WORKSHEET, WEEK 04: BUILD AND DEBUG ON LINUX

4.1 Building with Makefile

```
CC=g++
CFLAGS=-std=c++17

all: myprogram

myprogram: myprogram.cpp
	$(CC) $(CFLAGS) -o myprogram myprogram.cpp

.PHONY: clean
clean:
	rm -f myprogram
```

Utility *make* was invented in 1976. Analyzes dependencies (which file was updated when), target names are typically same as file names. In the above example

- In a *Makefile* each line should be either non-indented (a variable definition or a target) or be indented by exactly one *TAB* character. Preceding that *TAB* by an (invisible) whitespace would break it.
- *.PHONY clean* declaration says that the target name has nothing to do with any filename. (If we do not use it, then *clean* would not run provided there is a file *clean*.

Multiple C++ Sources:

```
CC=g++
CFLAGS=-std=c++11

SRCDIR=.
SRC=$(wildcard $(SRCDIR)/*.cpp)
OBJ=$(SRC:.cpp=.o)
EXEC=$(SRCDIR)/myprogram

all: $(EXEC)

$(EXEC): $(OBJ)
	$(CC) $(CFLAGS) -o $@ $^

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

.PHONY: clean
clean:
	rm -f $(SRCDIR)/*.o $(EXEC)
```

Use Makefile to Run Testfiles:

```
# Define variables
PROGRAM = myprogram
INPUT_FILES = $(wildcard test*.txt)
OUTPUT_FILES = $(patsubst test%.txt,output.test%.txt,$(INPUT_FILES))

.PHONY: test
test: $(OUTPUT_FILES)

output.test%.txt: $(PROGRAM) test%.txt
        ./$(PROGRAM) < $< > $@ || true

# Clean up target
clean:
        rm -f $(OUTPUT_FILES)
```

The snippet || *true* prevents stopping the `make` task, if some of the program execution returns non-zero return code or crashes.

Note: Currently C++ developers use mostly *CMake* – a "meta-build" tool that creates makefiles or other build artefacts from a description of the project and its dependencies described in the `CMakeLists.txt` file. CMake can participate in other build chains – such as Gradle build tasks, where Android app written in Java or Kotlin needs some native code in C++.

In other cases custom Bash shell, Python or Groovy can be used instead of Makefile or `CMakeLists.txt`. Build tools are often part of larger build infrastructures using `crontab` time-scheduling, Continuous Integration tools such as Jenkins,

---

CMakeLists.txt Example:

```
cmake_minimum_required(VERSION 3.10)
project(myprogram)
add_executable(myprogram myprogram.cpp)
```

Here is how to use it:

```
cmake .
make
```

4.2 Unit-tests with Catch2

To run a project with Catch2 tests we need two different build goals in *Makefile*. One of them is builds the executable you can run; another one builds the test harness (executable that can be used to run the unit tests).

4.3 Debugging with gdb

`gdb myprogram`: Start gdb and load the myprogram executable.

`run`: Start the program.

`break <line_number>`: Set a breakpoint at the specified line number.

`info break`: Show all defined breakpoints.

`delete <breakpoint_number>`: Delete the specified breakpoint.

`next`: Step over the current line.

`step`: Step into the function called on the current line.

`finish`: Continue execution until the current function returns.

`backtrace`: Show the current call stack.

`list`: Show the current source code around the current line.

`print <variable_name>`: Print the value of the specified variable.

`display <variable_name>`: Display the value of the specified variable after each step.

`watch <variable_name>`: Set a watchpoint on the specified variable.

`info registers`: Show the current state of all CPU registers.

`x/<length><format><address>`: Examine memory at the specified address, with the specified format and length.

`layout src`: Display the source code and assembly code in separate windows.

`layout regs`: Display the CPU registers and the source code in separate windows.

`layout split`: Display the source code and the program output in separate windows.

`layout next`: Switch to the next layout.

4.4 The Lifecycle of Data Structures

- Constructors for empty data structures and initializer lists.
- Copy constructors during assignments or function calls.
- When are the destructors called.
- When is a proper time to release memory?

4.5 Valgrind

Memory leak detection: Valgrind can detect memory leaks by identifying when memory is allocated but not freed. Use `--leak-check` option.

```
valgrind --leak-check=yes ./myprogram
# (or write directly to a file)
valgrind --leak-check=yes --log-file=leak_report.txt ./myprogram
```

Memory error detection: Valgrind can detect memory errors: accessing memory that has already been freed, accessing uninitialized memory, and writing to read-only memory. Use `--tool=memcheck` option.

```
valgrind --tool=memcheck ./myprogram
```

Performance profiling: Valgrind can help identify performance bottlenecks by profiling CPU usage, memory usage, and other metrics. Use `--tool=callgrind` option

```
valgrind --tool=callgrind ./myprogram
```

This generates a file called callgrind.out.<pid>. Can use a tool like `kcachegrind` to visualize the profiling data.

4.6 Problems

1. Answer some questions about `Makefile` builds:

   (A) What is a dependency in a Makefile, and how is it specified?

   (B) How does Makefile determine whether a target needs to be rebuilt or not?

   (C) What is the purpose of the `.PHONY` target in a Makefile, and when should it be used?

   (D) What is a pattern rule in a Makefile, and how is it used?

   (E) What is the meaning of variables `$@` and `$<`?

   (F) How can you specify conditional dependencies in a Makefile, and why would you want to do this? (stuff like `ifeq`, `else`, `endif`)