CSCI 112 Programming with C

Lab makefile

Outline

- Compiler
- Makefile

Makefile

What is Make - ? (mingw32-make.exe)

Makefile is an **automation tool** that manages the compilation and building process for software projects. It primarily serves two main purposes:

1. **Dependency Tracking:** Make reads a file called the **Makefile**, which lists all the **dependencies** (input files, like code or headers) needed to create a specific **target** (output file, like a program or object file).
2. **Smart Execution:** Make checks the **timestamps** of the target and its dependencies. If any input file is newer than the output file, Make knows the output is **out-of-date** and automatically runs the necessary commands to **rebuild only what has changed**.

Key Takeaway

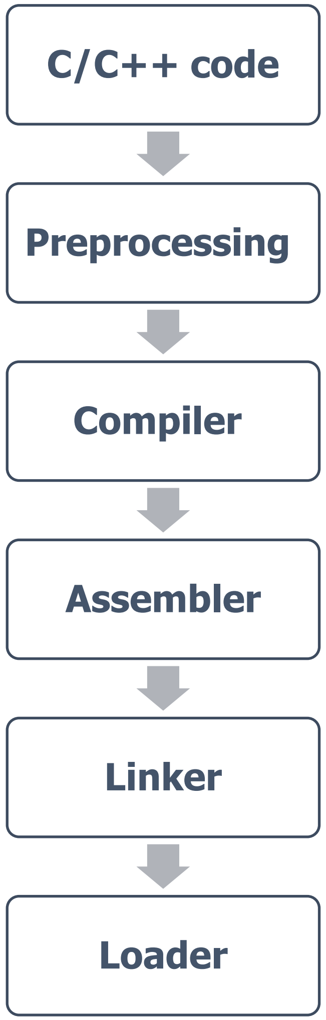
- **Make ensures that no unnecessary rebuilding is done, saving time and simplifying complex build steps.** It acts as a smart project manager that only works when work is actually needed.

Compiler

Compiler vs Interpreter

- A compiler takes the entire source code and translates it into a machine code file, often called an executable. This executable file contains instructions that the computer's processor can directly execute. Once compiled, the program can run independently without the need for the original source code or a compiler.
- An interpreter translates the source code line by line as the program is running. It doesn't create a separate executable file. Instead, it uses a virtual machine to execute the translated code. The virtual machine provides an environment that mimics a real computer, allowing the program to run even if the underlying hardware architecture is different

How does a C program executes?



This is the source code you have written in the C/C++ programming language. It forms the basis of your program.

During this stage, the preprocessor processes your source code. It includes header files (e.g., stdio.h, math.h) using directives like #include and expands macros defined with #define. The output of this stage is the preprocessed source code.

The compiler takes the preprocessed source code and translates it into assembly code. This assembly code is a low-level representation of your program that is specific to the target architecture.

The assembler converts the assembly code **into object code**. Object code is a machine-readable format that contains instructions and data.

The linker combines the object code with other necessary libraries (e.g., standard C library) to create the final executable program. This process resolves external references and creates a complete program that can be run.

The loader loads the executable program into memory (RAM) so that it can be executed by the CPU. The running program is often referred to as a process.

These steps are essential for transforming your C code into an executable program that can be run on a computer.

Example 1 - main.c

- Create a new folder,
- create a file named main.c inside it and fill it with the simplest possible code.

```c
// main.c
#include<stdio.h>

int main()
{
  printf("%s\n", "Hello, world!");

  return 0;
}
```

Process of compilation

1. Compile main.c into an object file main.o:

```
gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
```

```
# 1) Compile main.c into an object file main.o (no linking).
#    Includes debug symbols (-g), enables most warnings (-Wall), uses the C99 standard (-std=c99),
#    and enforces strict standard conformance (-pedantic).
```

2. Link the object file main.o into an executable main.exe:

```
gcc -g main.o -o main.exe
```

```
# 2) Link the object file into an executable named main.exe.
```

Process of compilation

3. Compile and link in one step (from main.c directly to main.exe):

```
gcc -g -Wall -std=c99 -pedantic main.c -o main.exe
```

```
# 3) Compile and link in one step: from main.c directly to main.exe,
#    with the same diagnostic/standard flags as in step 1.
```

About flags

- The -g flag tells the compiler to include debugging information in the output. Without it, setting breakpoints in the compiled file would not be possible. This flag should be disabled in the final compilation of the application after the development process is complete.
- The flags -Wall, -std, and -pedantic are compiler-specific and only necessary during the creation of object files. The -c and -o flags indicate source and object files, respectively.
- Therefore, if the compilation process is split into two stages — compilation and linking — only the -g flag should be repeated during linking.

What Does a Make Rule Look Like

A **Makefile** is just a text file with rules. Each rule has **four key parts**:

1. **Target** – a symbolic name.
2. **Colon :** – separates the target from its dependencies.
3. **Prerequisites (Dependencies):**
    1. another file (like utils.o)
    2. another target (for example, clean, build, all)
    3. mix of both
4. **Recipe** – one or more shell commands (written exactly as if you typed them in the terminal).
    - Each command **must start with a tab**.
    - Commands run one by one. If one fails (non-zero exit code), make stops.

```
target … : prerequisites …
        recipe
        …
        …
```

Simplest makefile

- Create a file "Makefile "

- And run in cmd: "mingw32-make"

```
target … : prerequisites …
        recipe
        …
        …
```

```
build: #The comment preceded by a hashmark
	gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
	gcc -g main.o -o main.exe
```

- The first command **compiles** the source code into an **object file**,

- The second command **links** the object file to produce the final **.exe executable**

Simplest makefile

- Modify to:

```
build_exe:
	gcc -g main.o -o main.exe
build_obj:
	gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
```

- And run in cmd: "mingw32-make"

Without any parameters, **make** will only execute the first target. Our first target is linking, so it will create the **.exe** file from the **.o** file. If we wanted to build the **.obj** (object file) alone, we would have to call that target as a program parameter: **'mingw32-make build_obj'**, followed by either **'mingw32-make'** or **'mingw32-make build_exe'**.

Simplest makefile

- Modify to:

```
build_exe: build_obj
     gcc -g main.o -o main.exe
build_obj:
     gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
```

- And run in cmd: "mingw32-make"

We can also modify the Makefile where **'build_obj' is a dependency of our 'build_exe' target**. This means that **'build_obj' will be executed before 'build_exe' runs**.

Consequently, we can then run **make** without any parameters, as it will find the dependencies and execute the entire compilation process.

Example 2 – main.c; other.h; other.c

- Modify and create files:

```c
// main.c
#include<stdio.h>
#include "other.h"

int main()
{
  extern_function();
  return 0;
}
```

```c
// other.h
#ifndef _OTHER
#define _OTHER
void extern_function();
#endif  //_OTHER
```

```c
// other.c
#include <stdio.h>
#include "other.h"
void extern_function()
{
    printf("text from extern_function - other.c\n");
}
```

Process of compilation

1. Compiling:

```
gcc -g -Wall -std=c99 -pedantic -c other.c -o other.o
```

```
gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
```

2. Linking:

```
gcc -g main.o other.o -o main.exe
```

Headers and the Compilation Process – Key Rules

1. **Including the same header more than once**
   - If the same header.h is included in multiple places, the compiler may get confused (errors like *redefinition*). To prevent this, we use *include guards*: #ifndef etc. This ensures the header is included only once.
2. **How to write #include**
   - In source files, we always use **just the filename**:
   - #include "header.h"
   - Do not write paths like ../include/header.h.
   - The compiler, with its options, decides where to look for headers. This keeps the code clean and independent from the folder structure.
3. **Role of headers in the compilation process**
   - Header files (.h) are **not passed directly to the compiler**.
   - The compiler automatically pulls them in if they are included with #include in .c files.
   - That's why in a Makefile, **we don't compile header files** – instead, we list them as **dependencies**. This way, if a header changes, the correct .o files will be rebuilt.

The key distinction: Headers are dependencies in the Makefile, but not arguments for the compiler. That's something students often confuse, so it's worth stressing early.

makefile

From just three command-line calls, we can create a **Makefile** that automates the exact same functionality. However, at this stage, our Makefile is essentially no different from a simple **batch script** (or a .bat file). Every single time we invoke **make**, the entire compilation process is executed from start to finish, regardless of whether any files have actually changed.

```
gcc -g -Wall -std=c99 -pedantic -c other.c -o other.o
gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
gcc -g main.o other.o -o main.exe
```

```
# makefile
build_exe: build_obj
    gcc -g main.o other.o -o main.exe
build_obj:
    gcc -g -Wall -std=c99 -pedantic -c other.c -o other.o
    gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
```

makefile

This happens because both of our current targets are essentially **artificial targets** (or **PHONY targets**). Their dependencies don't rely on the existence of any files; they simply establish that one target requires the execution of the other. The line defining this—**.PHONY: target1 target2...**—should be placed before the first occurrence of the first target. By doing this, **make** will not attempt to search for files with the same name as the target.

```
gcc -g -Wall -std=c99 -pedantic -c other.c -o other.o
gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
gcc -g main.o other.o -o main.exe
```

```
# makefile
.PHONY: build_exe build_obj
build_exe: build_obj
    gcc -g main.o other.o -o main.exe
build_obj:
    gcc -g -Wall -std=c99 -pedantic -c other.c -o other.o
    gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
```

The Nature of Make Targets

Targets in a Makefile can fall into several categories:

- **Abstract, Symbolic Names:** A target that is purely an artificial name (like clean) and does not correspond to an actual file in the directory.
- **A File Name:** A target that is the name of an existing or yet-to-be-created file (like main.exe or main.o).
- **Both:** A target can often be both a symbolic step *and* a file name, depending on the context.

How Make Treats Targets (The Decision-Making Process)

If a target is **not listed** in the **.PHONY** directive, make treats the target name as a **file name** and initiates a smart, two-step process:

1. **Existence Check:** make first checks if the file named by the target actually exists. If the file **does not exist**, make automatically executes the target's recipe to create it.
2. **Timestamp Comparison:** If the file **does exist**, make compares the **modification time** (timestamp) of the target file with the timestamps of all its prerequisites (dependencies).
   - If any of the **prerequisites is newer** (younger) than the target file, the target is considered **out-of-date**, and its recipe is executed. This forces the older file to be recompiled (or otherwise updated) to a newer version.

Why This Still Matters

While computational power is high today, this timestamp logic is still crucial, especially in very large projects. There is no reason to recompile every single part of a huge system if only a few small source files have changed. **This selective rebuilding is the core reason we use make—it saves enormous amounts of time and makes the build process efficient.**

Fully Functional Makefile: A Summary

```
.PHONY: clean

main.exe: main.o other.o
    gcc -g main.o other.o -o main.exe

main.o: main.c  # we dont have the main.h
    gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o

other.o: other.h other.c
    gcc -g -Wall -std=c99 -pedantic -c other.c -o other.o
clean:
    del main.o
    del other.o
```

At this point, we have a **complete and functional Makefile**. If we run it without any parameters, **make** will automatically build the final executable, main.exe.

Fully Functional Makefile: A Summary

Make achieves this by generating a **dependency tree** and determining the correct order of execution. We've defined this order as follows:

- The final executable (main.exe) depends on both object files: main.o and other.o.
- These object files, in turn, have their own prerequisites: main.o depends only on its source code, main.c, while **other.o depends on both its source file (other.c) AND its header file (other.h)**. This last dependency is crucial, ensuring that if the header is modified, other.o will be recompiled.

```
.PHONY: clean
main.exe: main.o other.o
    gcc -g main.o other.o -o main.exe
main.o: main.c  # we dont have the main.h
    gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
other.o: other.h other.c
    gcc -g -Wall -std=c99 -pedantic -c other.c -o other.o
clean:
    del main.o
    del other.o
```

Fully Functional Makefile: A Summary

- We also successfully created a **symbolic target, clean**, for removing temporary object files generated during compilation. This target will **not** be executed automatically; we must explicitly call it by name: make clean.
- Crucially, we only listed clean in the **.PHONY** directive because it's the only truly artificial target. All other targets (main.exe, main.o, other.o) are meant to be treated as actual file names, allowing **make** to use its smart timestamp logic.
- Note that while we needed to list other.h as a dependency for other.o, we **did not** need to define a separate target for it, as headers are only inputs, never outputs.

```
.PHONY: clean
main.exe: main.o other.o
    gcc -g main.o other.o -o main.exe
main.o: main.c  # we dont have the main.h
    gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o
other.o: other.h other.c
    gcc -g -Wall -std=c99 -pedantic -c other.c -o other.o
clean:
    del main.o
    del other.o
```

Modify and Run

```
mingw32-make clean
mingw32-make
mingw32-make
```

When we execute these commands sequentially in the command prompt (CMD):

1. **mingw32-make clean**: The process begins by removing the object files.
2. **First mingw32-make**: Because the object files no longer exist, the default target (main.exe) is triggered. Since main.exe depends on those missing object files, the entire compilation and linking process is executed from scratch to fulfill the dependency chain.

- **Second mingw32-make**: The subsequent invocation will return the message: **mingw32-make: 'main.exe' is up to date.**

This final message confirms that **make** has checked all dependencies and creation dates for every file. It found that the current version of main.exe is the newest one, meaning there is no need for **make** to perform any actions or recompilation.

Variables and Assignment in Makefiles

- In a Makefile, we can greatly enhance flexibility and readability by using **variables**. We've used the **:=** symbol for defining our variables, which denotes **simple assignment**.
- While alternative assignment operators exist (like =, !=, ?=, etc.), it's crucial to understand that make operates in **two distinct phases** (reading and execution). Using the simple assignment operator (:=) is the safest approach for creating straightforward string variables at this level. You should generally **avoid the = operator** for now, as it defines a **recursively expanded variable**, which is reserved for much more complex and advanced scenarios.
- When defining a variable, the content is **not enclosed in quotes**. The variable definition ends either at a newline character '\n' or at the comment symbol '#'. A common pitfall for new users is dealing with whitespace: trailing spaces are often invisible but can be unintentionally included in the variable's value, which may cause build errors.
- To use a variable anywhere in the Makefile, you must enclose its name in parentheses preceded by a dollar sign: **$(VARIABLE_NAME)**.

extension

```
.PHONY: clean

main.exe: main.o other.o
    gcc -g main.o other.o -o main.exe

main.o: main.c  # we dont have the main.h
    gcc -g -Wall -std=c99 -pedantic -c main.c -o main.o

other.o: other.h other.c
    gcc -g -Wall -std=c99 -pedantic -c other.c -o other.o
clean:
    del main.o
    del other.o
```

```
CC     := gcc
CFLAGS := -g -Wall -std=c99 -pedantic # compiler flags
LFLAG  := -g # linker flag

TARGET := main.exe
OBJS   := main.o other.o
.PHONY: all clean

all: $(TARGET)
$(TARGET): $(OBJS)
    $(CC) $(LFLAG) $(OBJS) -o $(TARGET)
main.o: main.c  # we dont have the main.h
    $(CC) $(CFLAGS) -c main.c -o main.o
other.o: other.h other.c
    $(CC) $(CFLAGS) -c other.c -o other.o
clean:
    del $(OBJS)
```

extension

```
CC     := gcc
CFLAGS := -g -Wall -std=c99 -pedantic # compiler flags
LFLAG  := -g # linker flag

TARGET := main.exe
OBJS   := main.o other.o
.PHONY: all clean

all: $(TARGET)
$(TARGET): $(OBJS)
    $(CC) $(LFLAG) $(OBJS) -o $(TARGET)
main.o: main.c  # we dont have the main.h
    $(CC) $(CFLAGS) -c main.c -o main.o
other.o: other.h other.c
    $(CC) $(CFLAGS) -c other.c -o other.o
clean:
    del $(OBJS)
```

- Our script is clearly becoming much smarter and more automated. As you can easily observe, **make** inserts the values of our variables wherever we reference them. The commonly accepted **default name for the main build target is all**, which we've defined as a **PHONY** target dependent on our actual **$(TARGET)** variable.
- However, it's essential to remember that this current Makefile is designed exclusively for the **Windows CMD** environment. If we tried to run the clean target in a Linux/Unix environment (such as **Bash**), it would fail because the command **del** is not recognized there.

Thank you