Exercise session 07

Introduction to GNU Make. Libraries: building and use.

Advanced Programming - SISSA, UniTS, 2025-2026

Pasquale Claudio Africa

18 Nov 2025

Introduction to GNU Make

Why learn Make?

From last week's lecture, you learned how to create and use static and shared libraries in C++.

But managing compilation commands manually becomes tedious:

```
g++ -c -fPIC math.cpp -o math.fpic.o
g++ -shared math.fpic.o -o libmath.so
g++ -c main.cpp -o main.o
g++ main.o -Lmath/ -lmath -Wl,-rpath,math/ -o main
```

Problems with manual compilation:

- 📝 Repetitive typing of long commands, many flags to remember, error-prone process
- ⏱️ No automatic tracking of which files are changed
- 🔄 Recompiling everything even when only one file is changed
- 👥 Difficult to share build instructions with team members

Make solves these problems by automating the build process with dependency tracking!

Real-world impact of build systems

- Large C++ projects can have **thousands of source files**
- Full rebuild might take **hours** without smart dependency tracking
- Make ensures only **modified files** (and their dependents) are recompiled

Example: Compiling the Linux kernel (~36k source files, ~26k header files)

- Full build: 30-60 minutes
- Incremental build (after small change): **seconds to minutes**

Today's goals:

1. Master Make's dependency tracking and automation
2. Build and link libraries efficiently and apply these skills to real libraries
3. Understand advanced topics: dynamic loading, link order

By the end: You'll have a tool to manage complex C++ projects professionally!

Getting started

Ensure the `make` program is installed by checking `make --version`. If not installed, use package managers such as `apt` on Debian/Ubuntu or `Homebrew` on macOS.

Let's start with a simple C++ program consisting of three files: `math.hpp`, `math.cpp`, and `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`).

To build the target, simply run:

```
make      # Builds the first target in the Makefile ('main', in this case).
make main # Explicitly builds 'main'.
```

Variables for clarity

We can enhance readability and maintainability by using variables:

```
CXX=g++                                                 # The compiler.
CPPFLAGS=-I.                                            # Preprocessor flags.
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror            # Compiler flags.
LDFLAGS=                                                # Linker flags (empty for now).
LDLIBS=                                                 # Libraries to link (empty for now).

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

Note: These variable names (`CXX`, `CPPFLAGS`, `CXXFLAGS`, `LDFLAGS`, `LDLIBS`) are GNU Make conventions. Using them allows users to override compilation settings from the command line:

```
make CXX=clang++ CPPFLAGS="-I/home/my/usr/include" CXXFLAGS="-O3 -march=native"
```

Pattern rules and automatic variables (1/2)

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

# This rule requires manually listing headers in DEPS.
# See next slide for automatic dependency generation.
%.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.

Pattern rules and automatic variables (2/2)

For truly automatic dependency generation, modern Makefiles use compiler flags like `-MMD -MP` to generate `.d` files containing actual header dependencies.

```
DEPFLAGS=-MMD -MP
DEPS=$(OBJ:.o=.d) # Pattern substitution.

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

-include $(DEPS)
```

Explanation:

- `-MMD`: Generate dependency files (`.d`) during compilation.
- `-MP`: Add phony targets for each dependency to avoid errors if headers are deleted.

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.

Without `.PHONY`, if a file named `clean` or `all` exists in the directory, `make` would check its timestamp instead of always executing the recipe.

Variables for source files

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

.PHONY: all clean

all: main

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

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

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

Debugging Makefiles

1. Print variable values:

```
$(info SRC=$(SRC))
$(info OBJ=$(OBJ))
```

2. Dry run (show commands without executing):

```
make -n
```

3. Print detailed information:

```
make --debug=v # Verbose debugging output.
```

4. See why Make rebuilds a target:

```
make -d
```

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=.fpic.o)
DEPS=math.hpp

LIB_NAME_STATIC=libmath.a
LIB_NAME_SHARED=libmath.so

.PHONY: all static shared clean

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)
	$(CXX) $(CXXFLAGS) -shared $^ -o $@

%.fpic.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)
        # Order matters! Object files first, then library paths, then libraries.
        $(CXX) $(CXXFLAGS) $^ $(LDFLAGS) $(LDLIBS) -o $@

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

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

Summary (1/2)

How Make determines what to rebuild:

1. Check if the target exists.
2. Compare timestamps of the target and its prerequisites.
3. Rebuild if any prerequisite is newer than the target.
4. Recursively check prerequisites of prerequisites.

A typical sequence for building and installing a 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 possible binaries to a user-specified folder, which defaults to the `/usr` or `/usr/local` directory.

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 (use `-j$(nproc)` for automatically using all the available cores).

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
tar xzvf 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 `libmuparserx.so`.
  Bonus: add a target `install` that copies header files and the shared library into a user-specified folder.
- Write a Makefile that compiles and links the program in `hints/ex1.cpp` against `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?

(⚠ Advanced) Exercise 4: dynamic loading

The `hints/ex4/` directory 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*.

1. Fill in the missing parts in `ex4.cpp` to dynamically load the library, using the functions `dlopen()`, `dlsym()`, and `dlclose()`.
2. Prompt the user for the name of the function 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.

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