPS = hellomake.h
OBJ = hellomake.o hellofunc.o

%.o: %.c $(DEPS)
   $(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
   $(CC) -o $@ $^ $(CFLAGS)
```

Let's break down each line:

CC=gcc: This line defines a variable `CC` and sets its value to `gcc`, indicating that the GNU C Compiler (`gcc`) will be used for compiling the source files.
CFLAGS=-I: This line defines a variable `CFLAGS` and sets its value to `-I`. `-I` is a compiler option used to specify additional directories to search for header files. However, it seems incomplete here as no directory is specified after `-I`.
DEPS = hellomake.h: This line defines a variable `DEPS` and sets its value to `hellomake.h`. `DEPS` typically stands for dependencies, and in this case, it specifies the header file(s) that the source files depend on.
OBJ = hellomake.o hellofunc.o:
This line defines a variable `OBJ` and sets its value to `hellomake.o hellofunc.o`. `OBJ` typically stands for object files, and it lists the object files that need to be generated by compiling the corresponding source files.
`%.o: %.c $(DEPS)`:
This is a pattern rule that specifies how to generate an object file (`%.o`) from a corresponding source file (`%.c`) along with its dependencies (`$(DEPS)`). When invoked, this rule tells `make` that for any `.o` file it needs to create, it can find the corresponding `.c` file and its dependencies in order to compile them.
`$(CC) -c -o $@ $< $(CFLAGS)`:

- This line is the recipe associated with the pattern rule defined above. It specifies the commands to compile a `.c` file into an object file.
- `$(CC)`: Evaluates to `gcc`, indicating the compiler to be used.
- `-c`: Instructs the compiler to generate an object file.
- `-o $@`: Specifies the output file name, where `$@` is a built-in variable representing the target of the rule, which in this case is the object file being generated.
- `$<`: Represents the first prerequisite of the rule, which is the source file (`%.c`).
- `$(CFLAGS)`: Contains additional compiler flags, though it's incomplete in this Makefile.

`hellomake: $(OBJ)`:
This line defines a target `hellomake` that depends on the object files listed in `$(OBJ)`. It specifies that the executable `hellomake` should be built using these object files.

`$(CC) -o $@ $^ $(CFLAGS)`:

- This line is the recipe associated with the target `hellomake`. It specifies the commands to link the object files into an executable.
- `$(CC)`: Evaluates to `gcc`, indicating the linker to be used.
- `-o $@`: Specifies the output file name, where `$@` is a built-in variable representing the target of the rule, which in this case is the executable being generated (`hellomake`).
- `$^`: Represents all the prerequisites of the rule (`$(OBJ)`), i.e., the object files.
- `$(CFLAGS)`: Contains additional linker flags, though it's incomplete in this Makefile.

Recompiling the Linux kernel

The general steps to recompile the Ubuntu kernel from sources with parameter changes:

```
$uname -r
```

Installing dependencies

- Install necessary build tools and dependencies. If you have not built a kernel on your system before, there are some packages needed before you can successfully build. You can get these installed with:

```
$sudo apt install build-essential
libncurses-dev bison flex libssl-dev
libelf-dev fakeroot

$sudo apt install dwarves
```

The above are for Ubuntu 20.04.

Download Kernel Sources

Next we need to need to download the kernel source from the offical https://www.kernel.org/ website.

Choose the latest longterm release and copy the link - of the tarball hyperlink. Then use this link to download and unpack the kernel source to your Ubuntu machine:

```
$wget
https://cdn.kernel.org/pub/linux/kerne
l/v6.x/linux-6.1.73.tar.xz
$tar -xf linux-6.1.73.tar.gz
$ cd linux-6.1.73
```

Configure Kernel:

- We start by coping existing configure as new configuration.

```
$cp -v /boot/config-$(uname -r).config
```

- Adjust the configuration using a menu-based interface:

```
$make menuconfig
```

- Use the arrows to make a selection or choose Helpto learn more about the options. When you finish making the changes, select Save, and then exit the menu.

- Disable the conflicting security certificates byexecuting the following commands below:

```
$ scripts/config --disable
SYSTEM_TRUSTED_KEYS
$ scripts/config --disable
SYSTEM_REVOCATION_KEYS
$ scripts/config --set-str
CONFIG_SYSTEM_TRUSTED_KEYS ""
$ scripts/config --set-str
CONFIG_SYSTEM_REVOCATION_KEYS ""
```

Compile Kernel:

- Build the kernel and modules:

```
$make -j$(nproc)
```

- The `-j$(nproc)` option enables parallel compilation, utilizing the number of available processor cores. Press Enter repeatedly to confirm thedefault options for the generation of new certificates.

Install Modules:

- Install the kernel modules:

```
$sudo make modules_install
```

Install New Kernel:

Install the new kernel:

```
$sudo make install
```

- This will update the bootloader configuration (GRUB) and copy the new kernel image to `/boot`.

Update GRUB Configuration:

- Ensure that GRUB is aware of the new kernel. Run:

```
$sudo update-grub
```

Reboot:

- Reboot the system to load the new kernel:

```
$sudo reboot
```

Verify:

- After rebooting, check the kernel version:

$uname -r

References:

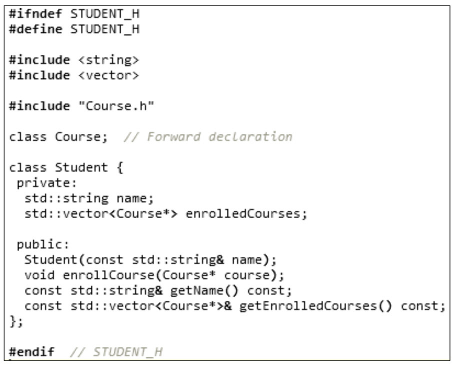
- https://phoenixnap.com/kb/build-linux-kernel
- https://davidaugustat.com/linux/how-to-compile-linux-kernel-on-ubuntu

In-Lab Questions:

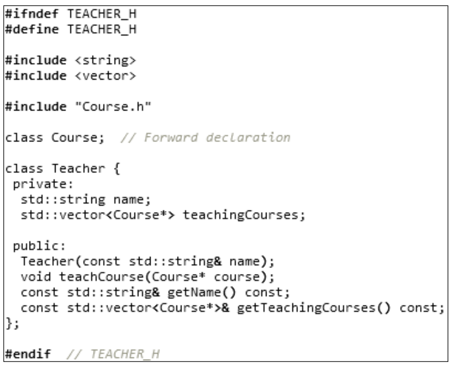
1) Create the following classes in separate files (using .h and .cpp files) Student, Teacher, Course.
   a. A student has a list of courses that he is enrolled in. A teacher has a list of courses that he is teaching.
   b. A course has a list of students that are studying it, and a list of teachers that are teaching thecourse. Create some objects of all classes in main function and populate them with data.

Now compile all classes using makefile.

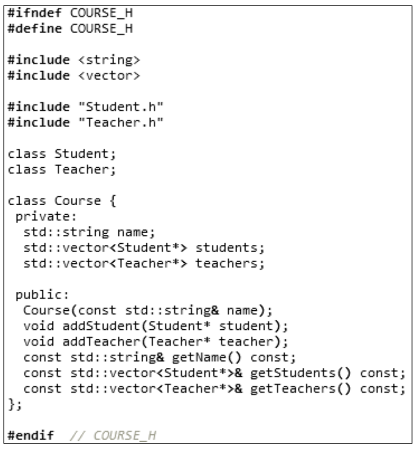
Template code generated by ChatGPT



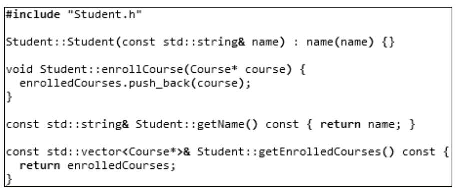
```cpp
#ifndef STUDENT_H
#define STUDENT_H

#include <string>
#include <vector>

#include "Course.h"

class Course;  // Forward declaration

class Student {
 private:
  std::string name;
  std::vector<Course*> enrolledCourses;

 public:
  Student(const std::string& name);
  void enrollCourse(Course* course);
  const std::string& getName() const;
  const std::vector<Course*>& getEnrolledCourses() const;
};

#endif  // STUDENT_H
```



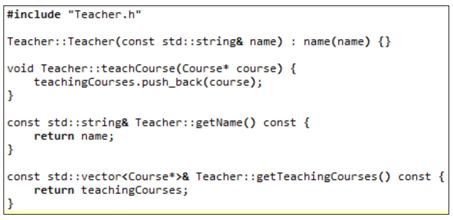
```cpp
#ifndef TEACHER_H
#define TEACHER_H

#include <string>
#include <vector>

#include "Course.h"

class Course;  // Forward declaration

class Teacher {
 private:
  std::string name;
  std::vector<Course*> teachingCourses;

 public:
  Teacher(const std::string& name);
  void teachCourse(Course* course);
  const std::string& getName() const;
  const std::vector<Course*>& getTeachingCourses() const;
};

#endif  // TEACHER_H
```



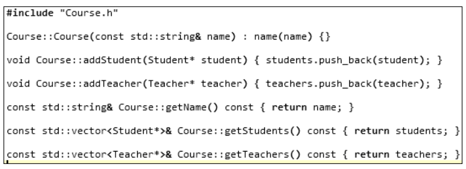
```cpp
#ifndef COURSE_H
#define COURSE_H

#include <string>
#include <vector>

#include "Student.h"
#include "Teacher.h"

class Student;
class Teacher;

class Course {
 private:
  std::string name;
  std::vector<Student*> students;
  std::vector<Teacher*> teachers;

 public:
  Course(const std::string& name);
  void addStudent(Student* student);
  void addTeacher(Teacher* teacher);
  const std::string& getName() const;
  const std::vector<Student*>& getStudents() const;
  const std::vector<Teacher*>& getTeachers() const;
};

#endif  // COURSE_H
```



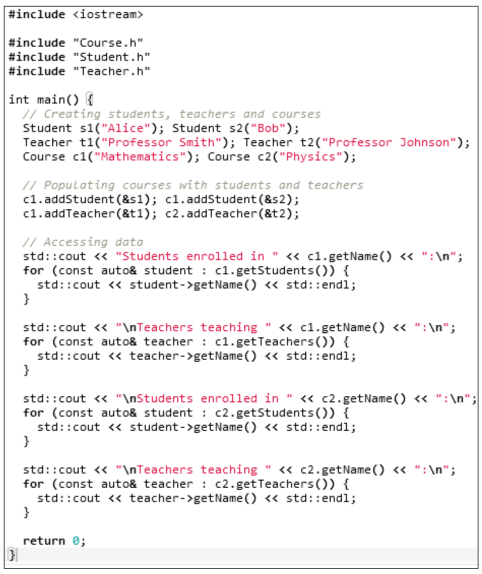
```cpp
#include "Student.h"

Student::Student(const std::string& name) : name(name) {}

void Student::enrollCourse(Course* course) {
  enrolledCourses.push_back(course);
}

const std::string& Student::getName() const { return name; }

const std::vector<Course*>& Student::getEnrolledCourses() const {
  return enrolledCourses;
}
```



```cpp
#include "Teacher.h"

Teacher::Teacher(const std::string& name) : name(name) {}

void Teacher::teachCourse(Course* course) {
    teachingCourses.push_back(course);
}

const std::string& Teacher::getName() const {
    return name;
}

const std::vector<Course*>& Teacher::getTeachingCourses() const {
    return teachingCourses;
}
```

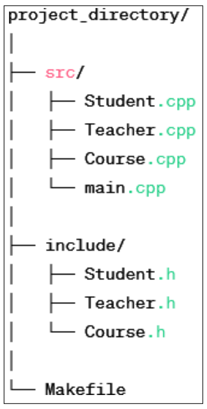


```cpp
#include "Course.h"

Course::Course(const std::string& name) : name(name) {}

void Course::addStudent(Student* student) { students.push_back(student); }

void Course::addTeacher(Teacher* teacher) { teachers.push_back(teacher); }

const std::string& Course::getName() const { return name; }

const std::vector<Student*>& Course::getStudents() const { return students; }

const std::vector<Teacher*>& Course::getTeachers() const { return teachers; }
```



```cpp
#include <iostream>

#include "Course.h"
#include "Student.h"
#include "Teacher.h"

int main() {
  // Creating students, teachers and courses
  Student s1("Alice"); Student s2("Bob");
  Teacher t1("Professor Smith"); Teacher t2("Professor Johnson");
  Course c1("Mathematics"); Course c2("Physics");

  // Populating courses with students and teachers
  c1.addStudent(&s1); c1.addStudent(&s2);
  c1.addTeacher(&t1); c2.addTeacher(&t2);

  // Accessing data
  std::cout << "Students enrolled in " << c1.getName() << ":\n";
  for (const auto& student : c1.getStudents()) {
    std::cout << student->getName() << std::endl;
  }

  std::cout << "\nTeachers teaching " << c1.getName() << ":\n";
  for (const auto& teacher : c1.getTeachers()) {
    std::cout << teacher->getName() << std::endl;
  }

  std::cout << "\nStudents enrolled in " << c2.getName() << ":\n";
  for (const auto& student : c2.getStudents()) {
    std::cout << student->getName() << std::endl;
  }

  std::cout << "\nTeachers teaching " << c2.getName() << ":\n";
  for (const auto& teacher : c2.getTeachers()) {
    std::cout << teacher->getName() << std::endl;
  }

  return 0;
}
```

Put your files in the following directory structure.



```
project_directory/
│
├─ src/
│   ├─ Student.cpp
│   ├─ Teacher.cpp
│   ├─ Course.cpp
│   └─ main.cpp
│
├─ include/
│   ├─ Student.h
│   ├─ Teacher.h
│   └─ Course.h
│
└─ Makefile
```

```makefile
CC = g++
CFLAGS = -Wall -Wextra -std=c++11
SRC_DIR = src
OBJ_DIR = obj
BIN_DIR = bin
TARGET = main

SRC_FILES := $(wildcard $(SRC_DIR)/*.cpp)
OBJ_FILES := $(patsubst $(SRC_DIR)/%.cpp,$(OBJ_DIR)/%.o,$(SRC_FILES))

$(BIN_DIR)/$(TARGET): $(OBJ_FILES)
	@mkdir -p $(BIN_DIR)
	$(CC) $(CFLAGS) $^ -o $@

$(OBJ_DIR)/%.o: $(SRC_DIR)/%.cpp
	@mkdir -p $(OBJ_DIR)
	$(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean

clean:
	rm -rf $(OBJ_DIR) $(BIN_DIR)
```

Explanation of the Makefile:
  CC: Compiler to use (in this case, g++).
  CFLAGS: Compiler flags (including -Wall for all warnings, -Wextra for extra warnings, and -std=c++11 to enable C++11 features).
  SRC_DIR: Directory containing the source files (.cpp files).
  OBJ_DIR: Directory where object files will be stored.
  BIN_DIR: Directory where the final executable will be stored.
  TARGET: Name of the final executable.

The Makefile uses pattern rules to automatically compile each .cpp file into an object file and then link all the object files together to create the final executable. It also includes a clean target to remove all generated object files and the executable. To use this Makefile, create a directory structure as shown above.

2) Write a C or C++ program that accepts a file name as command line argument and prints the file'scontents on console. If the file does not exist, print some error on the screen.

3) Write a C or C++ program that accepts a list of integers as command line arguments sorts

   theintegers and print the sortedintegers on the screen.