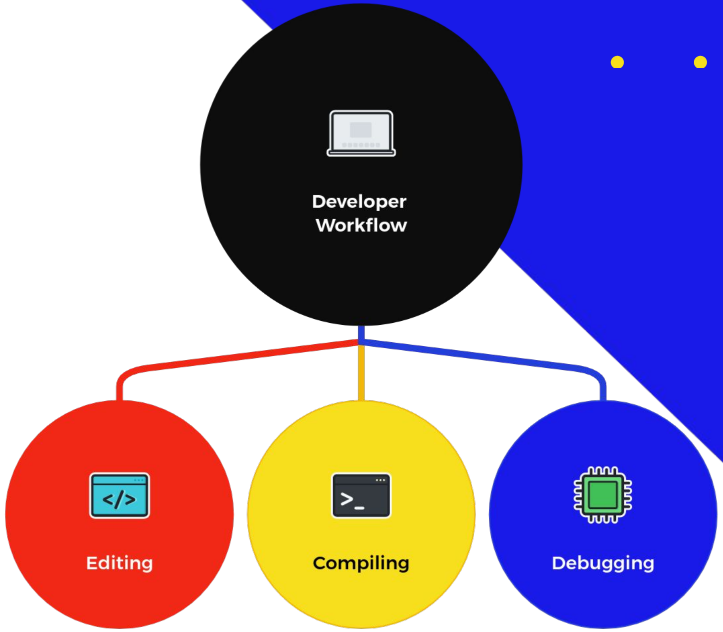
Makefile

By: Ali Ghorab

Contents

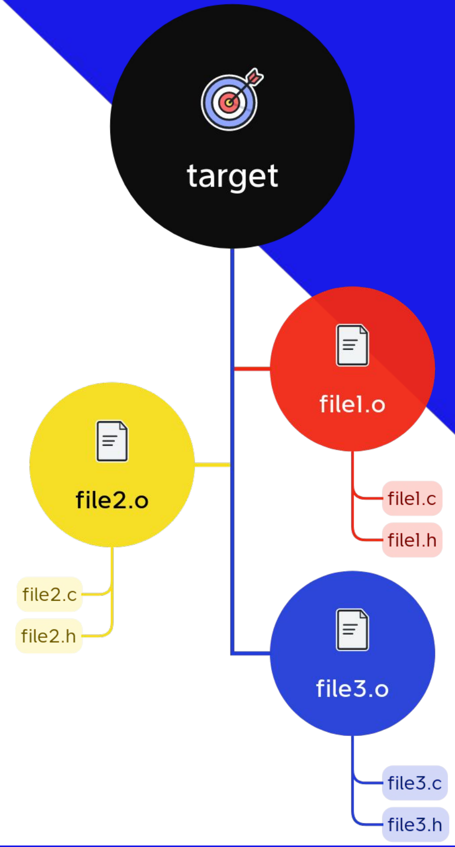
The Problem

- The developer workflow usually follows a fairly simple routine of:
    - Editing source files
    - Compiling the source
    - Debugging the result
- Converting source code into an executable can be time-consuming.
- Procedural errors, such as not re-compiling or re-linking, can cause frustration and make the process error-prone, especially with complex programs.



The Solution: Build Tools

- The *build tools* are intended to automate the repetitive aspects of transforming source code into an executable.
- *`make`* as a build tool offers advantages over scripts by allowing you to define relationships between program elements.
- It optimizes the build process by redoing only necessary steps based on timestamps and specified dependencies.



Makefile: Basics

Syntax

```
target [target ...]: [dependency ...]
<Tab>[command 1]
<Tab>.
<Tab>.
<Tab>.
<Tab>[command n]
```

Makefile: Basics

Example 1: hello from bash!

```
hello:
	echo "Hello from bash!"
```

Makefile: Basics

Example 2: hello from C!

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("Hello from C!\n");
    return 0;
}
```

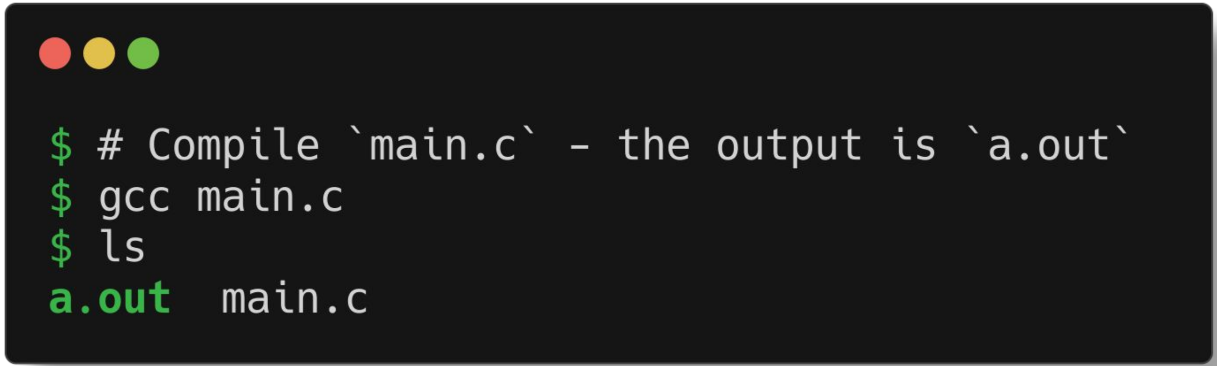
Makefile: Basics

Example 2: hello from C!

- gcc - GNU project C and C++ compiler
- GCC, or GNU Compiler Collection, is a suite of compilers for various programming languages like C, C++, Objective-C, Fortran, Ada, D, and Go.

Makefile: Basics

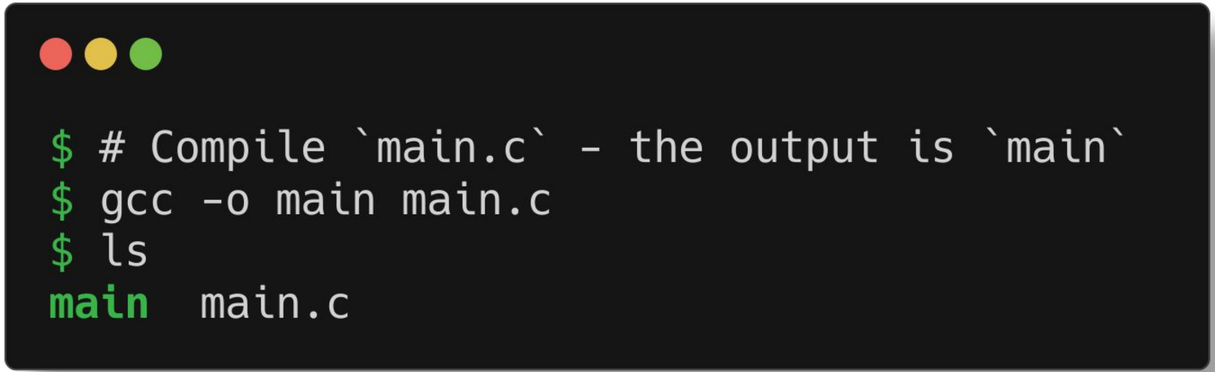
Example 2: hello from C!



```
$ # Compile `main.c` - the output is `a.out`
$ gcc main.c
$ ls
a.out  main.c
```

Makefile: Basics

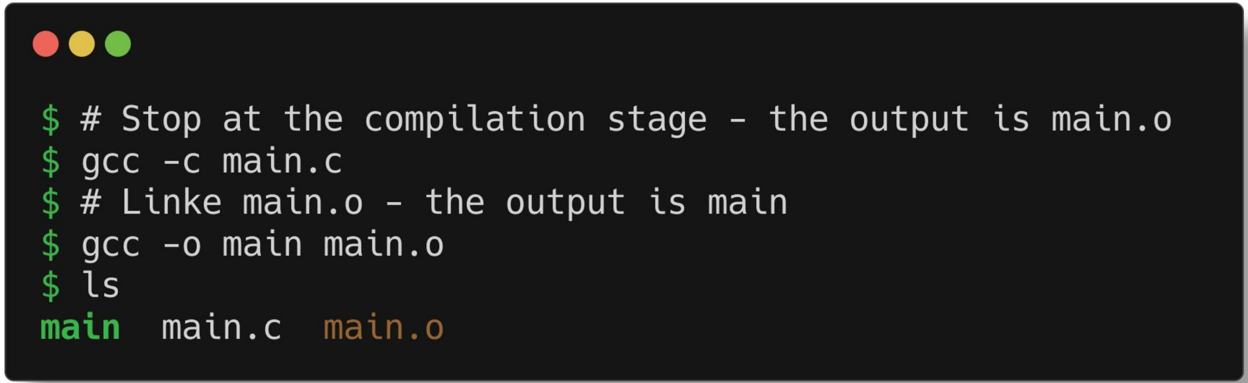
Example 2: hello from C!



```
$ # Compile `main.c` - the output is `main`
$ gcc -o main main.c
$ ls
main  main.c
```

Makefile: Basics

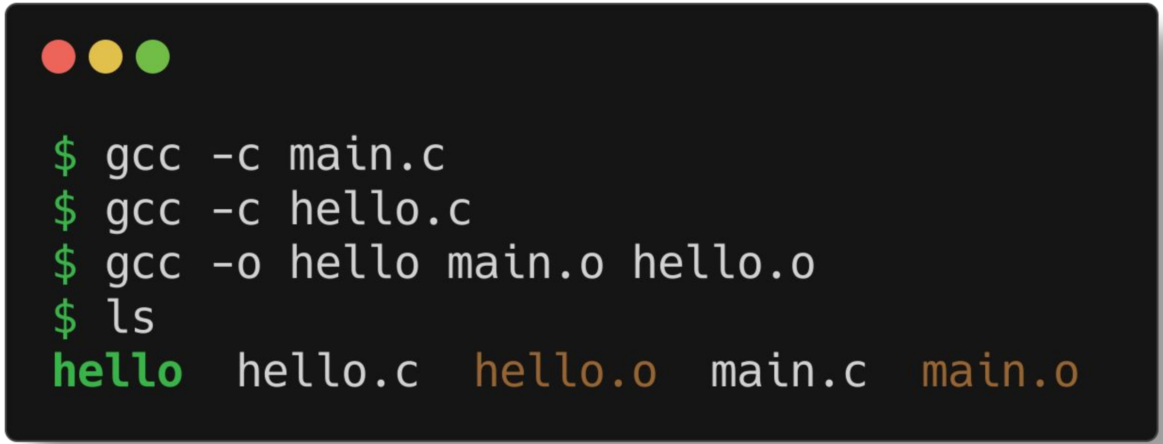
Example 2: hello from C!



```
$ # Stop at the compilation stage - the output is main.o
$ gcc -c main.c
$ # Linke main.o - the output is main
$ gcc -o main main.o
$ ls
main  main.c  main.o
```

Makefile: Basics

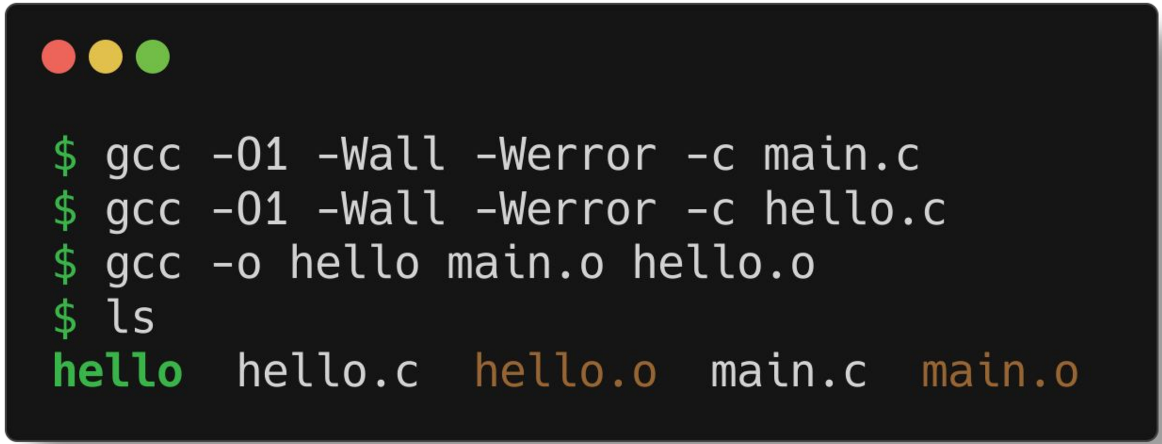
Example 2: hello from C!



```
$ gcc -c main.c
$ gcc -c hello.c
$ gcc -o hello main.o hello.o
$ ls
hello  hello.c  hello.o  main.c  main.o
```

Makefile: Basics

Example 2: hello from C!



```
$ gcc -O1 -Wall -Werror -c main.c
$ gcc -O1 -Wall -Werror -c hello.c
$ gcc -o hello main.o hello.o
$ ls
hello  hello.c  hello.o  main.c  main.o
```

Makefile: Basics

Example 2: hello from C!

```
all: hello

hello: hello.c
	gcc -O1 -Wall -Werror -o hello hello.c

.PHONY: clean

clean:
	rm -rf hello
```

Makefile: Basics

Example 2: hello from C!

```
RM := rm -rf

CC := gcc
CFLAGS := -O1 -Wall -Werror

SRC := hello.c
TARGET := hello
```

Makefile: Basics

Example 2: hello from C!

```makefile
all: $(TARGET)

$(TARGET): $(SRC)
	$(CC) $(CFLAGS) -o $(TARGET) $(SRC)

.PHONY: clean

clean:
	$(RM) $(TARGET)
```

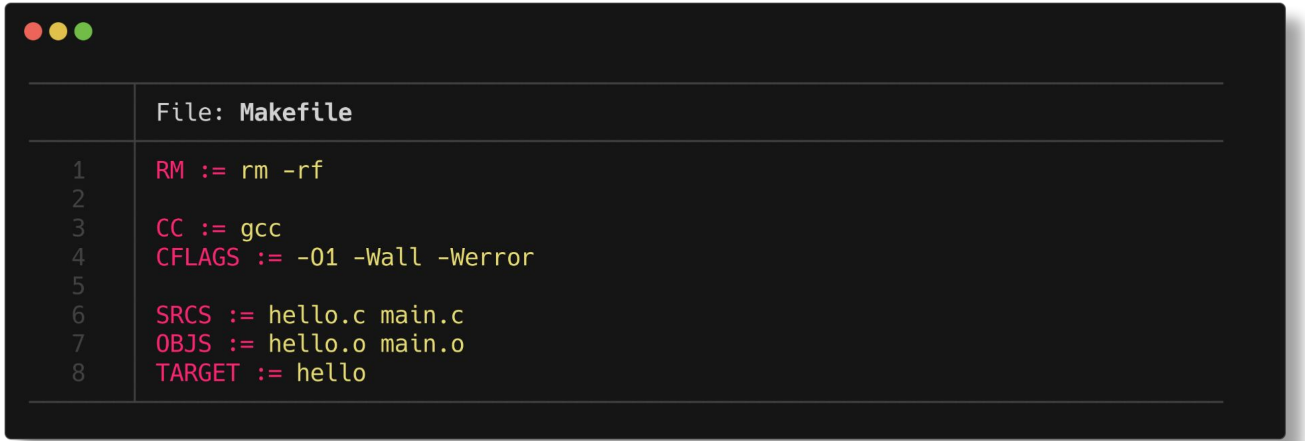
Makefile: Basics

Lab 1

- Create a Makefile that builds a project with multiple source files and resolves dependencies automatically. Each source file should be compiled into an object file, and the final executable should be linked from these object files.

Makefile: Basics

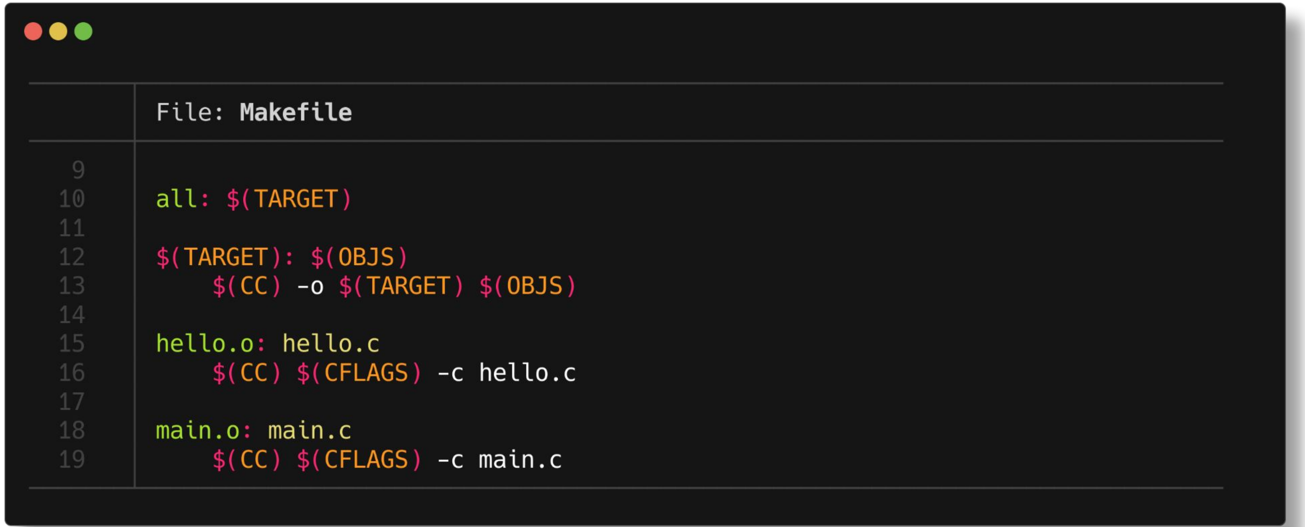
Lab 1


File: **Makefile**

```
RM := rm -rf

CC := gcc
CFLAGS := -O1 -Wall -Werror

SRCS := hello.c main.c
OBJS := hello.o main.o
TARGET := hello
```

Makefile: Basics

Lab 1

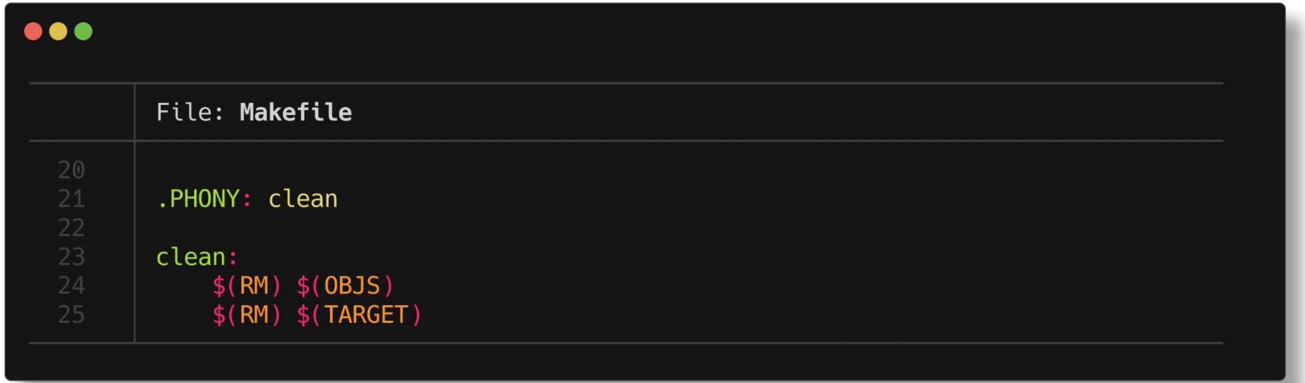


File: **Makefile**

```

all: $(TARGET)

$(TARGET): $(OBJS)
	$(CC) -o $(TARGET) $(OBJS)

hello.o: hello.c
	$(CC) $(CFLAGS) -c hello.c

main.o: main.c
	$(CC) $(CFLAGS) -c main.c
```

Makefile: Basics

Lab 1


File: **Makefile**

```
.PHONY: clean

clean:
	$(RM) $(OBJS)
	$(RM) $(TARGET)
```

Makefile: Variables

Overview

- Make is a language consisting of two parts: one for describing dependency graphs and another for textual substitution.
- It allows you to define shortcuts for longer sequences of characters.
- Makefile variables differ from traditional programming variables as they are expanded in place to form text strings.

Makefile: Variables

1. Simply Expanded Variables

- A *simply expanded* variable (or a simple variable) is defined using the `:=` assignment operator
- It is called "*simply expanded*" because its right-hand side is expanded immediately upon reading the line from the *`makefile`*.

Makefile: Variables

2. Recursively Expanded Variables

- A *recursively expanded* variable (or a recursive variable) is defined using the = assignment operator
- It is called “*recursively/lazily expanded*” because variables in `makefiles` are stored as values without immediate evaluation.
- Evaluation occurs when the variable is used, allowing deferred or "*lazy*" expansion, enabling assignments to be performed in a non-sequential order.

Makefile: Variables

3. Automatic Variables

- Automatic variables are set by make after a rule is matched.
- They provide access to elements from the target and prerequisite lists so you don’t have to explicitly specify any filenames.
- They are very useful for avoiding code duplication, but are critical when defining more general pattern rules

Makefile: Variables

3. Automatic Variables

- These are the common automatic variables:
  1. `$@` The filename representing the target.
  2. `$<` The filename of the first prerequisite.
  3. `$^` List of prerequisite filenames with duplicates removed, typically used for compiling, copying, etc., separated by spaces.
  4. `$?` The names of all prerequisites that are newer than the target, separated by spaces.

Makefile: Variables

Lab 2

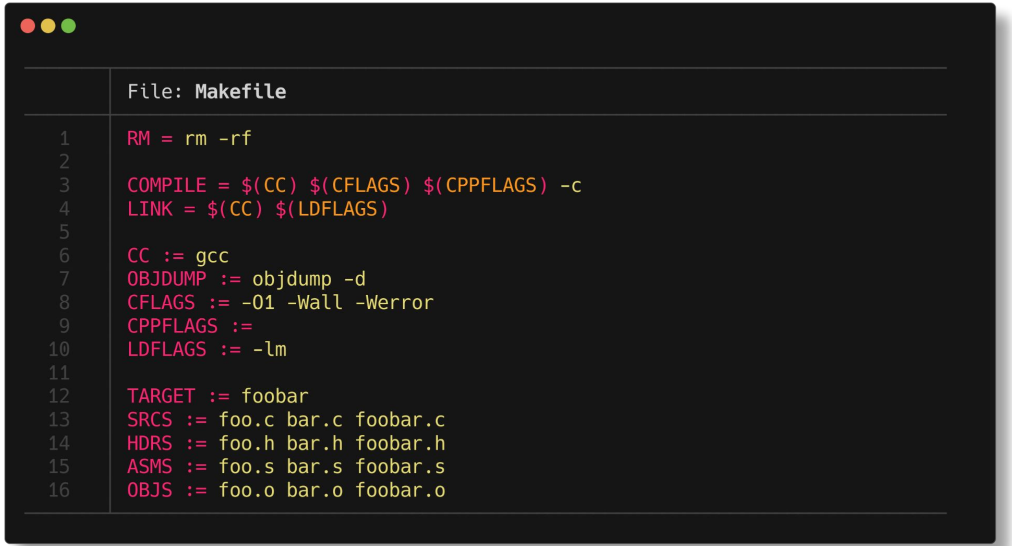
Create a *`Makefile`* that generates a disassembly *`filename.s`* after compiling each source file and *`target.s`* after linking the executable.

Notes:

- Don't use *`gcc -S`* flag, use *`objdump`* instead
- Search *`objdump`* or *`man objdump`* for more info

Makefile: Basics

Lab 2

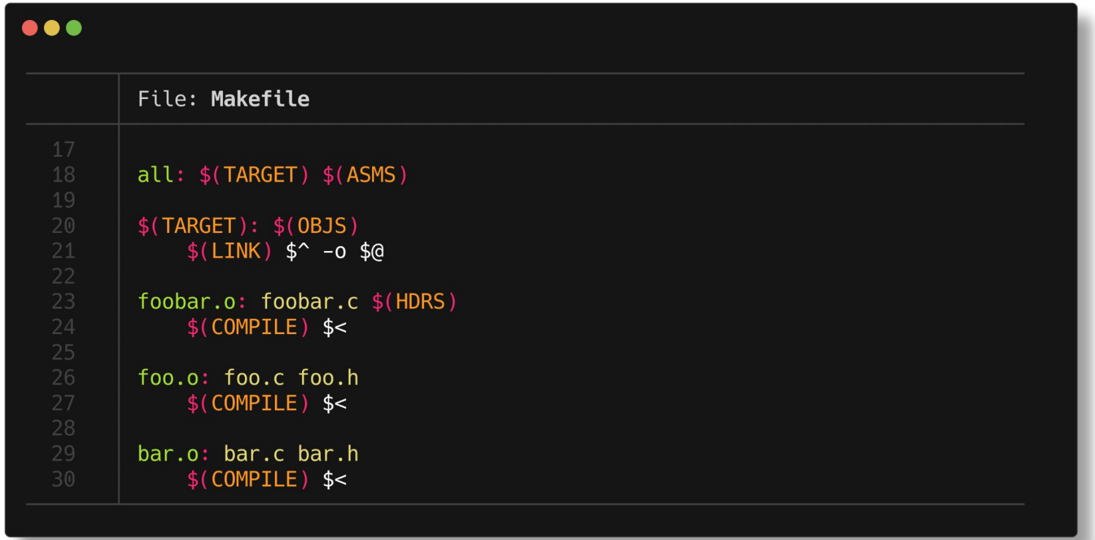


File: **Makefile**

```
RM = rm -rf

COMPILE = $(CC) $(CFLAGS) $(CPPFLAGS) -c
LINK = $(CC) $(LDFLAGS)

CC := gcc
OBJDUMP := objdump -d
CFLAGS := -O1 -Wall -Werror
CPPFLAGS :=
LDFLAGS := -lm

TARGET := foobar
SRCS := foo.c bar.c foobar.c
HDRS := foo.h bar.h foobar.h
ASMS := foo.s bar.s foobar.s
OBJS := foo.o bar.o foobar.o
```

Makefile: Basics

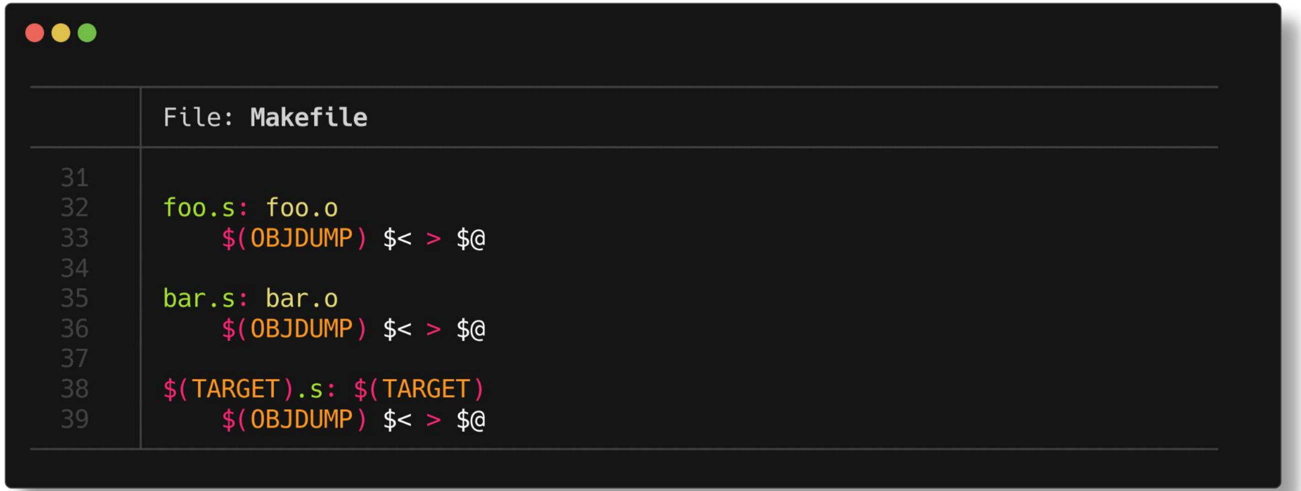
Lab 2


File: Makefile

```makefile

all: $(TARGET) $(ASMS)

$(TARGET): $(OBJS)
	$(LINK) $^ -o $@

foobar.o: foobar.c $(HDRS)
	$(COMPILE) $<

foo.o: foo.c foo.h
	$(COMPILE) $<

bar.o: bar.c bar.h
	$(COMPILE) $<
```

Makefile: Basics

Lab 2

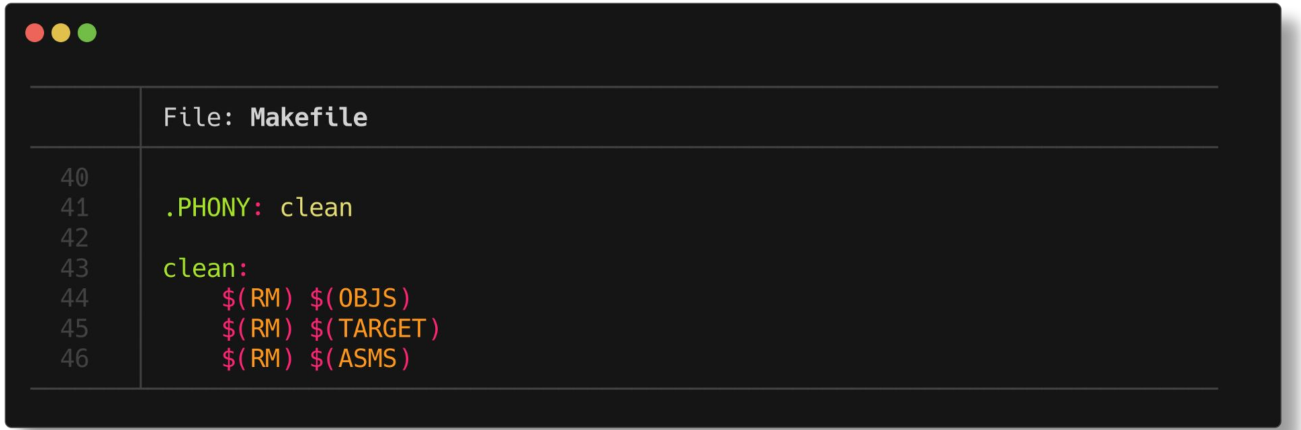


File: **Makefile**

```
foo.s: foo.o
	$(OBJDUMP) $< > $@

bar.s: bar.o
	$(OBJDUMP) $< > $@

$(TARGET).s: $(TARGET)
	$(OBJDUMP) $< > $@
```

Makefile: Basics

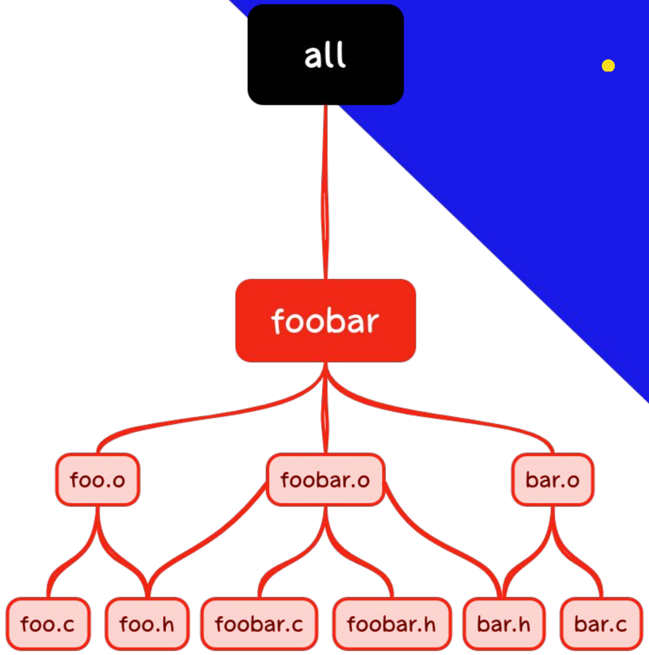
Lab 2



```
File: Makefile

.PHONY: clean

clean:
	$(RM) $(OBJS)
	$(RM) $(TARGET)
	$(RM) $(ASMS)
```

Makefile: Rules

- The target of one rule can be referenced as a prerequisite in another rule.
- The set of targets and prerequisites form a chain or graph of *dependencies* (short for "*dependency graph*").
- Building and processing this dependency graph to update the requested target is what `make` is all about.



Makefile: Rules

1. Explicit Rules

- Most rules you will write are explicit rules that specify particular files as targets and prerequisites.
- A rule can have more than one target. This means that each target has the same set of prerequisites as the others.

Makefile: Rules

1. Explicit Rules

```
vpath.o variable.o: make.h config.h getopt.h
```

≡

```
vpath.o: make.h config.h getopt.h
variable.o: make.h config.h getopt.h
```

Makefile: Rules

2. Explicit Pattern Rules

- `Makefiles` can become cumbersome and difficult to maintain for large programs with numerous files.
- Specifying targets, prerequisites, and command scripts for each file becomes impractical, leading to duplicate code and potential bugs.
- This poses a significant maintenance challenge and a potential source of errors.

Makefile: Rules

2. Explicit Pattern Rules

- Many programs that read one file type and output another conform to standard conventions.
- For instance, all C compilers assume that files that have a .c suffix contain C source code and that the object filename can be derived by replacing the .c suffix with .o

Makefile: Rules

2. Explicit Pattern Rules

```
foobar.o: foobar.c foobar.h
    $(CC) $(CFLAGS) $(CPPFLAGS) -c foobar.c

foo.o: foo.c foo.h
    $(CC) $(CFLAGS) $(CPPFLAGS) -c foo.c

bar.o: bar.c bar.h
    $(CC) $(CFLAGS) $(CPPFLAGS) -c bar.c
```

Makefile: Rules

2. Explicit Pattern Rules

```
%.o: %.c %.h
	$(CC) $(CFLAGS) $(CPPFLAGS) -c $<
```

Makefile: Rules

2. Implicit Pattern Rules

- `GNU make 3.8` has about 90 built-in implicit rules.
- An implicit rule is either a pattern rule or a suffix rule.
- There are built-in pattern rules for *C, C++, Pascal, FORTRAN, ratfor, Modula, Texinfo, TeX, Emacs Lisp, RCS, and SCCS.*

Makefile: Rules

Lab 3

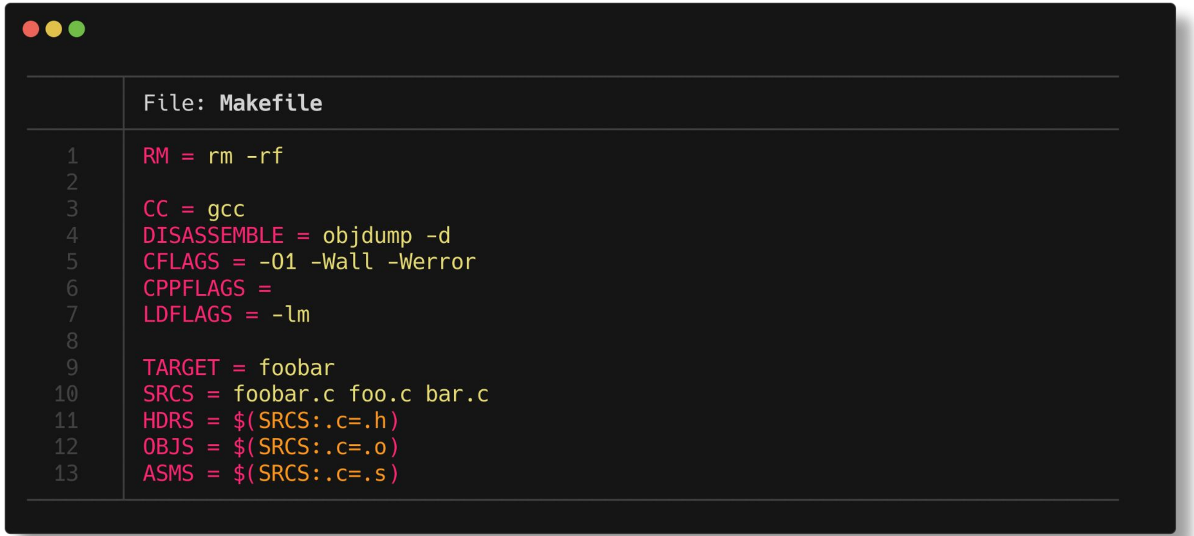
Create a *`Makefile`* that generates a disassembly *`filename.s`* after compiling each source file and *`target.s`* after linking the executable.

Notes:

- Don’t use *`gcc -S`* flag, use *`objdump`* instead
- Search *`objdump`* or *`man objdump`* for more info
- Use implicit rules
- Use automatic variables

Makefile: Rules

Lab 3

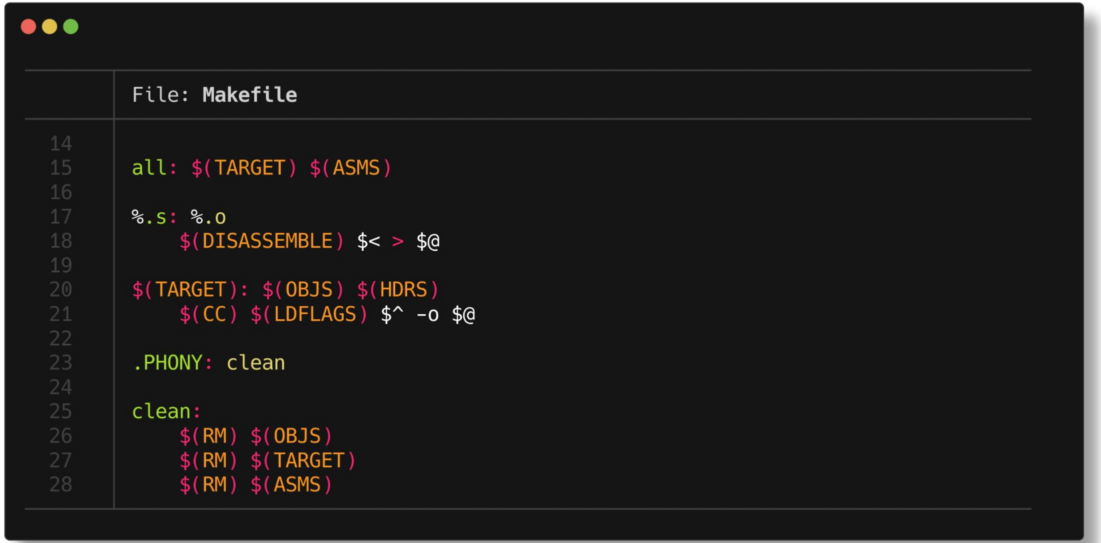


File: **Makefile**

```
RM = rm -rf

CC = gcc
DISASSEMBLE = objdump -d
CFLAGS = -O1 -Wall -Werror
CPPFLAGS =
LDFLAGS = -lm

TARGET = foobar
SRCS = foobar.c foo.c bar.c
HDRS = $(SRCS:.c=.h)
OBJS = $(SRCS:.c=.o)
ASMS = $(SRCS:.c=.s)
```

Makefile: Rules

Lab 3



File: **Makefile**

```
all: $(TARGET) $(ASMS)

%.s: %.o
	$(DISASSEMBLE) $< > $@

$(TARGET): $(OBJS) $(HDRS)
	$(CC) $(LDFLAGS) $^ -o $@

.PHONY: clean

clean:
	$(RM) $(OBJS)
	$(RM) $(TARGET)
	$(RM) $(ASMS)
```

Makefile: Functions

- `GNU make` supports both built-in and user-defined functions.
- A function invocation looks much like a variable reference, but includes one or more parameters separated by commas.
- Most built-in functions expand to some value that is then assigned to a variable or passed to a subshell.

Resources

- [GNU Make Manual]
- [Managing Projects with GNU Make, 3rd Edition by Robert Mecklenburg]