Λειτουργικά Συστήματα

Χειμερινό Εξάμηνο 2023-24

«Μεταγλώττιση και Makefiles»

Χρήστος Μακρής
Σπύρος Σιούτας
Παναγιώτης Χατζηδούκας
Αριστείδης Ηλίας

Compilation

- It is important to understand how programs are compiled to have a better understanding of how different parts of a computer interact with each other.
- Fundamental aspect of how computers run code.

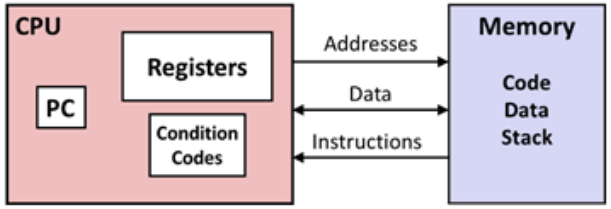
Levels of abstraction

- C [and other high-level languages] are easy for programmers to understand, but computers require lots of software to process them
- Machine code is just the opposite: easy for the computer to process, but humans need lots of help to understand it
- Assembly language is a compromise between the two: readable by humans (barely), close correspondence to machine code

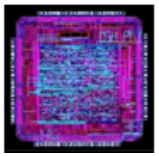
C programmer

```
#include <stdio.h>
int main(){
  int i, n = 10, t1 = 0, t2 = 1, nxt;
  for (i = 1; i <= n; ++i){
    printf("%d, ", t1);
    nxt = t1 + t2;
    t1 = t2;
    t2 = nxt; }
  return 0; }
```

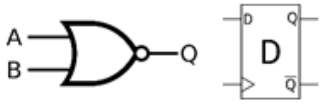
Assembly programmer



Computer designer

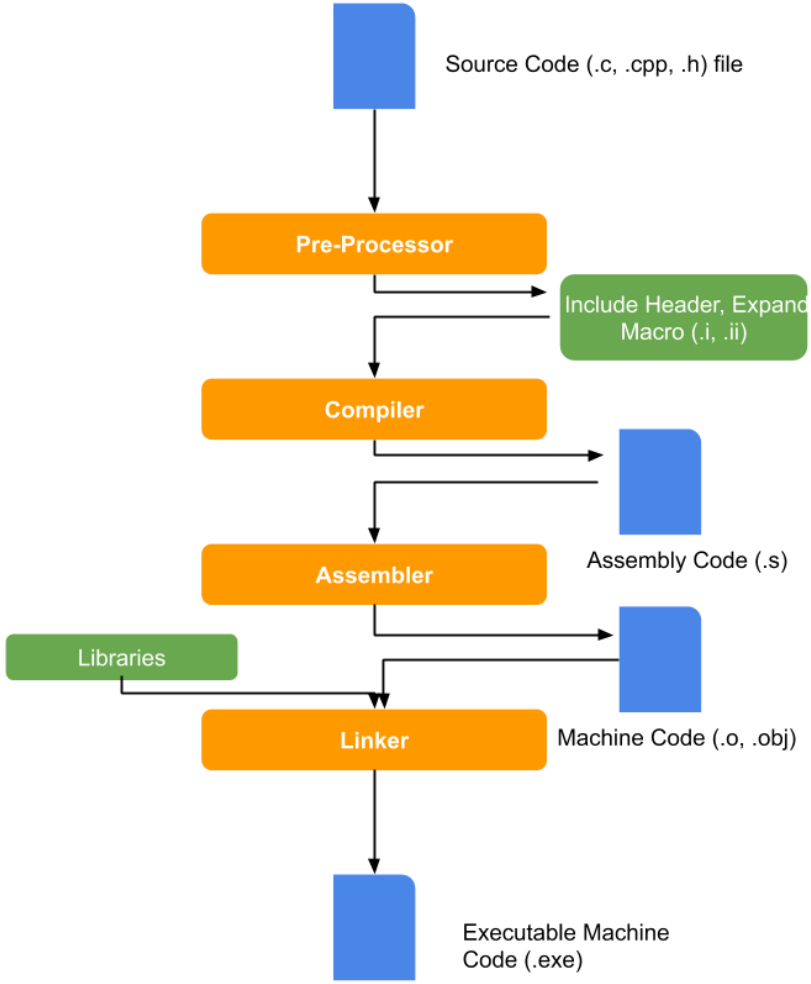


Gates, clocks, circuit layout, …



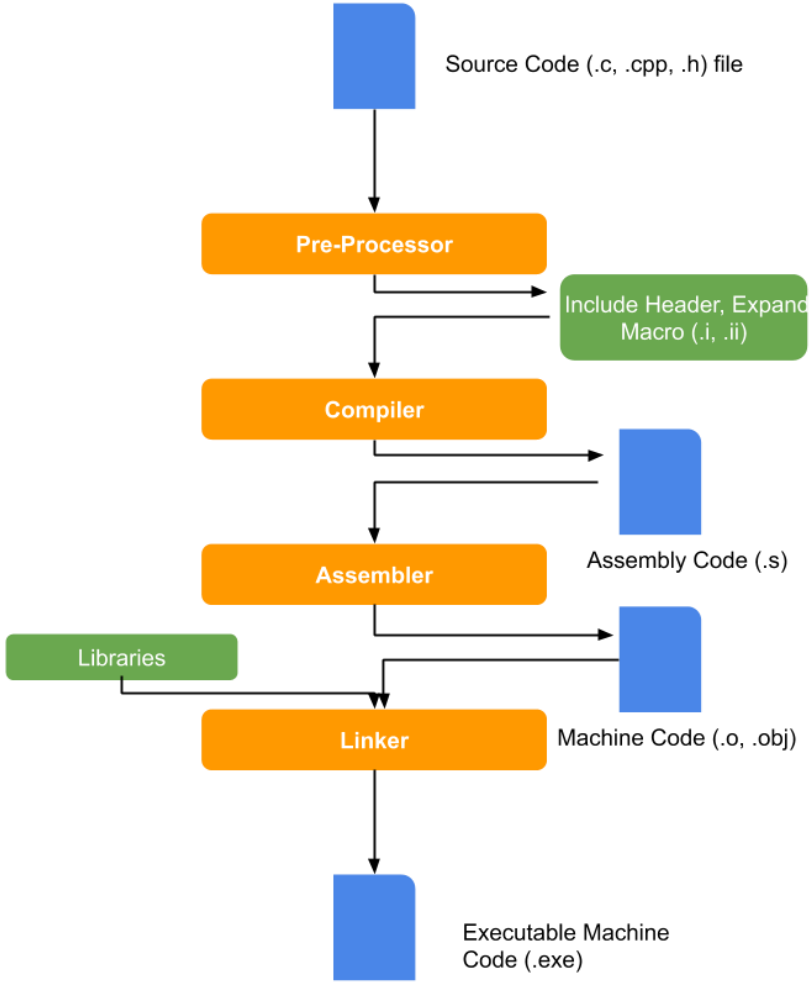
Code compilation

- The computer only understands machine code directly
- All other languages must be either
  - interpreted: executed by software
  - compiled: translated to machine code by software



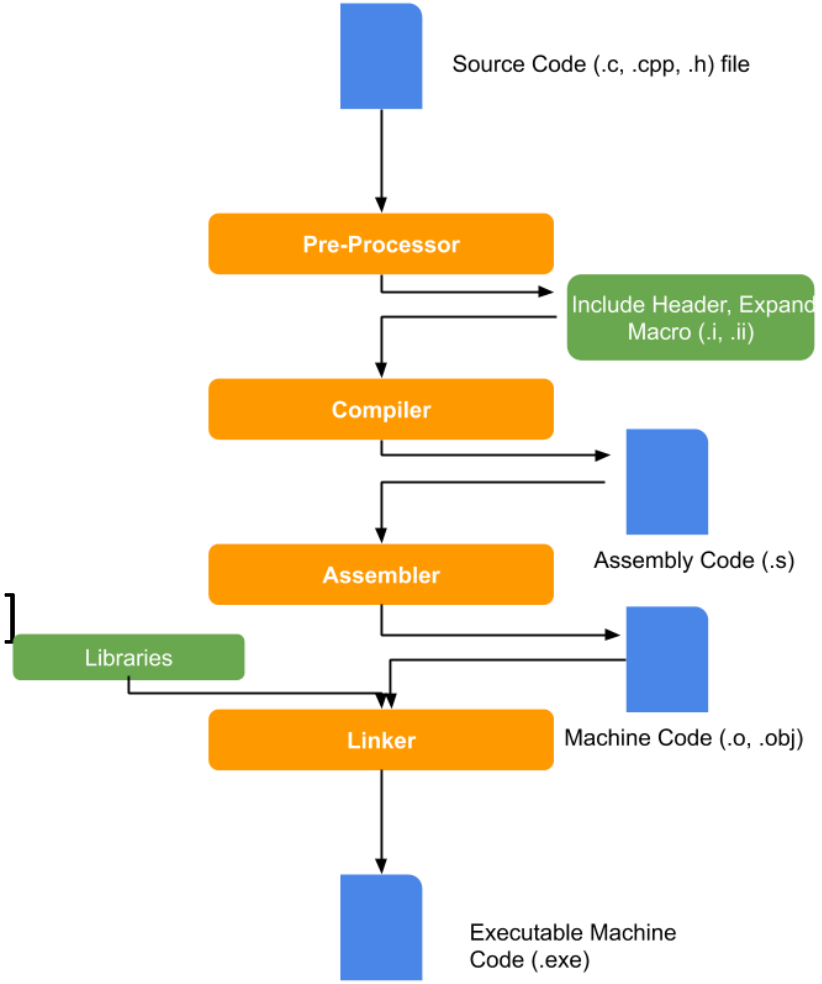
Code compilation

- Computer follows steps to translate your code into something the computer can understand
- This is the process of compiling code [a compiler completes these actions]
- Four steps for C: preprocessing, compiling, assembling, linking
  - Most other compiled languages don't have the preprocessing step, but do have the other three



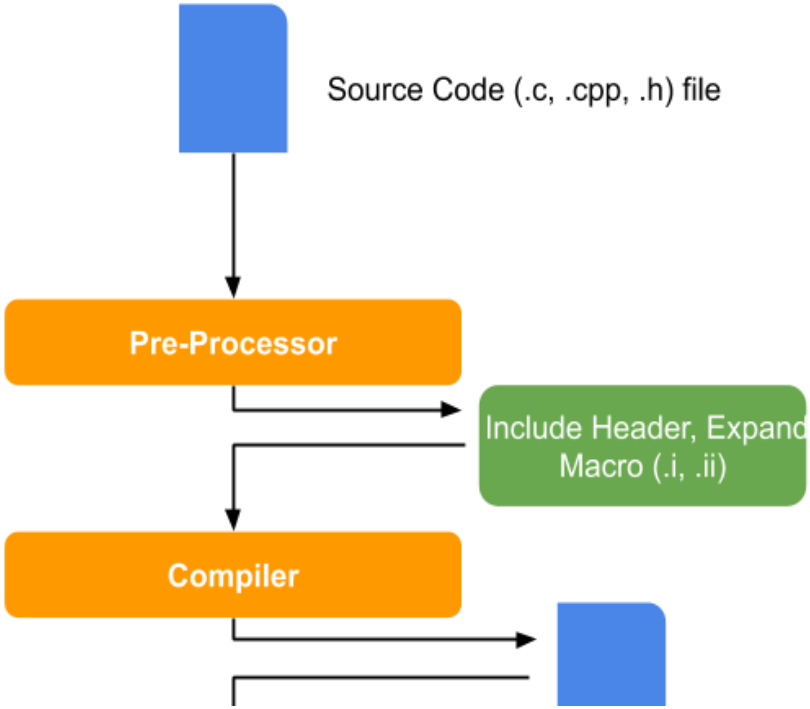
Compilation stages

- Pre-Processor
  $ gcc -E [flags] [filenames]
- Compiler
  $ gcc -S [flags] [filenames]
- Assembler
  $ gcc -c [flags] [filenames]
  $ objdump -d [filenames]
- Linker
  $ gcc -o [exename] [flags] [filenames]



Pre-Processor

- Unique to the C family; other languages don’t have this
- Processes #include, #define, #if, macros
  - Combines main source file with headers (textually)
  - Defines and expands macros (token-based shorthand)
  - Conditionally removes parts of the code (e.g. specialize for Linux, Mac, …)
- Removes all comments
- Output looks like C



Before and after preprocessing

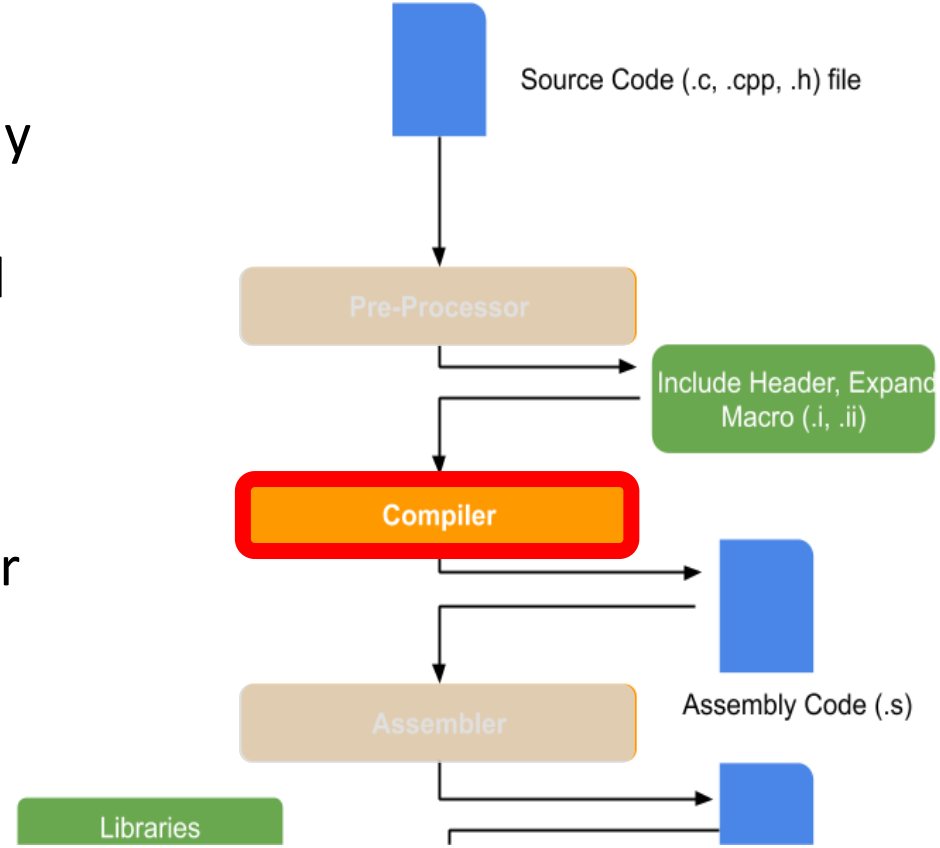
```c
#include <limits.h>
#include <stdio.h>

int main(void) {

    printf("CHAR_MIN = %d\n"
           "CHAR_MAX = %d\n",
           CHAR_MIN, CHAR_MAX);
    return 0;
}
```

```c
# 1 "test.c"
# 1 "/usr/lib/gcc/x86_64-linux-gnu/10/include/limits.h" 1 3 4

# 1 "/usr/include/stdio.h" 1 3 4

extern int fprintf (FILE *__restrict __stream,
         const char *__restrict __format, ...);
extern int printf (const char *__restrict __format, ...);

# 874 "/usr/include/stdio.h" 3 4
# 3 "test.c" 2

int main(void) {
    printf("CHAR_MIN = %d\n"
           "CHAR_MAX = %d\n",
# 6 "test.c" 3 4
           (-0x7f - 1)
# 6 "test.c"
                   , 0x7f);
    return 0;
}
```

- Contents of header files inserted inline
- Comments removed
- Macros expanded
- “Directive” lines (beginning with #) communicate things like original line numbers

Compiler

- The compiler translates the preprocessed code into assembly code
  - This changes the format and structure of the code but preserves the semantics (what it does)
  - Can change lots of details for optimization, as long as the overall effect is the same



Before and after compilation

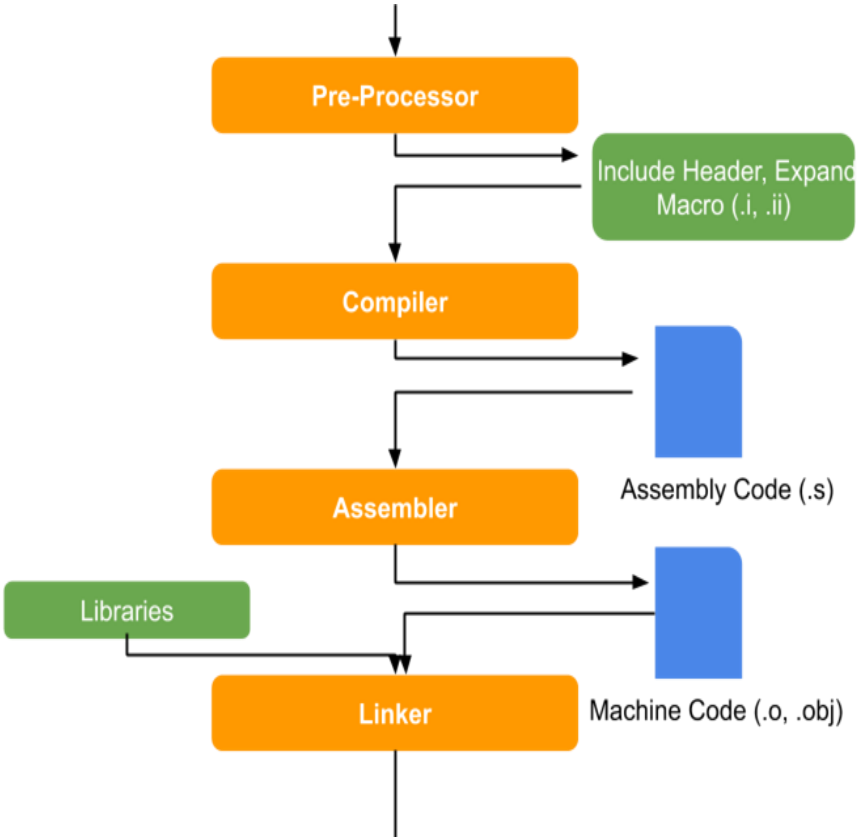
```
extern int printf (const char *__restrict
                   __format, ...);
int main(void) {
    printf("CHAR_MIN = %d\n"
           "CHAR_MAX = %d\n",
           (-0x7f - 1), 0x7f);
    return 0;
}
```

```
    .file   "test.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string    "CHAR_MIN = %d\nCHAR_MAX = %d\n"
    .text
    .globl    main
main:
    subq    $8, %rsp
    movl    $127, %edx
    movl    $-128, %esi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    call    printf@PLT
    xorl    %eax, %eax
    addq    $8, %rsp
    ret
    .size    main, .-main
```

- C *source code* converted to *assembly language*
- Textual, but 1:1 correspondence to machine language
- printf just referred to, not declared

Assembler

- Parses assembly code and mainly translates into bits
- There is some flexibility to generate the most efficient version of machine code, but mostly responsible for just translating to bits.



Before and after assembling

```
    .file    "test.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string    "CHAR_MIN = %d\nCHAR_MAX = %d\n"
    .text
    .globl    main
main:
    subq    $8, %rsp
    movl    $127, %edx
    movl    $-128, %esi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    call    printf@PLT
    xorl    %eax, %eax
    addq    $8, %rsp
    ret
    .size    main, .-main
```

```
$ objdump -s -r test.o
test.o:  file format elf64-x86-64

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000011 R_X86_64_PC32     .LC0-0x0000000000000004
0000000000000018 R_X86_64_PLT32    printf-0x0000000000000004

Contents of section .rodata.str1.1:
 0000 43484152 5f4d494e 203d2025 640a4348  CHAR_MIN = %d.CH
 0010 41525f4d 4158203d 2025640a 00        AR_MAX = %d..

Contents of section .text:
 0000 4883ec08 ba7f0000 00be80ff ffff488d  H.............H.
 0010 3d000000 0031c0e8 00000000 31c04883  =....1......1.H.
 0020 c408c3                               ...
```

- Everything is now binary

Before and after assembling

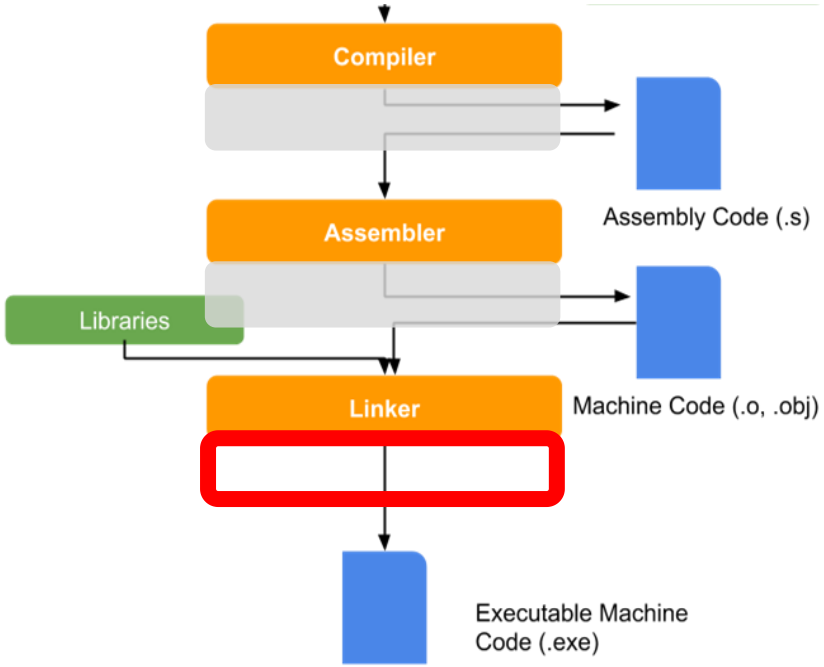
```
    .file    "test.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string    "CHAR_MIN = %d\nCHAR_MAX = %d\n"
    .text
    .globl    main
main:
    subq    $8, %rsp
    movl    $127, %edx
    movl    $-128, %esi
    leaq    .LC0(%rip), %rdi
    xorl    %eax, %eax
    call    printf@PLT
    xorl    %eax, %eax
    addq    $8, %rsp
    ret
    .size    main, .-main
```

```
$ objdump -d -r test.o
test.o:  file format elf64-x86-64
Disassembly of section .text.startup:

0000000000000000 <main>:
   0:    48 83 ec 08            sub $0x8,%rsp
   4:    ba 7f 00 00 00         mov $0x7f,%edx
   9:    be 80 ff ff ff         mov $0xffffff80,%esi
   e:    48 8d 3d 00 00 00 00   lea 0x0(%rip),%rdi
  11: R_X86_64_PC32 .LC0-0x4
  15:    31 c0                  xor %eax,%eax
  17:    e8 00 00 00 00         call   1c <main+0x1c>
  18: R_X86_64_PLT32    printf-0x4
  1c:    31 c0                  xor %eax,%eax
  1e:    48 83 c4 08            add $0x8,%rsp
  22:    c3                     ret
```

- Just to emphasize that 1:1 correspondence between assembly and machine instructions

Linker

- For static libraries
- Aggregates multiple independently compiled files containing machine code
- Fills in those unknown addresses
- The goal is to create 1 file with all of the needed code to run the program
  - This is the file you run to check your code



GCC

- GNU Compiler Collection
  - GCC is a set of compilers for various languages. It provides all of the infrastructure for building software in those languages from source code to assembly.
- The compiler can handle compiling everything on its own, but you can use various flags to breakdown the compilation steps
- Example:

  gcc [flags] [infile(s)]

GCC Common Flags

-o [EXECUTABLE NAME] : names executable file

-O*x* : Code optimization

-O0 : Compile as fast as possible, don't optimize [this is the default]

-O1, **-O2**, **-O3**: Optimize for reduced execution time [higher numbers are more optimized]

-Os : Optimize for code size instead of execution time.

-Og : Optimize for execution time but try to avoid making interactive debugging harder.

-g : produce "debug info": annotate assembly so gdb can find variables and source code

-Wall : enable many "warning" messages that *should* be on by default, but aren't

- Does *not* turn on all of the warning messages GCC can produce.
- See https://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Warning-Options.html

-Werror : turns all warnings into errors

-std=c99 : use the 1999 version of the C standard and disable some (not all!) extensions

Makefile

- Automates the process of creating files (using a compiler)
- For example, create **bomb** from bomb.c, phases.c, and util.c
- Running *make bomb* will update *bomb*
  - *Only* if any of the source files have changed;
    avoids unnecessary work
  - Remembers complicated compiler commands for you
- Can also store recipes for automating development tasks
  - *make format* to reformat source files

Makefile

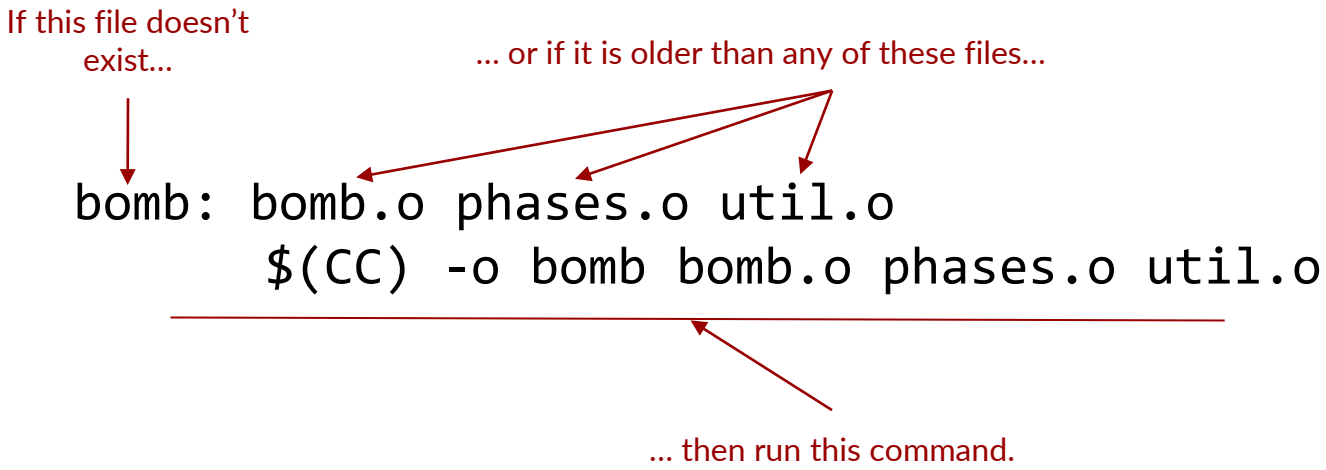
Makefiles are lists of rules

- There are two kinds of rules: **normal** and **phony**
  - Normal rules create files
  - Phony rules don't directly create files
- Each rule has a **target**.
  - For **normal** rules, the target is the name of the file that the rule will create
  - For **phony** rules, the target is an arbitrary name for what the rule does
- Rules may have **prerequisites** (also known as **dependencies**)
  - Prerequisites are the files that are needed to create the target
  - If any of the prerequisites doesn't exist, it must be created first
  - If any of the prerequisites is newer than the target, the target is "out of date" and must be re-created
- Rules may have **commands**.
  - One or more shell commands that create the target from its prerequisites
  - For phony rules, just some commands to be run

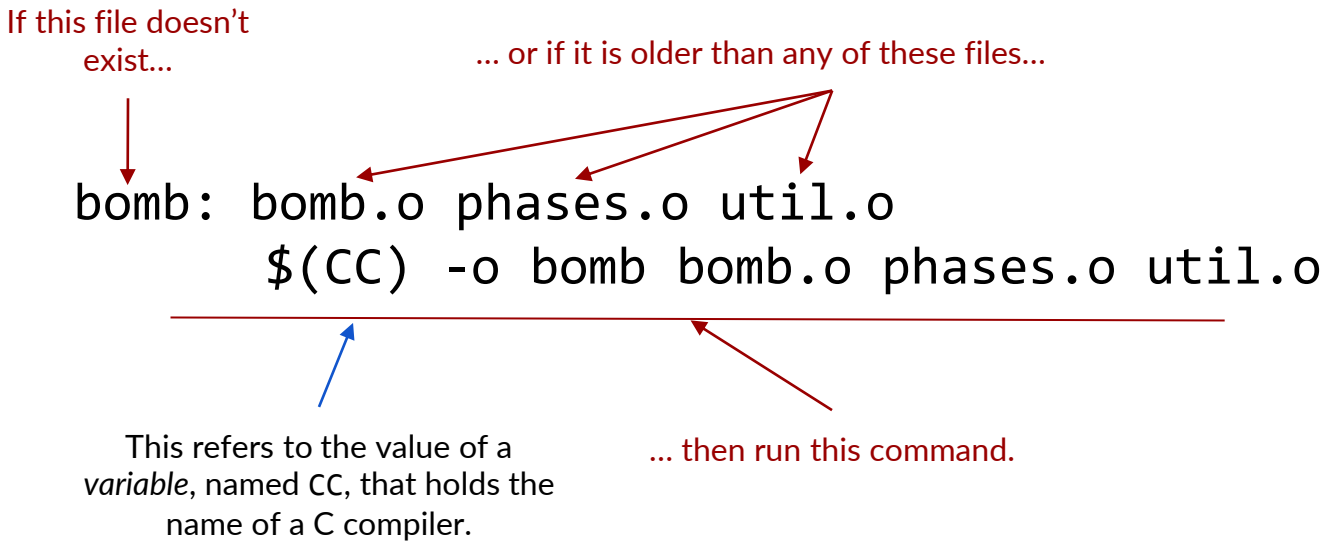
Normal rule example

```
bomb: bomb.o phases.o util.o
	$(CC) -o bomb bomb.o phases.o util.o
```

Normal rule example



Normal rule example



Normal rule without prerequisites

```
output_dir:
	mkdir output_dir
```

- Run mkdir output_dir if output_dir does not exist
- If it does exist, no action

Normal rule without commands

```
bomb.o: bomb.c support.h phases.h
```

- Re-create bomb.o if any of bomb.c, support.h, phases.h is newer
- The commands to do this are given somewhere else
  - A *pattern rule* elsewhere in the Makefile
  - An *implicit rule* built into Make

Pattern and implicit rules

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

- To create an .o file from a .c file with the same base name, use this command
- Special variables $@ and $< give the name of the .o and .c files respectively
- Variables CC and CFLAGS can be set to customize behavior

- This rule is *implicit* — built into Make — you don't have to write it yourself

Phony rule example

```
all: bomb bomb-solve
.PHONY: all
```

- When asked to create “all”, create bomb and bomb-solve
- Does **not** create a file named “all”
- The .PHONY annotation can be anywhere in the makefile

Phony rule example 2

```
clean:
	rm -f bomb bomb-solve *.o
.PHONY: clean
```

- When asked to create “clean”, run this command
  - Which deletes bomb, bomb-solve, and all object files
- Does **not** create a file named “clean”

The make command

- Running **make** in the shell will cause the shell to look for a Makefile in the current directory. If it finds one, it will attempt to create the first target listed in the Makefile.
- You can also run **make** <target_name> to indicate exactly which target you want to create.
- By convention, the first target is a phony target named all
  - so make and make all do the same thing
  - as the name implies, this is to create everything that the makefile knows how to create
- Phony rules serve as entry points into the Makefile
  - make all creates everything, make clean deletes all generated files, make check runs tests, …
  - But you can also make bomb.o if that's the only thing you want

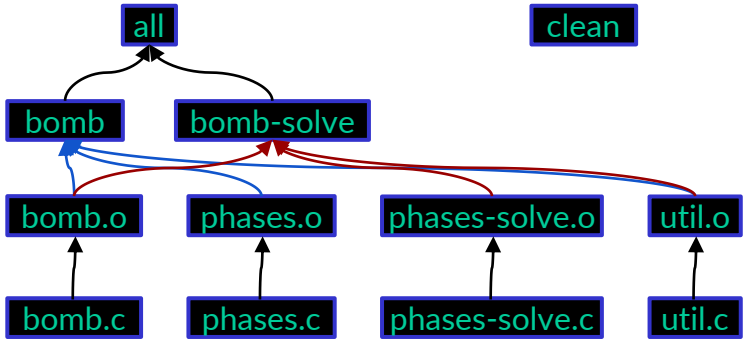
Makefile

```
CC = gcc
CFLAGS = -std=c99 -g -O2 -Wall -Werror

all: bomb bomb-solve
bomb: bomb.o phases.o util.o
    $(CC) $(LDFLAGS) -o $@ $^ $(LIBS)

bomb-solve: bomb.o phases-solve.o util.o
    $(CC) $(LDFLAGS) -o $@ $^ $(LIBS)

bomb.o: bomb.c phases.h support.h
phases.o: phases.c phases.h support.h
phases-solve.o: phases-solve.c phases.h support.h
util.o: util.c support.h

clean:
    rm -f bomb bomb-solve *.o

.PHONY: all clean
```

- OK to use undefined variables
  - LDFLAGS, LIBS
  - Found in environment or treated as empty
- Don't need to give commands to create object files from C source
  - But do need to list header file dependencies for each object file
- Do need to give commands to create executables (missing feature)
- all rule at the top, clean rule at the bottom
- One .PHONY annotation for all phony rules

Rules form a graph

```
CC = gcc
CFLAGS = -std=c99 -g -O2 -Wall -Werror

all: bomb bomb-solve
bomb: bomb.o phases.o util.o
    $(CC) $(LDFLAGS) -o $@ $^ $(LIBS)

bomb-solve: bomb.o phases-solve.o util.o
    $(CC) $(LDFLAGS) -o $@ $^ $(LIBS)

bomb.o: bomb.c phases.h support.h
phases.o: phases.c phases.h support.h
phases-solve.o: phases-solve.c phases.h support.h
util.o: util.c support.h

clean:
    rm -f bomb bomb-solve *.o

.PHONY: all clean
```



Make avoids unnecessary work

- If *bomb.c* changes, *make all* will re-create bomb.o, bomb, bomb-solve
- If *phases.c* changes, *make all* will only re-create *phases.o* and *bomb*

Make can see through missing targets

- If *bomb.o* does not exist, *make bomb* creates it from *bomb.c*