Exercise session 07

Introduction to GNU Make. Libraries: building and use.

Advanced Programming - SISSA, UniTS, 2023-2024

Pasquale Claudio Africa

16 Nov 2023

Introduction to GNU Make

Introduction

Prerequisites

Ensure the `make` program is installed by checking `make --version`. If not installed, use package managers such as `apt` on Unix or `Homebrew` on macOS. Additionally, have a C++ compiler like `g++` or `clang++` ready for compilation.

Getting started

Let's start with a simple C++ program consisting of three files:

- `math.hpp`
- `math.cpp`
- `main.cpp`

Manual compilation

```
g++ -c -I. -std=c++17 -Wall -Wpedantic -Werror main.cpp math.cpp
g++ -Wall -Wpedantic -Werror main.o math.o -o main

# Alternatively, in a single line:
g++ -I. -Wall -Wpedantic -Werror main.cpp math.cpp -o main
```

This process involves creating object files and linking them to generate the executable. Now, let's simplify this with a Makefile.

Definitions

- In a Makefile, a **target** represents the desired output or action. It can be an executable, an object file, or a specific action like "clean."
- **Prerequisites** are files or conditions that a target depends on. If any of the prerequisites have been modified more recently than the target, or if the target does not exist, the associated recipe is executed.
- A **recipe** is a set of shell commands that are executed to build or update the target. Recipes follow the prerequisites and are indented with a **<TAB> character**. Each line in the recipe typically represents a separate command.

Creating a basic Makefile for C++

Putting it all together:

```
main: main.cpp math.cpp
	g++ -I. -Wall -Wpedantic -Werror main.cpp math.cpp -o main
```

- **Target (`main`):** The executable we want to create.
- **Prerequisites (`main.cpp math.cpp`):** The source files required to build the target.
- **Recipe (`g++ [...] main.cpp math.cpp -o main`):** The shell command to compile (`g++`) and link (`-o main`) the source files into the executable (`main`).

Variables for clarity

Enhance readability and maintainability by using variables:

```
CXX=g++
CPPFLAGS=-I.
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror

main: main.cpp math.cpp
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) main.cpp math.cpp -o main
```

- **CXX:** Compiler variable.
- **CPPFLAGS:** Preprocessor flags variable.
- **CXXFLAGS:** Compiler flags variable.

Automatic dependency generation

```
CXX=g++
CPPFLAGS=-I.
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror
DEPS=math.hpp

%.o: %.cpp $(DEPS)
	$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@

main: main.o math.o
	$(CXX) $(CXXFLAGS) $^ -o $@

clean:
	rm -f *.o main
```

- `%.o`: A generic rule for creating files with `.o` extension.
- `$@`, `$<`, `$^`: Automatic variables representing the target, the first prerequisite, and all prerequisites, respectively.

Phony targets

Define phony targets for non-file related tasks:

```
.PHONY: all clean

all: main

main: main.o math.o
	$(CXX) $(CXXFLAGS) $^ -o $<

clean:
	rm -f *.o main
```

- `.PHONY`: Marks targets that don't represent files.
- `all`: Default target.

Variables for source files

```
CXX=g++
CPPFLAGS=-I.
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror
DEPS=math.hpp
SRC=$(wildcard *.cpp)
OBJ=$(SRC:.cpp=.o)

.PHONY: all clean

all: main

main: $(OBJ)
	$(CXX) $(CXXFLAGS) $^ -o $@

%.o: %.cpp $(DEPS)
	$(CXX) -c $(CXXFLAGS) $< -o $@

clean:
	rm -f *.o main
```

Building a library and linking against it

Suppose we have a simple C++ library with two files

- `math.hpp`
- `math.cpp`

Additionally, we have a program, `main.cpp`, that uses functions from this library.

Now, let's create a Makefile to build the library and another one to link our program against it.

Makefile to build a library (1/2)

```
CXX=g++
CPPFLAGS=-I.
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror

SRC=math.cpp
OBJ=$(SRC:.cpp=.o)
OBJ_fPIC=$(SRC:.cpp=.shared.o)
DEPS=math.hpp

LIB_NAME_STATIC=libmath.a
LIB_NAME_SHARED=libmath.so

all: static shared

static: $(LIB_NAME_STATIC)
shared: $(LIB_NAME_SHARED)
```

Makefile to build a library (2/2)

```
$(LIB_NAME_STATIC): $(OBJ)
	ar rcs $@ $^

$(LIB_NAME_SHARED): $(OBJ_fPIC)
	g++ $(CXXFLAGS) -shared $^ -o $@

%.shared.o: %.cpp $(DEPS)
	$(CXX) -c -fPIC $(CPPFLAGS) $(CXXFLAGS) $< -o $@

%.o: %.cpp $(DEPS)
	$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@

clean:
	rm -f *.o $(LIB_NAME_STATIC) $(LIB_NAME_SHARED)
```

Makefile to link against a library

```
CXX=g++
CPPFLAGS=-Imath/
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror
LDFLAGS=-Wl,-rpath,math/ -Lmath/ # For dynamic linking.
# LDFLAGS=-Lmath/ -static        # For static linking.
LDLIBS=-lmath

SRC=main.cpp
OBJ=$(SRC:.cpp=.o)

all: main

main: $(OBJ)
        $(CXX) $(CXXFLAGS) $^ $(LDFLAGS) $(LDLIBS) -o $@

%.o: %.cpp
        $(CXX) -c $< $(CPPFLAGS) $(CXXFLAGS) -o $@

clean:
        rm -f *.o main
```

Summary (1/2)

`make` efficiently determines the need to regenerate a target by checking its existence and the up-to-dateness of prerequisite files. This feature enables it to avoid unnecessary target regeneration.

Make simplifies the installation of numerous libraries through a concise set of commands. A typical sequence for installing an open-source library involves using the following commands:

```
make
make install
```

Typically, the `make` command builds the library, while `make install` copies the library's headers, the libraries and the binaries to a user-specified folder, which defaults to the `/usr` or `/usr/local` directory. This streamlined process facilitates the integration of the installed library into your source code.

Summary (2/2)

In some circumstances, the build process can be optimized by employing the `make -j<N>` command, where `N` represents the number of parallel jobs or commands executed concurrently.

Despite its advantages, Makefiles are platform-dependent, necessitating adaptation to different operating systems. To address this issue, we will explore CMake as a potential solution, providing a platform-independent alternative for managing and generating build systems.

Further readings

- A simple makefile tutorial : Essential tutorial on `make` and Makefile.
- Makefile tutorial : A GitHub repository with numerous makefile examples.
- GNU make : Official documentation for `make` and Makefile.

Exercise 1: building and using `muParserX`

- Download and extract `muParserX`:

```
wget https://github.com/beltoforion/muparserx/archive/refs/tags/v4.0.12.tar.gz
```

- The source files of `muParserX` are located inside the `muparserx-4.0.12/parser/` folder.
- In that folder, write a Makefile to compile `muParserX` into a shared library.
- Write a Makefile that compiles and links the program in `hints/ex1.cpp` with `muParserX`.

Exercise 2: shared libraries

The `hints/ex2/` directory contains a library that implements a gradient descent algorithm for linear regression, accompanied by a source file `ex2.cpp` utilizing this library.

Unfortunately, the gradient descent code within the library contains a bug.

Your tasks are:

1. Compile the library and test file, using the provided Makefiles.
2. Inspect the code to locate the bug within the gradient descent algorithm.
3. Once the bug is identified, fix it within the code. Then, compile an updated version of the library, incorporating the bug fix.
4. Execute the test case to verify that the bug fix successfully addresses the issue. Please note that, since we are dealing with a shared library, this verification should be conducted **without** the need for recompilation or relinking of the test file.

Exercise 3: order matters

The `hints/ex3/` directory contains a source file `ex3.cpp` that uses a library `graphics_lib`, which depends on another library `math_lib`.

1. Generate a static library `libmath.a`.
2. Generate a static library `libgraphics.a`.
3. Compile `ex3.cpp` into an object file `ex3.o`.
4. Link `main.o` against `libmath.a` and `libgraphics.a` to produce the final executable.

What is the correct order for passing `ex3.o`, `libmath.a`, and `libgraphics.a` to the linker to successfully resolve all the symbols?

Would the same considerations apply if dynamic linking (shared libraries) were used instead of static linking?

Exercise 4: dynamic loading

This exercise showcases dynamic loading, the building block for implementing a *plugin* system.

The `hints/ex4/` contains a module `functions` containing the definition of three mathematical functions. The source file `functions.cpp` gets compiled into a shared library `libfunctions.so`, using *C linkage* to prevent *name mangling*.

Notably, when compiling the source file `ex4.cpp` into an executable, there is no need to link against `libfunctions.so`.

1. Fill in the missing parts in `ex4.cpp` to dynamically load the library.
2. Prompt the user for the function name to evaluate at a given point, selecting from the ones available in the library.
3. Perform the evaluation and print the result.
4. Release the library.