




Course

-  ELEC-A7151

 Course materials

 Your points

Code

-  Code Vault

Course Pages

-  MyCourses

 Teams Channel



This course has already ended.

The latest instance of the course can be found at: [Object oriented programming with C++: 2023 Autumn](#)

« Software development tools

Course materials

2 Software libraries »

ELEC-A7151 / Software development tools / 1 Built automation tools

# Built automation tools

This page describes the old basic **Make** tool, and **CMake**, which is considered as the modern way of building projects.

(Traditional) make

[Make](#) is a build automation tool especially used in unix systems. It is used to build programs that consist of many different source code files. It is meant to help programmer build and link a program so the programmer does not need to compile a large project one file at a time. You can imagine makefile as an automated robot that writes the

```
g++ -std=c++11 -Wall -g -o main main.cpp
```

automatically for you when you just run **make** from the command line.

Make needs a rule file, *Makefile* in order to work. In this makefile the programmer specifies the files that need to be compiled, the dependencies of the files and build instructions. Both executables and libraries can be built with make. They are usually built by building smaller object files first and then linking those object files to form an executable or a library file. The programmer can make things even easier using make macros in this makefile.

You can access an [introductory Makefile tutorial on this page](#).

Below is a commented makefile from this course. This makefile belongs to the exercise ToString. It is modifiable, and the code itself will pass the tests. You can try to edit it and see if it still compiles. The **run-test** target is the one being used. This makefile is fairly complex to support the external testing library, but it features many of the interesting features of makefiles that can be used.

```
1MAIN_FILES = main.cpp main.o
2SOURCES=main.cpp
3HEADERS=to_string.hpp
4OBJECTS=$(SOURCES:.cpp=o)
5EXECNAME=main
6
7TESTNAME = test
8TEST_SOURCE = test_source.cpp
9
10SOURCES := $(filter-out $(MAIN_FILES), $(SOURCES))
11OBJECTS := $(filter-out $(MAIN_FILES), $(OBJECTS))
12HEADERS := $(filter-out $(MAIN_FILES), $(HEADERS))
13
14# The testing library sources
15GCHECK_DIR = ../../gcheck
16
17include $(GCHECK_DIR)/vars.make
18GCHECK_OBJECTS:=$(foreach OBJECT, $(GCHECK_OBJECTS), $(GCHECK_DIR)/$(OBJECT))
19
20# Where to find user code.
21USER_DIR = ../src
22TEST_DIR = .
23
24HEADERS := $(foreach HEADER, $(HEADERS), $(USER_DIR)/$(HEADER))
25
26# Flags passed to the preprocessor.
27# Set gcheck's header directory as a system directory, such that
28# the compiler doesn't generate warnings in gcheck headers.
29CPPFLAGS += -c -isystem $(GCHECK_DIR)
30
31# Flags passed to the C++ compiler.
32CXXFLAGS += -std=c++17 -g -Wall -Wextra -Wno-missing-field-initializers
33
34# All gcheck headers. Usually you shouldn't change this definition.
35GCHECK_HEADERS = $(GCHECK_DIR)/gcheck.h $(GCHECK_DIR)/argument.h
36
37ifeq ($(OS),Windows_NT)
38  RM=del /f /q
39  TEST:=$(TESTNAME).exe
40else
41  RM=rm -f
42  TEST:=$(TESTNAME)
43endif
44
45# House-keeping build targets.
46
47.PHONY: all clean clean-all run run-test valgrind-run get-results get-points get-report run-target
48
49all : $(TEST)
50
51clean :
52 $(RM) $(TESTNAME) $(TESTNAME).exe *.o report.json output.html valgrind_out.txt $(CLEAN_FILES)
53
54clean-all : clean
55 $(MAKE) -C $(GCHECK_DIR) clean
56
57run: $(TEST)
58 ./${TEST}
59
60# This is target being run when you press the run button,
61# you can try changing the contents here.
62run-test: clean $(TEST)
63 ./${TEST} --json report.json
64
65valgrind-run: clean $(TEST)
66 valgrind --track-origins=yes --leak-check=full ./${TEST}
67
68# $(CXX) $(CPPFLAGS) -I$(GTEST_DIR) $(CXXFLAGS) -c $(GTEST_DIR)/src/gtest-all.cc
69
70# Builds a sample test.
71
72$(TEST).o : $(TEST_DIR)/$(TEST_SOURCE) $(HEADERS)
73 $(CXX) $(CPPFLAGS) $(CXXFLAGS) $(TEST_DIR)/$(TEST_SOURCE) -o $@
74
75$(OBJECTS): %.o : $(USER_DIR)/%.cpp $(HEADERS)
76 $(CXX) $(CPPFLAGS) $(CXXFLAGS) $< -o $@
77
78$(GCHECK_OBJECTS): %.o : %.cpp $(GCHECK_HEADERS)
79 $(CXX) $(CPPFLAGS) $(CXXFLAGS) $< -o $@
80
81$(COBJECTS): %.o : $(USER_DIR)/%.c $(HEADERS)
82 $(CC) $(CPPFLAGS) $(CFLAGS) $< -o $@
83
84$(TEST): $(OBJECTS) $(COBJECTS) $(TEST).o $(GCHECK_OBJECTS)
85 $(CXX) $(LDFLAGS) $(LOADLIBES) $(LDLIBS) $(OBJECTS) $(COBJECTS) $(TEST).o $(GCHECK_OBJECTS) -o $@
```

CMake

[CMake](#) is a more advanced build tool that generates the makefile. CMake also works in Windows and OS X environments (it can be integrated with Visual Studio or Xcode). With CMake it is easy to make so called “out-of-place builds” which means that the generated object and executable files are not in the same directory as the source code. This helps keeping the source directory tidy.

CMake configuration is defined in file titled “CMakeLists.txt”. It defines the high-level configuration about source file locations, target files, and library dependencies. Additionally CMake scans the system build environment, and generates the actual Makefiles based on this information that will build the running program. Benefit of CMake is that it makes it easier to build the source code in different environments.

You can read more about CMake from here:

- [Hello world introductions to CMake](#)
- [CMake tutorial](#)
- [CMake community wiki](#)
- [How to use CMake with SFML](#) (does not work with SFML 2.5)

Below is a simple CMakeLists.txt that builds a project using the SFML and Box2D. It assumes that source files are under *src* directory. SFML inclusion relies on an additional file, “FindSFML.cmake” to be located in *cmake\_modules* directory. See additional instructions from above tutorial.

```
1 cmake_minimum_required (VERSION 2.6)
2 project (MyGame)
3 include_directories("src")
4
5 # Set the name of the executable to be produced
6 set(EXECUTABLE_NAME MyGame)
7
8 # Add all .cpp - files under src/ directory
9 file(GLOB SOURCES src/*.cpp)
10 add_executable(${EXECUTABLE_NAME} ${SOURCES})
11 set(CMAKE_BUILD_TYPE Debug)
12
13 # Find Box2D library
14 find_package(Box2D REQUIRED)
15 target_link_libraries(${EXECUTABLE_NAME} Box2D)
16
17 # Detect and add SFML
18 set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/cmake_modules" ${CMAKE_MODULE_PATH})
19 #Find any version 2.X of SFML
20 #See the FindSFML.cmake file for additional details and instructions
21 find_package(SFML 2 REQUIRED network audio graphics window system)
22 if(SFML_FOUND)
23  include_directories(${SFML_INCLUDE_DIR})
24  target_link_libraries(${EXECUTABLE_NAME} ${SFML_LIBRARIES} ${SFML_DEPENDENCIES})
25 endif()
26
27 # For installing targets (not mandatory)
28 install (TARGETS ${EXECUTABLE_NAME} DESTINATION bin)
```

« Software development tools

Course materials

2 Software libraries »