Misagh Mohaghegh

Multi-File Projects & Makefile

Advanced Programming

University of Tehran

Why Multi-File?

- Easier navigation
- Easier to understand
- Faster build times
- Working on different files

A Common Mistake

Don’t single file then multi-file!

Compilation Steps

- Preprocessor
- Compiler
- Assembler
- Linker

Preprocessor

- Outputs the original code, with preprocessing done
- Preprocessing such as:
  - Macro and define expansion
  - Include expansion
  - Removing comments

```
g++ -E file.cpp -o file.i
```

Compiler

- Outputs the assembly CPU instructions of code
- Different output for different CPU architectures

```
g++ -S file.cpp -o file.s
```

Assembler

- Outputs the binary machine code of the assembly file

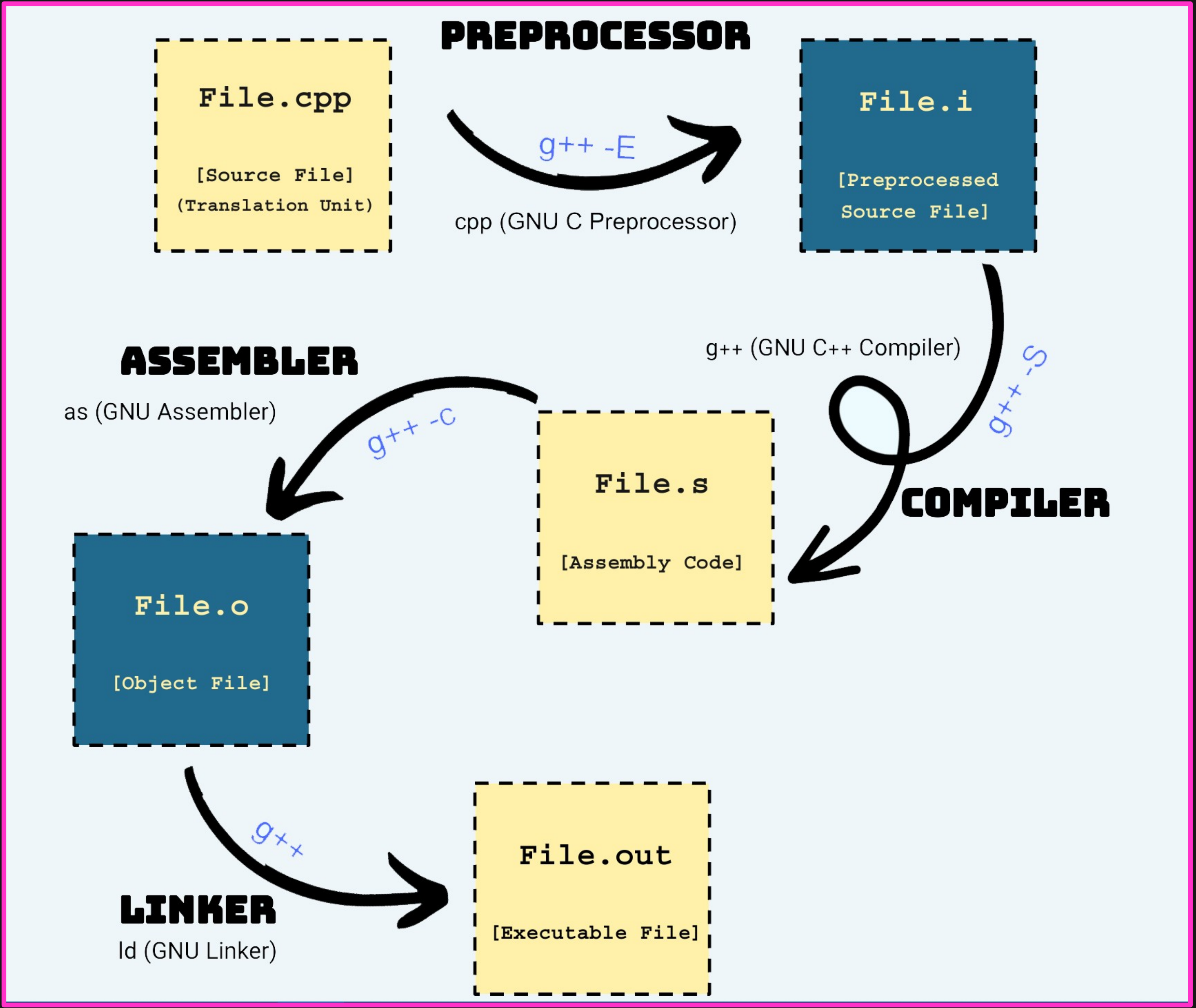
```
g++ -c file.cpp -o file.o
```

Linker

- Outputs the executable code
- Combines all object files (.o) and libraries (.a)

```
g++ file1.o file2.o -o file.out
```

Overview



PREPROCESSOR
File.cpp
[Source File]
(Translation Unit)
g++ -E
cpp (GNU C Preprocessor)
File.i
[Preprocessed Source File]
g++ (GNU C++ Compiler)
g++ -S
COMPILER
File.s
[Assembly Code]
ASSEMBLER
as (GNU Assembler)
g++ -c
File.o
[Object File]
g++
LINKER
ld (GNU Linker)
File.out
[Executable File]

How about…

```
g++ file.cpp –o file.out
```

- Does the 4 step process for file.cpp
- Single file, a single change requires full re-compilation

How about multiple files?

Enter Makefile

- We want files to only recompile if they have changed
- Dependency management
- In a Makefile, we write the dependencies of each source file
- Make uses the last modified date of the file to decide if it should recompile

Makefile Rule

```
target: prerequisites
⟷ recipe
```

- Missing separator error
- An example rule:

```
file.o: file.cpp file.hpp file2.hpp
g++ -c file.cpp -o file.o
```

Automatic Variables (1)

- If we have:

```
file.o: file.cpp file.hpp file2.hpp
```

- Then we can use:

```
$@ = file.o                          (the target)
$^  = file.cpp file.hpp file2.hpp    (all prerequisites)
$<  = file.cpp                       (the first prerequisite)
```

Automatic Variables (2)

- We can turn:

```
file.o: file.cpp file.hpp file2.hpp
g++ -c file.cpp -o file.o
```

- Into:

```
file.o: file.cpp file.hpp file2.hpp
g++ -c $< -o $@
```

Variables

- We can use variables to increase readability

```
FILES = src/file1.cpp \
        src/file2.cpp
```

- Variables can be used as follows: (variable expansion)

```
$(FILES) or ${FILES}
```

Variable Assignment

- There are 2 ways to assign variables, = and :=

comment →

```
foo = abc
bar = $(foo) bar
foo = xyz
# $(bar) is now "xyz bar"
```

Functions

- Functions are written in $(), the first word is the name

```
$(info Printing a variable: $(var))
```

```
$(wildcard *.txt)
```

```
FILES = $(shell find src/ -name "*.cpp" -type f)
```

Substitution Functions

- Output: file.cpp file3.cpp

```
$(patsubst pattern,replacement,text)
$(patsubst %.o,%.cpp,file.o file2.s file3.o)
```

- Substitution reference:

```
$(var:pattern=replacement)
$(var:%.o=%.cpp)
```

Recipe

- Recall the Makefile rule:

```
target: prerequisites
	recipe
```

- Recipe can be multiple lines which are all executed

```
print-help:
	echo This prints echo and the text
	@echo This only prints the text
	echo This doesn’t show the text >/dev/null
```

Target

- The special target .PHONY is used for targets that are not files

```
.PHONY: all print-help clean
all: $(EXE)
# ...
clean:
	rm -f $(FILES)
```

Misagh Mohaghegh

Advanced Programming

University of Tehran

Multi-File Projects & Makefile

The End