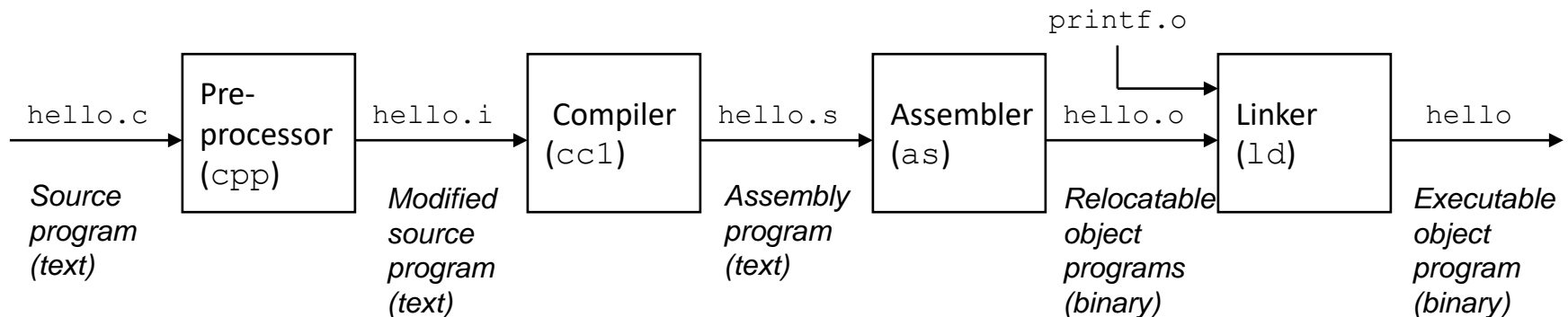


Building Large Programs



Topics

- ❖ Multi-file C programs, modularity
- ❖ Make
- ❖ More on C Preprocessor
- ❖ More on Linker

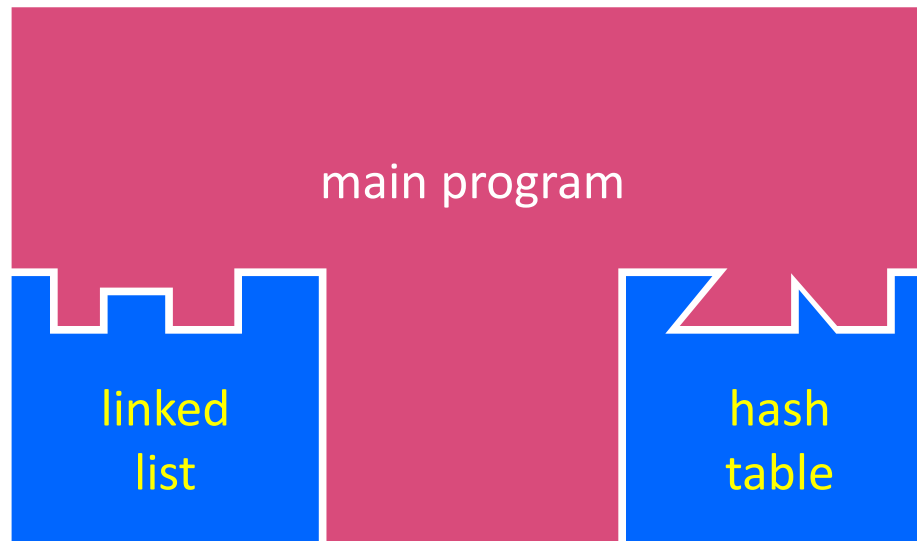


Dealing with big programs

- ❖ When dealing with large programs, we need to divide programs into modules.
- ❖ We may need to create our own libraries.
- ❖ There are many advantages to this approach:
 - The modules will naturally divide into common groups of functions.
 - We can compile each module separately and link in compiled modules (more on this later).
 - UNIX utilities such as **make** help us maintain large systems.

Modularity

- ❖ The degree to which components of a system can be separated and recombined
 - Modules can be developed independently
 - Modules can be re-used in different projects



C Module Conventions

- ❖ Most C projects adhere to the following rules:
 - `.h` files only contain *declarations*, never *definitions*
 - `.c` files never contain prototype declarations for functions that are intended to be exported through the module interface
 - Those function prototype declarations belong in the `.h` file
 - **NEVER** `#include .c` file – only `#include .h` files
 - `#include` all of headers you reference, even if another header (accidentally or not) includes some of them
 - Any `.c` file with an associated `.h` file should be able to be compiled into a `.o` file
 - The `.c` file should `#include` the `.h` file; the compiler will check definitions and declarations for consistency

Where Do the Comments Go?





- ❖ If a function is declared in a header file (.h) and defined in a C file (.c):
 - *The header needs full documentation because it is the public specification*
 - No need to copy/paste the comment into the .c file
 - Don't want two copies that can get out of sync
 - Recommended to leave “specified in <filename>.h” comment in .c file code to help the reader



Topics

- ❖ Multi-file C programs, modularity
- ❖ **Make**
- ❖ More on C Preprocessor
- ❖ More on Linker

Building Software

- ❖ Programmers spend a lot of time “building”
 - Creating programs from source code
 - Both programs that they write and other people write
- ❖ Programmers like to automate repetitive tasks
 - Repetitive: `gcc -Wall -g -std=c11 -o widget foo.c bar.c baz.c`
 - Retype this every time: 
 - Use up-arrow or history:  (still retype after logout)
 - Have an alias or bash script: 
 - Have a Makefile:  (you're ahead of us)

“Real” Build Process

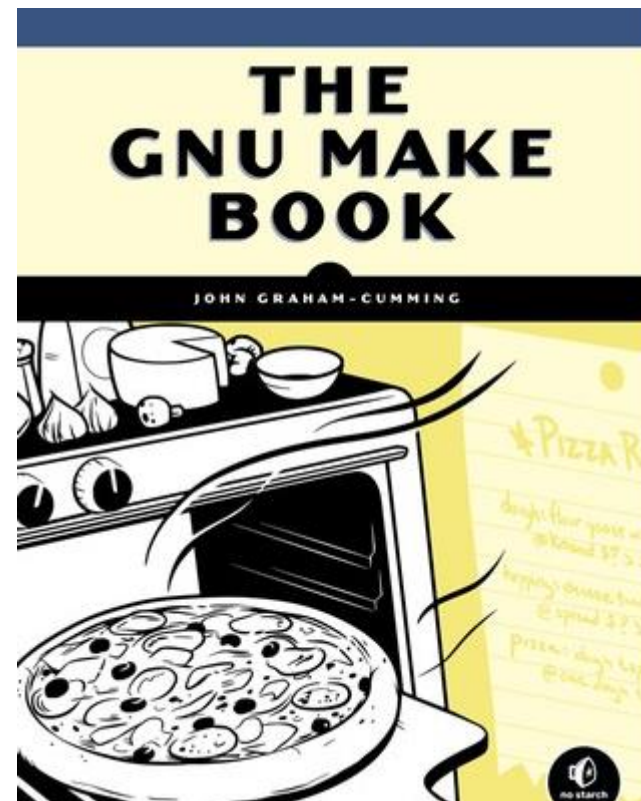
- ❖ On larger projects, you can't or don't want to have one big (set of) command(s) that redoes everything every time you change anything:
 - 1) If `gcc` didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
 - 2) You don't want to have to document the build logic when you distribute source code
 - 3) You don't want to recompile everything every time you change something (especially if you have 10^5 - 10^7 files of source code)

Make

■ <https://www.gnu.org/software/make/manual/make.html#toc-An-Introduction-to-Makefiles>

■ The **make** utility is a utility for building complex systems/program

1. Enables the end user to build and install your package without knowing the details of how that is done
2. Figure out automatically which parts of system are out of date
3. Issue command to create the intermediate and final project files
4. Not limited to any particular language.



Make basics

- Each system you want to build has one (or more) files defining how to build, called the “**Makefile**”

- ❖ A Makefile defines the things to build ...
- ❖ ... the way to build them
- ❖ ... and the way they relate

- Terminology

- **Target** - a thing to build
- **Prerequisites** - things that the target depends on
- **Dependencies** – dependency relationship between files, e.g., *a.o* depends on *a.c*
- **Variables** - data that defines elements of interest to the build
- **Rules** - are statements of targets, prerequisites and commands




for GNU Make Version 3.79.1
by Richard M. Stallman and Roland McGrath
Copyrighted Material



Makefile rules...

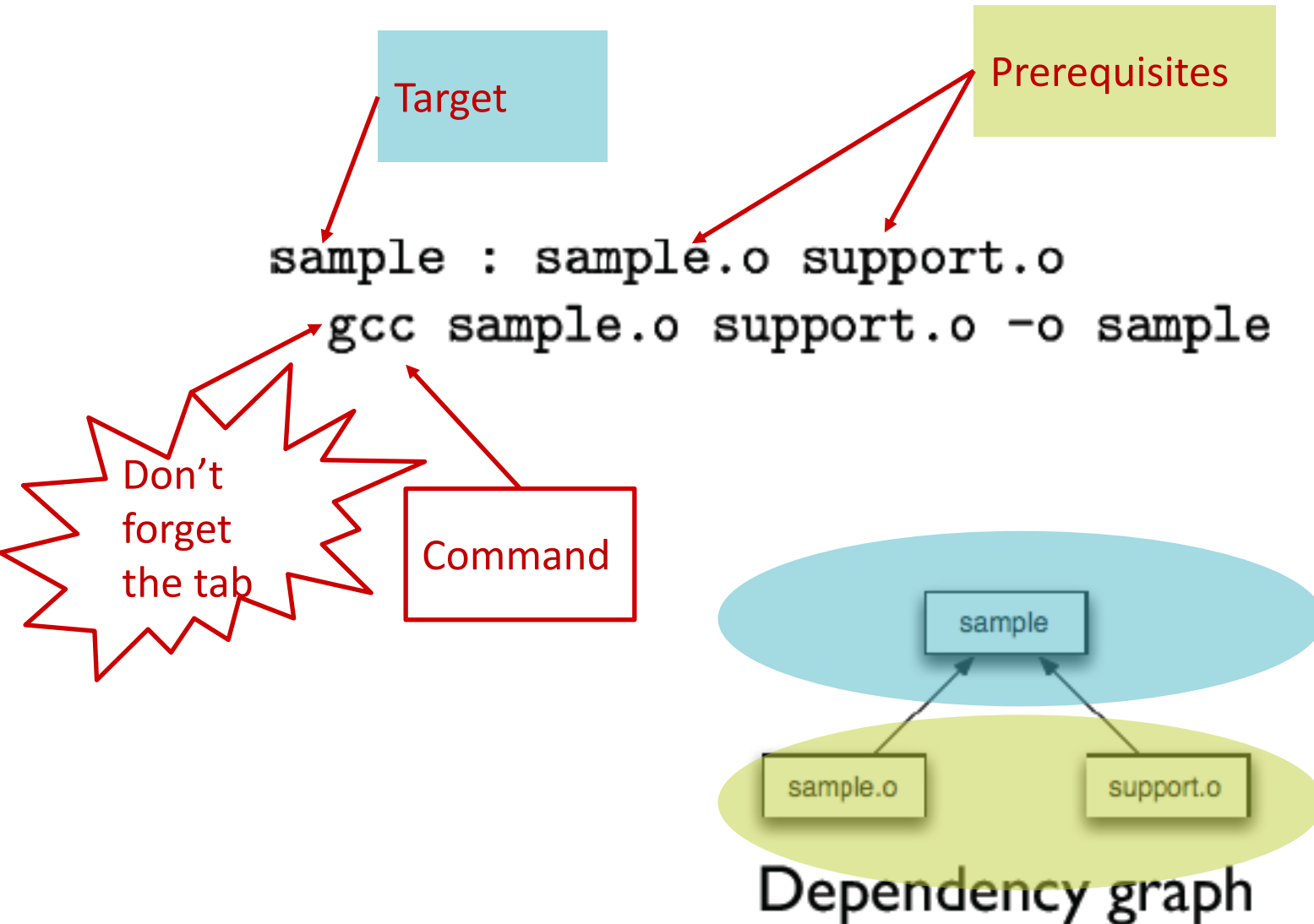
- Also known as a *production*, a rule defines how a particular item is built. The rule syntax is:

```
target: prereq1, prereq2, prereq3, ...  
     command1  
(MUST BE TABBED OVER)
```

- Where
 - target is thing to be built
 - prereq(1|2|3) are things needed to make the target
 - commands are the UNIX commands to run to make target
- *Key Idea: run the (commands) to build (the target) when (any of the prerequisites are out of date)*



Rule example



What about the object files?

Linking

```
sample : sample.o support.o  
gcc sample.o support.o -o sample
```

Compiling

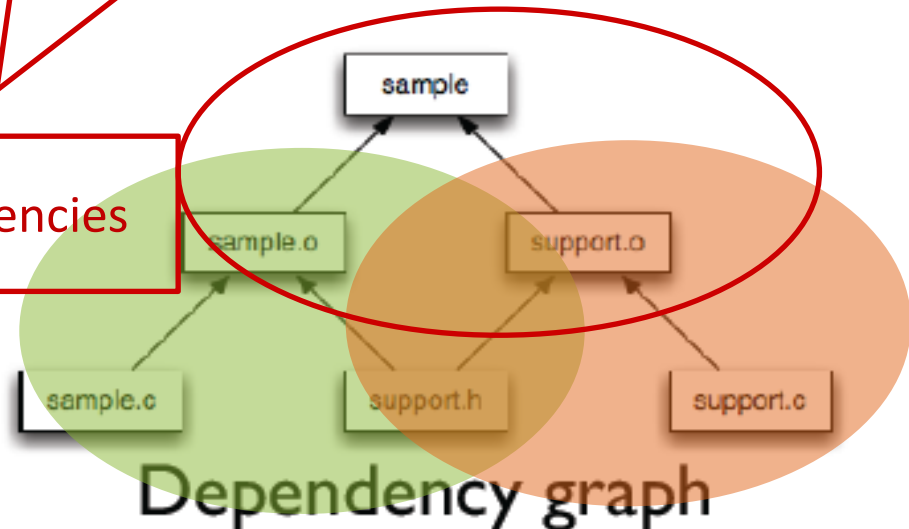
```
sample.o : sample.c support.h  
gcc -c -Wall -I. sample.c -o sample.o
```

Compiling

```
support.o : support.c support.h  
gcc -c -Wall -I. support.c -o support.o
```

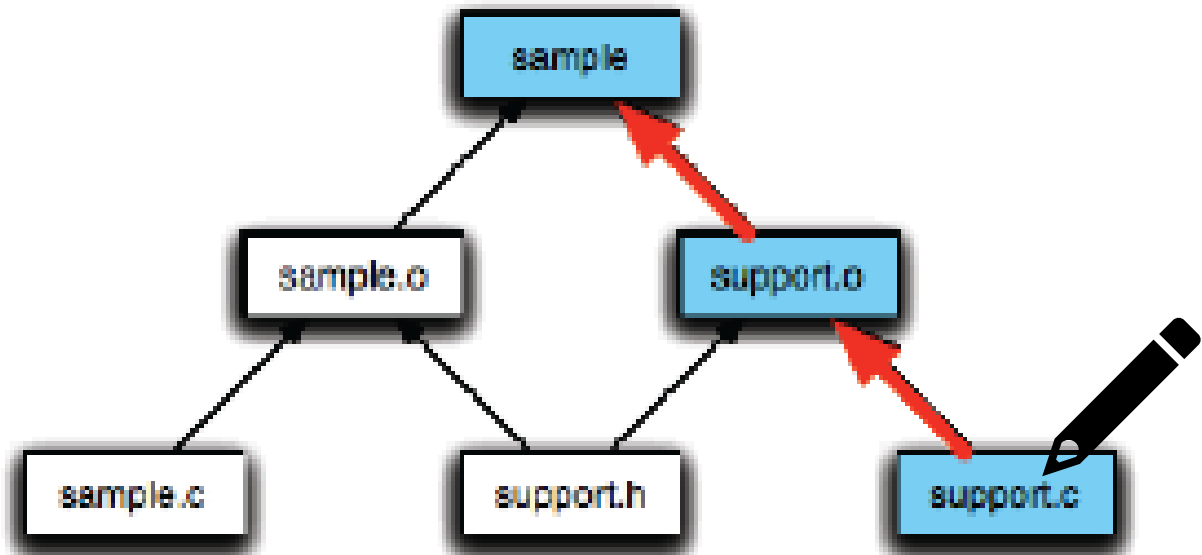
Target

Dependencies



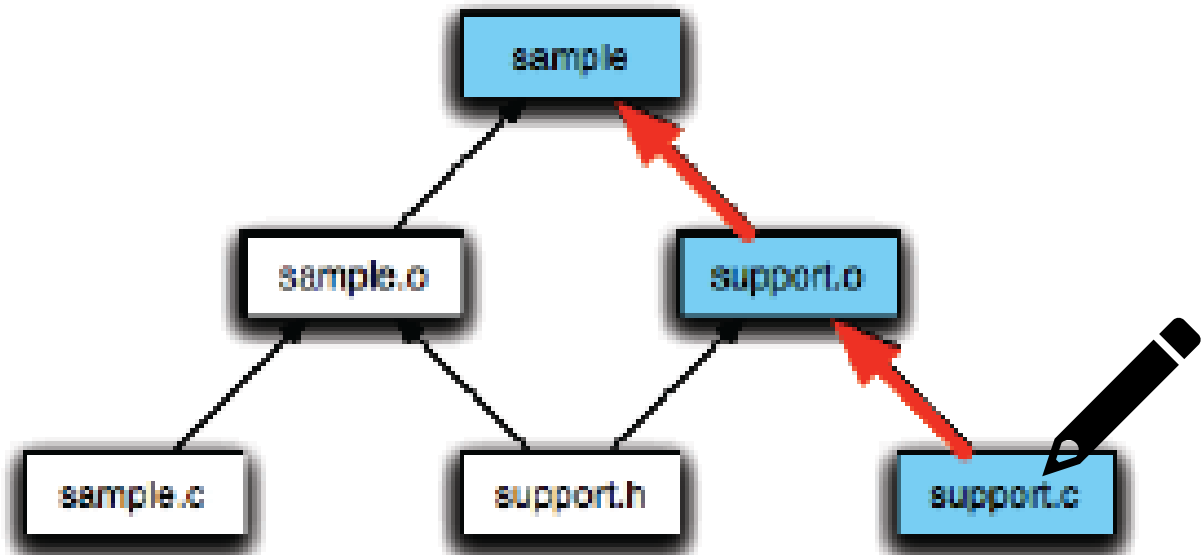
Running make

- To run make, just type **make** at the UNIX prompt.
 - It will open and interpret the **Makefile** (or makefile) and build any targets that are out of date
 - Thus ... it will look at the dependency graph
 - E.g., consider if I edit support.c ...



Running make

- To run make, just type **make** at the UNIX prompt.
 - It will open and interpret the **Makefile** (or makefile) and build any targets that are out of date
 - Thus ... it will look at the dependency graph
 - E.g., consider if I edit support.c ...



Variables

- Makefile variables allow you to replace some repetitive (and changing) text with others

```
OBJECT_FILES= sample.o support.o
```

```
sample : $(OBJECT_FILES)  
    gcc $(OBJECT_FILES) -o sample
```

- Some standard variables include:

```
CC=gcc  
LINK=gcc  
CFLAGS=-c -Wall -I.
```



Automatic variables


- Make supports a range of “special” variables that are used while evaluating each rule (called built-ins or Automatic variables)
- Three of the most popular built-ins are
 - “\$@” is the current rule target
 - “\$^” is the prerequisite list
 - “\$<” is the first prerequisite
- Built-ins are used to make builds cleaner ...

```
OBJECT_FILES= sample.o support.o      sample : $(OBJECT_FILES)
                                       $(CC) $^ -o $@
sample : $(OBJECT_FILES)
  gcc $(OBJECT_FILES) -o sample

sample.o : sample.c support.h
  $(CC) $(FLAGS) sample.c -o $@

support.o : support.c support.h
  $(CC) $(FLAGS) support.c -o $@

CC=gcc
LINK=gcc
CFLAGS=-c -Wall -I.
```



Understanding our project1 Makefile

```
CC=gcc
CFLAGS=-c -g -Wall
LINKARGS=-lm

# Files
OBJECT_FILES=  p1.o p1-support.o

# Productions
all : cmisc257-s20-p1

cmisc257-s20-p1 : $(OBJECT_FILES)
    $(CC) $(LINKARGS) $(OBJECT_FILES) -o $@

p1.o : cmisc257-s20-p1.c p1-support.h
    $(CC) $(CFLAGS) $< -o $@

p1-support.o : p1-support.c p1-support.h
    $(CC) $(CFLAGS) $< -o $@

clean :
    rm -v cmisc257-f20-p1 $(OBJECT_FILES)
```

-c means compile: will
create object file

-g means generate
debugging information

-Wall means generate all
warnings

Understanding our project1 Makefile

```
CC=gcc
CFLAGS=-c -g -Wall
LINKARGS=-lm

# Files
OBJECT_FILES=  p1.o p1-support.o

# Productions
all : cmisc257-s20-p1

cmisc257-s20-p1 : $(OBJECT_FILES)
    $(CC) $(LINKARGS) $(OBJECT_FILES) -o $@

p1.o : cmisc257-s20-p1.c p1-support.h
    $(CC) $(CFLAGS) $< -o $@

p1-support.o : p1-support.c p1-support.h
    $(CC) $(CFLAGS) $< -o $@

clean :
    rm -v cmisc257-f20-p1 $(OBJECT_FILES)
```

We reuse the list of
OBJECT_FILES

Understanding our project1 Makefile

```
CC=gcc
CFLAGS=-c -g -Wall
LINKARGS=-lm

# Files
OBJECT_FILES=    p1.o p1-support.o

# Productions
all: cmsc257-s20-p1

cmsc257-s20-p1: $(OBJECT_FILES)
    $(CC) $(LINKARGS) $(OBJECT_FILES) -o $@

p1.o: cmsc257-s20-p1.c p1-support.h
    $(CC) $(CFLAGS) $< -o $@

p1-support.o: p1-support.c p1-support.h
    $(CC) $(CFLAGS) $< -o $@

clean:
    rm -v cmsc257-f20-p1 $(OBJECT_FILES)
```

Targets

Understanding our project1 Makefile

```
CC=gcc
CFLAGS=-c -g -Wall
LINKARGS=-lm
```

```
# Files
```

```
OBJECT_FILES= p1.o p1-support.o
```

```
# Productions
```

```
all : cmesc257-s20-p1
```

```
cmesc257-s20-p1 : $(OBJECT_FILES)
                  $(CC) $(LINKARGS) $(OBJECT_FILES) -o $@
```

```
p1.o : cmesc257-s20-p1.c p1-support.h
       $(CC) $(CFLAGS) $< -o $@
```

```
p1-support.o : p1-support.c p1-support.h
               $(CC) $(CFLAGS) $< -o $@
```

```
clean :
        rm -v cmesc257-f20-p1 $(OBJECT_FILES)
```

Prereqs

Dependencies

Target

First

Dependency

Understanding our project1 Makefile

```
CC=gcc
CFLAGS=-c -g -Wall
LINKARGS=-lm

# Files
OBJECT_FILES=    p1.o p1-support.o

# Productions
all : ←cmsc257-s20-p1

cmsc257-s20-p1 : $(OBJECT_FILES)
    $(CC) $(LINKARGS) $(OBJECT_FILES) -o $@

p1.o : cmsc257-s20-p1.c p1-support.h
    $(CC) $(CFLAGS) $< -o $@

p1-support.o : p1-support.c p1-support.h
    $(CC) $(CFLAGS) $< -o $@

clean : ←
    rm -v cmsc257-f20-p1 $(OBJECT_FILES)
```

Phony target all: if you are building multiple programs, you can list all programs here and make will compile all of them

Rule for cleaning the directory. Better have **.PHONY : clean** before. This will prevent make confuse rule clean with file clean

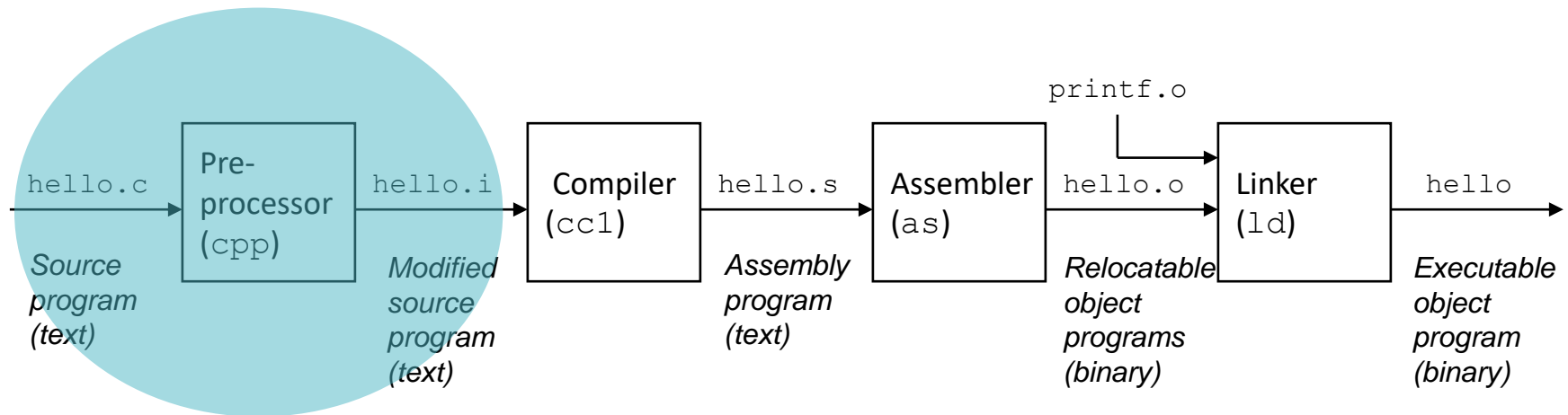
Since clean is not a prereq for cmsc257-f20-p1 it will not run automatically.

For more on Makefiles

[https://www.gnu.org/software/make/
manual/make.html](https://www.gnu.org/software/make/manual/make.html)

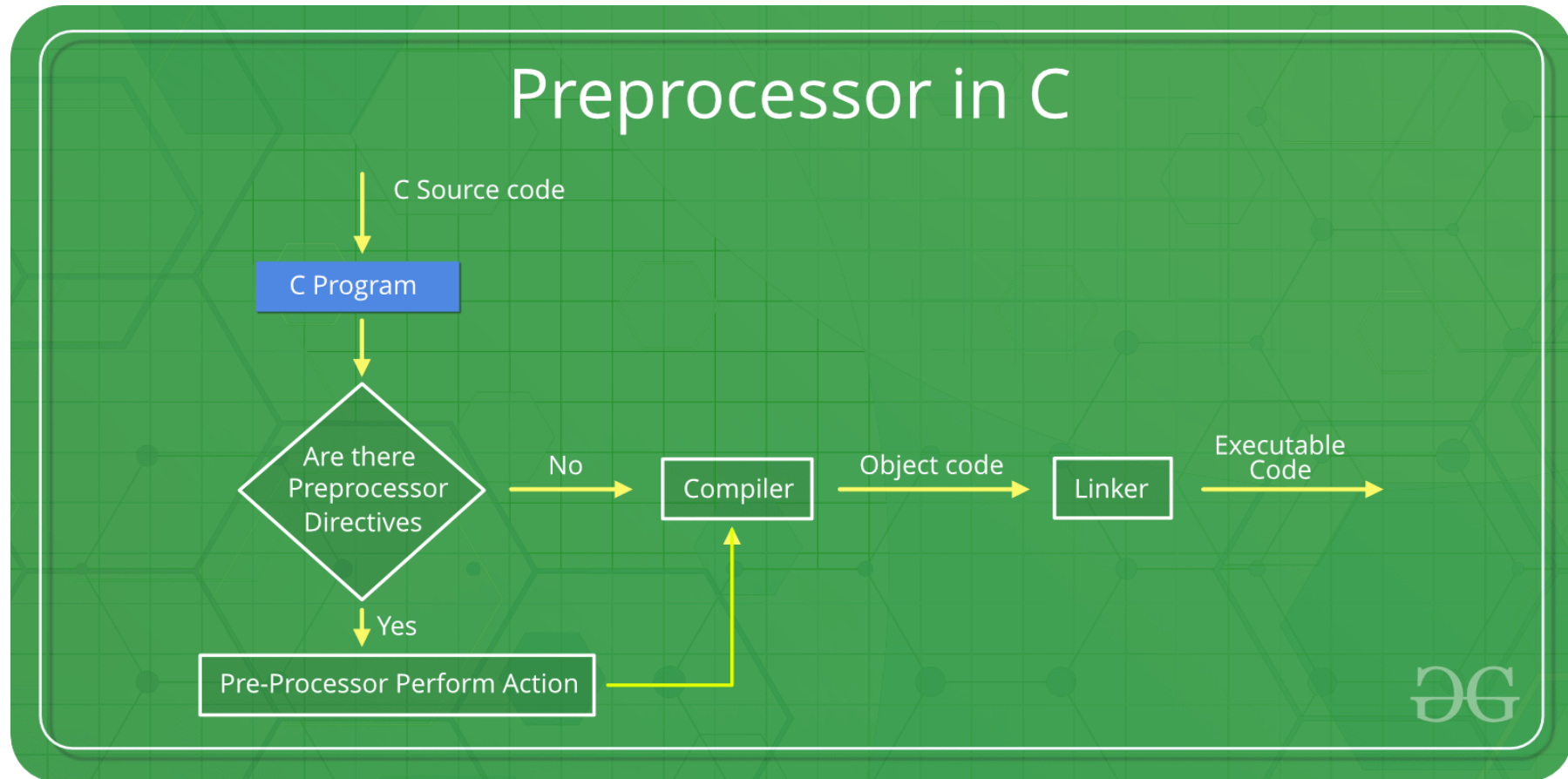
Topics

- ❖ Multi-file C programs, modularity
- ❖ Make
- ❖ **More on C Preprocessor**
- ❖ More on Linker



More on C Preprocessor

- Preprocessors are programs that process our source code before compilation



<https://www.geeksforgeeks.org/cc-preprocessors/>

More on C Preprocessor

- ❖ Preprocessor programs provide preprocessor directives which tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a '#' (hash) symbol.
- ❖ There are 4 main types of preprocessor directives:
 - Macros
 - File Inclusion (as we covered already)
 - Conditional Compilation
 - Other directives

<https://www.geeksforgeeks.org/cc-preprocessors/>

#include directive

- ❖ Both user and system header files are included using the preprocessing directive ‘#include’. Two variants:
- ❖ #include <*file*>
 - for system header files.
 - searches for a file named file in a standard list of system directories. prepend directories to this list with the -I option
- ❖ #include "*file*"
 - Used for header files of your own program.
 - Searches for a file named file first in the directory containing the current file, then in the quote directories and then the same directories used for <file>. You can prepend directories to the list of quote directories with the -iquote option.

Include directive

- ❖ Scan specified file, and replace #include line with the contents of the header file

program.c

```
int x;  
#include "header.h"  
  
int  
main (void)  
{  
    puts (test ());  
}
```

header.h

```
char *test (void);
```



```
int x;  
char *test (void);  
  
int  
main (void)  
{  
    puts (test ());  
}
```

Search Path

- ❖ GCC looks in several different places for system file headers. On a normal Unix system, if you do not instruct it otherwise, it will look for headers requested with `#include <file>` in:

- `/usr/local/include`
- `/usr/lib/gcc-lib/target/version/include`
- `/usr/target/include`
- `/usr/include`

Where is stdio.h

```
sonmeza@cm5c257:/usr/include
[sonmeza@cm5c257 include]$ cd /usr/include
[sonmeza@cm5c257 include]$ ls
aio.h          fnmatch.h      limits.h        pcreposix.h    stdio.h
aliases.h      fpu_control.h  link.h          pcre_scanner.h stdlib.h
alloca.h        fstab.h        linux           pcre_stringpiece.h string.h
a.out.h         fts.h          locale.h        poll.h          strings.h
argp.h          ftw.h          malloc.h        printf.h        sys
argz.h          _G_config.h    math.h          protocols       syscall.h
ar.h            gconv.h        mcheck.h        pthread.h       sysexits.h
arpa            gelf.h         memory.h        pty.h          syslog.h
asm             getopt.h       misc            pwd.h           tar.h
asm-generic     gettext-po.h   mntent.h        python2.7       termio.h
assert.h        gio-unix-2.0   monetary.h      python3.6m      termios.h
autosprintf.h  glib-2.0       mono-2.0        rdma            tgmath.h
bits           glob.h         mqueue.h        re_comp.h       thread_db.h
byteswap.h     gnu            mtd             regex.h         time.h
c++            gnu-versions.h net              regexp.h        ttyent.h
complex.h      grp.h          netash          resolv.h        uapi
cpio.h         gshadow.h     netatalk        rpc             uchar.h
cpufreq.h      iconv.h       netax25         rpcsvc          ucontext.h
crypt.h        ieee754.h     netdb.h         sasl            ulimit.h
ctype.h        ifaddrs.h     neteconet       sched.h         unistd.h
dirent.h       inttypes.h    netinet         scsi            ustat.h
dlfcn.h        langinfo.h    netipx          search.h        utime.h
```

Once- Only Headers

- ❖ If a header file happens to be included twice, the compiler will process its contents twice.
- ❖ This is very likely to cause an error, e.g. when the compiler sees the same structure definition twice.
 - Even if it does not, it will certainly waste time.

- ❖ Solution:

```
/* File foo. */  
#ifndef FILE_FOO_SEEN  
#define FILE_FOO_SEEN  
  
the entire file  
  
#endif /* !FILE_FOO_SEEN */
```

When the header is included again, the conditional will be false, because FILE_FOO_SEEN is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

For more on Header files:

- ❖ https://gcc.gnu.org/onlinedocs/gcc-3.0.2/cpp_2.html

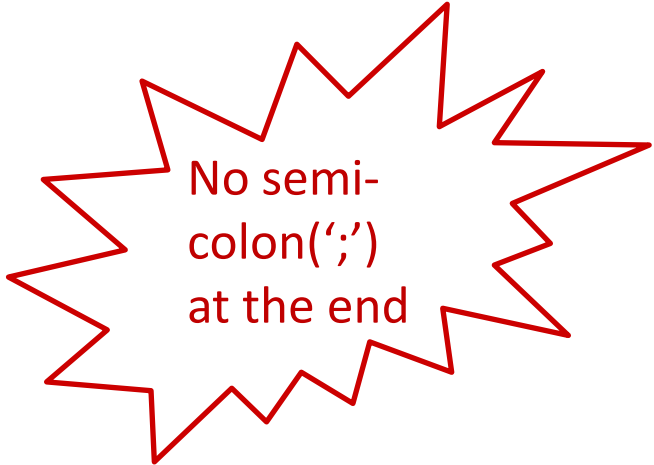
Macros

- ❖ A piece of code in a program which is given some name. Whenever this name is encountered by the preprocessor the preprocessor replaces the name with the actual piece of code.
- ❖ The '#define' directive is used to define a macro.

```
#include <stdio.h>

// macro definition
#define LIMIT 5
int main()
{
    for (int i = 0; i < LIMIT; i++) {
        printf("%d \n",i);
    }

    return 0;
}
```



No semi-
colon(';')
at the end

<https://ide.geeksforgeeks.org/PPf528NsDA>

Macros with arguments

- ❖ We can also pass arguments to macros. Macros defined with arguments works similarly as functions.

```
1  #include <stdio.h>
2
3  // macro with parameter
4  #define AREA(length, width) (length * width)
5  int main()
6  {
7      int length = 3, width = 5;
8
9      printf("Area of rectangle is: %d", AREA(length , width));
10
11     return 0;
12 }
```

Area of rectangle is: 15

No new stack, no function call overhead.
Replaced during preprocessing.

Macros with arguments

- ❖ Simplify printing an integer

```
1  #include <stdio.h>
2
3  // macro with parameter
4  #define PINT(x) printf("%d\n",x)
5  int main()
6  {
7      PINT(10);
8      return 0;
9  }
```

<https://ide.geeksforgeeks.org/9w2MpjKWJ2>

Other advanced defines

- ❖ If you are using $2 * \text{PI}$ very often
 - You can use define as:
 - `#define TWO_PI 2.0*3.141592654`
 - Then use `TWO_PI` as
 - Return `TWO_PI * r`

- ❖ It doesn't have to be a valid expression
 - `#define LEFT_SHIFT_4 <<4`
 - `x = y LEFT_SHIFT_4 ;`

Other advanced defines

- ❖ You can change syntax with define

```
#define AND &&
```

```
#define OR    ||
```

```
#define EQUALS ==
```

If (x>0 AND x<10)....

If ((y EQUALS 0) OR (y EQUALS value))/* no more possible use of
single = by mistake! */

Other advanced defines

- ❖ Defined value can itself reference to another defined value

```
#define PI 3.141592654
```

```
#define TWO_PI 2.0*PI
```

- ❖ The following is also valid, just needs to be defined before use of TWO_PI

```
#define TWO_PI 2.0*PI
```

```
#define PI 3.141592654
```

Pitfalls

- ❖ Let's say we have a define such as

```
#define SQUARE(x) x*x
```

- ❖ The statement $y = \text{SQUARE}(v)$ assigns value of v^2 to y
- ❖ What would happen in case of the following?

```
y = SQUARE (v+1);
```

- ❖ Is y going to be assigned value of $(v+1)^2$
- ❖ Let's replace x with $v+1$, statement becomes

```
y= v + 1 * v + 1
```

- How we can fix this?

Pitfalls

- ❖ Here is the fix:

```
#define SQUARE(x) ((x) * (x))
```

- ❖ The statement `y = SQUARE (v+1);`
- ❖ Will then be correctly evaluated as
`y = ((v+1) * (v+1));`

Define operators

- ❖ If you place # in front of a parameter, preprocessor creates a constant string out of macro argument. Preprocessor inserts double quotation around
- ❖ If we have the following define

```
#define STR(x) #x
```
- ❖ The following statement

```
STR(hello)
```

 - Will be expanded into “hello”

<https://onlinegdb.com/ryamRN1BL>

Conditional Compilation

- ❖ You can conditionally compile parts of a program using the **#if**, **#else**, **#endif**, **#ifdef**, **#undef** and **#ifndef** directives

```
#define DEFINED
...

#if 0
/* This does not get compiled */
#else
/* This does get compiled */
#endif

#ifdef UNKNOWNVALUE
/* This does not get compiled */
#else
/* This does get compiled */
#endif

#ifndef DEFINED
/* This does not get compiled */
#else
/* This does get compiled */
#endif
```

```
/* A quick way to comment out code,
as typically used in doing debugging and
unit testing */

int main( void ) {

    // Declare your variables here
    float myFloats[NUMBER_ENTRIES];

    #if 0
    // Read float values
    for ( i=0; i<NUMBER_ENTRIES; i++ ) {
        scanf( "%f", &myFloats[i] );
    }

    // Show the list of unsorted values
    printCharline( '*', 69 );
    printf( "Received and computed\n" );
    #endif

    ...
}
```

Conditional Compilation

- ❖ System specific compilation

```
#ifdef UNIX
```

```
    #define DATADIR "/uxn1/data"
```

```
#else
```

```
    #define DATADIR "\\usr\\data"
```

- ❖ When compiling, if the sytem is UNIX

```
gcc -D UNIX program.c -o program
```

Avoiding Multiple inclusion of headers

- ❖ Multiple inclusion may cause errors.

```
#ifndef HEADER
```

```
#define HEADER
```

```
...
```

```
#endif
```

- ❖ If it is first time the header added, HEADER will become defined and code will be inserted; otherwise code will not be inserted during preprocessing.

Other Directives

❖ line directive

- #line 100
 - Sets the next line of the program to 100

```
1  #include<stdio.h>
2
3  int main(){
4      int i = 0;
5      #line 100
6
7
8
9      printf( "%d\n",__LINE__);
10
11     printf( "%d\n",__LINE__);
12 }
```

```
103
105
```

<https://onlinegdb.com/SJ6I1ykFH>

<https://onlinegdb.com/S1EQHskSL>

Other Directives

- ❖ error command
 - #error message
 - Prints error message detected by preprocessor

```
1  #define CMSC257
2
3  #ifdef CMSC257
4      #error You are in the right place
5  #endif
6
7  #undef CMSC257
8
9  #ifndef CMSC257
10     #error You are in the wrong place
11 #endif
12
13 #include<stdio.h>
14
15 int main(){
16     int i = 0;
17 }
```

<https://onlinegdb.com/ry7nbykYH>

```
main.c:4:6: error: #error You are in the right place
      #error You are in the right place
      ^~~~~~
main.c:10:6: error: #error You are in the wrong place
      #error You are in the wrong place
      ^~~~~~
```

Other Directives

❖ pragma directive

■ pragma tokens

- Causes compiler to perform implementation-defined actions
- Such as turning on and off some features

```
#include<stdio.h>

#pragma GCC poison printf

int main()
{
    printf("Hello");
}
```

```
main.c: In function 'main':
main.c:9:3: error: attempt to use poisoned "printf"
    printf("Hello");
    ^
```

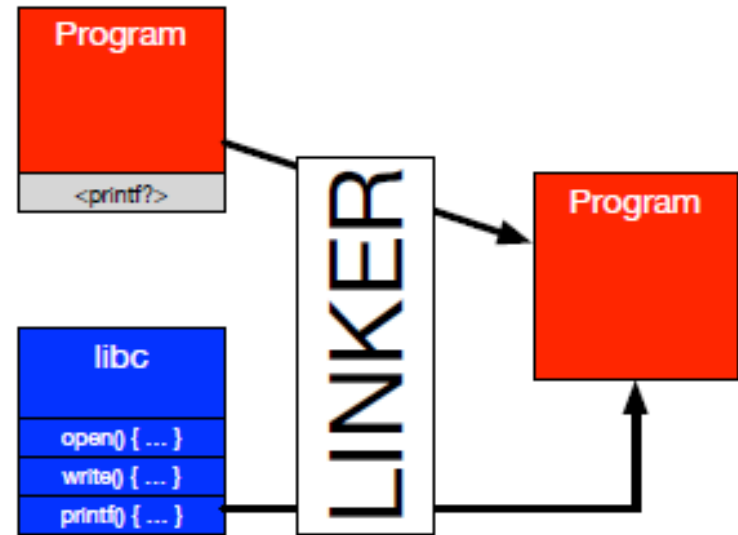
<https://onlinegdb.com/BJ9fXJ1KB>

Topics

- ❖ Multi-file C programs, modularity
- ❖ Make
- ❖ More on C Preprocessor
- ❖ **More on Linker**
 - **Linking and Dynamic Linking**
 - **Duplicated Symbol Resolution**
 - **Creating Libraries**

What is a “static” library

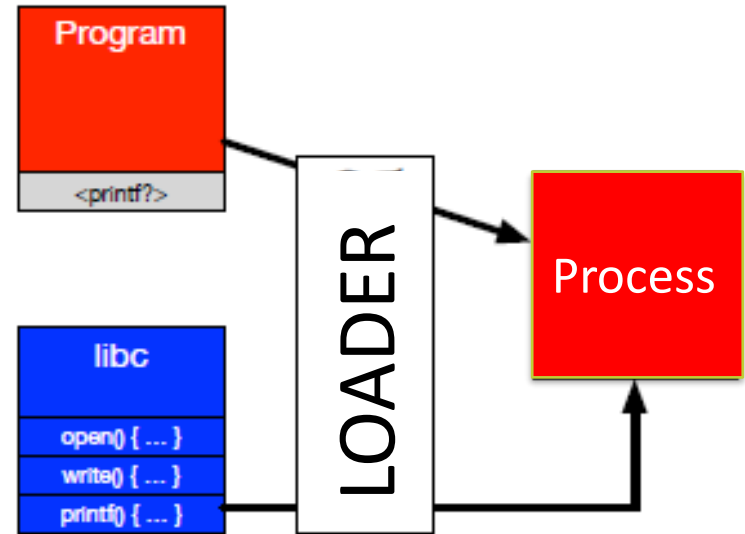
- ❖ A library is a collection of related code and functions that are “linked” against a C program.
- ❖ Your program object code has “*unresolved symbols*” in the code



- ❖ The linker pulls chunks of the library containing those symbols and places them into the program
- ❖ The program is done when all the pieces are resolved
- ❖ called “static” linking because this is **done at link time**

What is a “dynamic” library?

- ❖ A dynamic library is a collection of related code and functions that are “resolved” at run time.
- ❖ Your program object code has “*unresolved symbols*” in the code



- ❖ The loader pulls chunks of the library containing those symbols and places them into the process
- ❖ The symbols are resolved when the process is started or later during execution
- ❖ called “dynamic” because it **can be done any time ...**

Why Linkers?

❖ Reason 1: Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions e.g., Math library, standard C library



Why Linkers? (cont)

❖ Reason 2: Efficiency

■ Time: Separate compilation

- Change one source file, compile, and then relink.
- No need to recompile other source files.

■ Space: Libraries

- Common functions can be aggregated into a single file...
- Executable files contain only code for the functions they actually use.



What Do Linkers Do?

- Programs define and reference *symbols* (global variables and functions):
 - `void swap() {...}` `/* define symbol swap */`
 - `swap();` `/* reference symbol swap */`
 - `int *xp = &x;` `/* define symbol xp, reference x */`
- Symbol definitions are stored in object file (by assembler) in *symbol table*.
 - Symbol table is an array of `structs`
 - Each entry includes name, size, and location of symbol.
- **The linker associates each symbol reference with exactly one symbol definition.**
- Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.



Three Kinds of Object Files (Modules)

❖ Relocatable object file (`.o` file)

- Contains code and data in a form that can be combined with other relocatable object files to form executable object file.
 - Each `.o` file is produced from exactly one source (`.c`) file

❖ Executable object file (a `.out` file)

- Contains code and data in a form that can be copied directly into memory and then executed.

❖ Shared object file (`.so` file)

- Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
- Called *Dynamic Link Libraries* (DLLs) by Windows



Executable and Linkable Format (ELF)

- ❖ Standard binary format for object files
- ❖ One unified format for
 - Relocatable object files (`.o`),
 - Executable object files (`a.out`)
 - Shared object files (`.so`)
- ❖ Generic name: ELF binaries
- ❖ We will not go into detail of ELF format, please read textbook

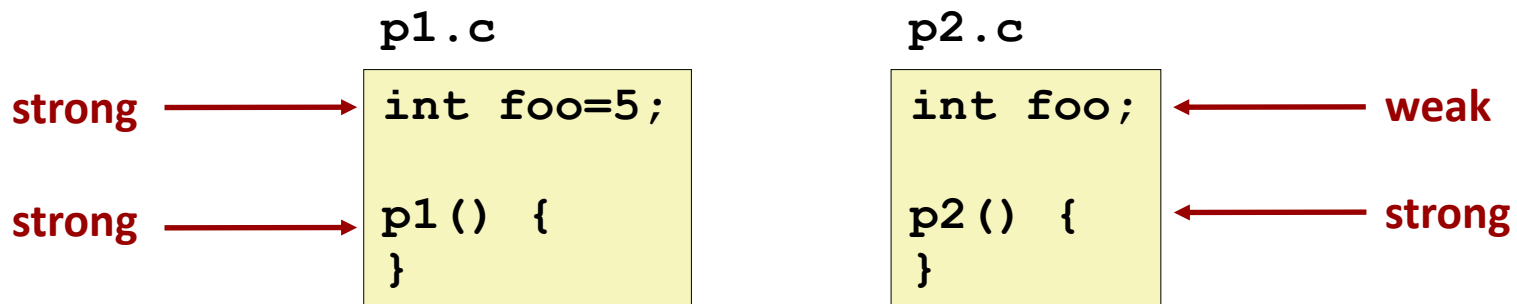


Topics

- ❖ Multi-file C programs, modularity
- ❖ Make
- ❖ More on C Preprocessor
- ❖ **More on Linker**
 - Linkers and Loaders
 - Duplicated Symbol Resolution
 - Creating Libraries

How Linker Resolves Duplicate Symbol Definitions

- ❖ Program symbols are either *strong* or *weak*
 - **Strong**: procedures and initialized globals
 - **Weak**: uninitialized globals



Linker's Symbol Rules

- ❖ Rule 1: Multiple strong symbols are not allowed
 - Each item can be defined only once
 - Otherwise: Linker error
- ❖ Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol
 - References to the weak symbol resolve to the strong symbol
- ❖ Rule 3: If there are multiple weak symbols, pick an arbitrary one



Linker Puzzles

```
int x;  
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (p1, Rule1)

```
int x;  
p1() {}
```

```
int x;  
p2() {}
```

References to `x` will refer to the same uninitialized int. Is this what you really want? (Rule3)

```
int x;  
int y;  
p1() {}
```

```
double x;  
p2() {}
```

Writes to `x` in `p2` might overwrite `y` in `p1`!
Evil! (Rule3)

```
int x=7;  
int y=5;  
p1() {}
```

```
double x;  
p2() {}
```

Writes to `x` in `p2` will overwrite `y`!
Nasty! (Rule2, `int x` is strong)

```
int x=7;  
p1() {}
```

```
int x;  
p2() {}
```

References to `x` will refer to the same initialized variable.



Sample

```
1 #include <stdio.h>
2
3 int x=10;
4 int y=4;
5 int main()
6 {
7
8     printf("Hello World\n");
9     printf("x in main before foo = %d\n", x);
10    printf("y = %d\n", y);
11    foo();
12    printf("x in main after foo: = %d\n", x);
13    printf("y = %d\n", y);
14
15
16    return 0;
17 }
```

```
1 #include <stdio.h>
2
3 double x;
4
5 int foo()
6 {
7     x = 67899009987766554433.00;
8     printf("x in foo: %lf\n", x);
9
10    return 0;
11 }
```

```
Hello World
x in main before foo = 10
y = 4
x in foo: 67899009987766550528.000000
x in main after foo: = -289977292
y = 1141731917
```

How to prevent

- ❖ This happens silently without any warning.
- ❖ Manifest itself much later in the execution
- ❖ In a large system with hundreds of modules, a bug of this kind is extremely hard to fix.
- ❖ When in doubt use flag:
 - `-fno-common`
 - Triggers error if it encounters multiply defined global symbols

Global Variables

- ❖ Avoid if you can
- ❖ Otherwise
 - Use **static** if you can
 - Initialize if you define a global variable
 - Use **extern** if you reference an external global variable



External Linkage

- ❖ `extern` makes a *declaration* of something externally-visible

```
#include <stdio.h>

// A global variable, defined and
// initialized here in foo.c.
// It has external linkage by
// default.
int counter = 1;

int main(int argc, char** argv) {
    printf("%d\n", counter);
    bar();
    printf("%d\n", counter);
    return 0;
}
```

foo.c

```
#include <stdio.h>

// "counter" is defined and
// initialized in foo.c.
// Here, we declare it, and
// specify external linkage
// by using the extern specifier.
extern int counter;

void bar() {
    counter++;
    printf("(b): counter = %d\n",
           counter);
}
```

bar.c

Internal Linkage

- ❖ `static` (in the global context) restricts a definition to visibility within that file

```
#include <stdio.h>

// A global variable, defined and
// initialized here in foo.c.
// We force internal linkage by
// using the static specifier.
static int counter = 1;

int main(int argc, char** argv) {
    printf("%d\n", counter);
    bar();
    printf("%d\n", counter);
    return 0;
}
```

foo.c

```
#include <stdio.h>

// A global variable, defined and
// initialized here in bar.c.
// We force internal linkage by
// using the static specifier.
static int counter = 100;

void bar() {
    counter++;
    printf("(b): counter = %d\n",
           counter);
}
```

bar.c



Function Visibility

```
// By using the static specifier, we are indicating  
// that foo() should have internal linkage. Other  
// .c files cannot see or invoke foo().  
static int foo(int x) {  
    return x*3 + 1;  
}  
  
// Bar is "extern" by default. Thus, other .c files  
// could declare our bar() and invoke it.  
int bar(int x) {  
    return 2*foo(x);  
}
```

bar.c

```
#include <stdio.h>  
  
extern int bar(int x); // "extern" is default, usually omit  
  
int main(int argc, char** argv) {  
    printf("%d\n", bar(5));  
    return 0;  
}
```

main.c



Linkage Issues

- ❖ Every global (variables and functions) is `extern` by default
 - Unless you add the `static` specifier, if some other module uses the same name, you'll end up with a collision!
 - Best case: compiler (or linker) error
 - Worst case: stomp all over each other
- ❖ It's good practice to:
 - Use `static` to “defend” your globals
 - Hide your private stuff!
 - Place external declarations in a module's header file
 - Header is the public specification

Static Confusion...

- ❖ C has a *different* use for the word “static”: to create a persistent *local* variable
 - The storage for that variable is allocated when the program loads, in the `.data` segment
 - Retains its value across multiple function invocations

```
void foo() {  
    static int count = 1;  
    printf("foo has been called %d times\n", count++);  
}  
  
void bar() {  
    int count = 1;  
    printf("bar has been called %d times\n", count++);  
}  
  
int main(int argc, char** argv) {  
    foo(); foo(); bar(); bar(); return 0;  
}
```

Topics

- ❖ Multi-file C programs, modularity
- ❖ Make
- ❖ More on C Preprocessor
- ❖ **More on Linker**
 - Linkers and Loaders
 - Duplicated Symbol Resolution
 - Creating Libraries

Packaging Commonly Used Functions

- ❖ How to package functions commonly used by programmers?
 - Math, I/O, memory management, string manipulation, etc.
 - **Option 1:** Put all functions into a single source file
 - Programmers link big object file into their programs
 - Space and time inefficient
 - **Option 2:** Put each function in a separate source file
 - Programmers explicitly link appropriate binaries into their programs
 - More efficient, but burdensome on the programmer



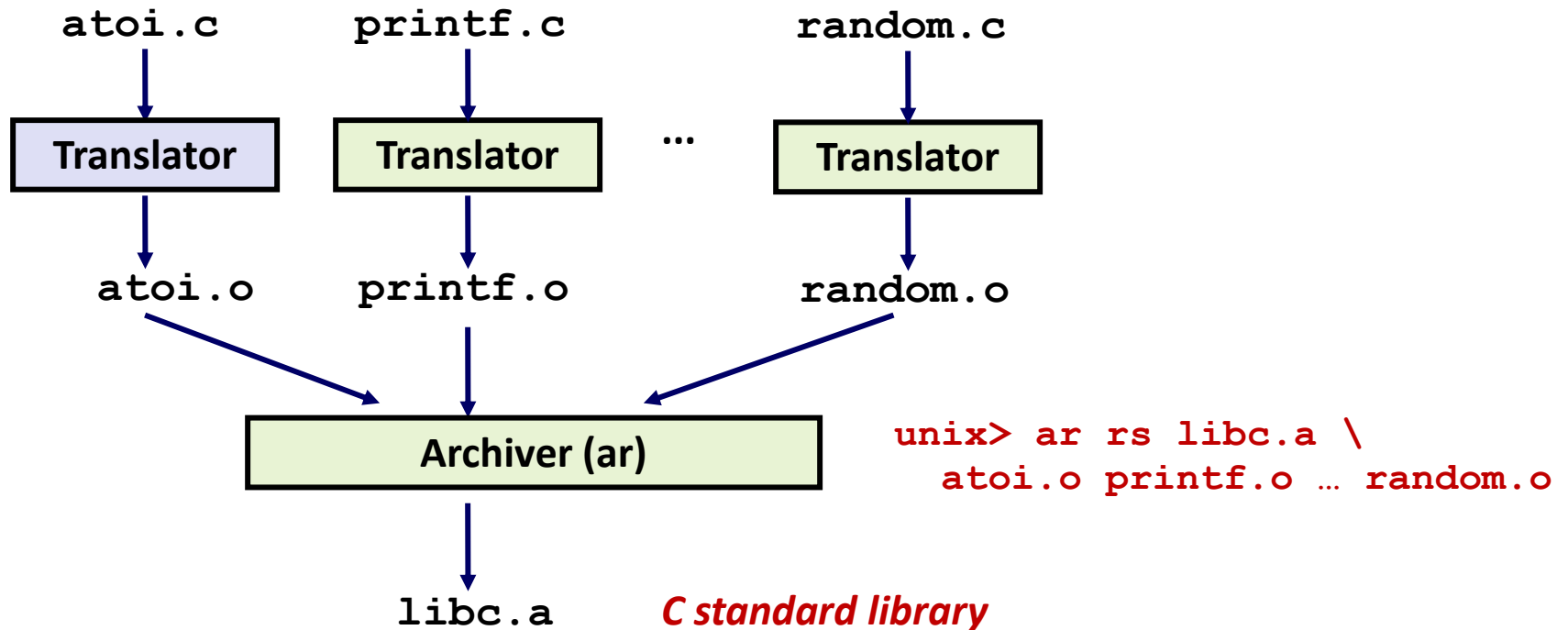
Old-fashioned Solution: Static Libraries

❖ Static libraries (.a archive files)

- Concatenate related relocatable object files into a single file with an index (called an *archive*).
- Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
- If an archive member file resolves reference, link it into the executable.



Creating Static Libraries



- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.
- r-> add into archive, s->update index of archive



Commonly Used Libraries

`libc.a` (the C standard library)

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, ...)

```
% ar -t libc.a | sort
...
fork.o
...
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
...
```

```
% ar -t libm.a | sort
...
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
...
```

To find where they are located type:
`gcc --print-file-name=libc.a`

`cd` to the folder then use `ar -t`
(means display contents of archive), You can find in VM, in server there is only `libc.so`



Linking with Static Libraries

```
#include <stdio.h>
#include "vector.h"
```

```
int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];
```

```
int main()
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n",
        z[0], z[1]);
    return 0;
}
```

main2.c

libvector.a

```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

addvec.c

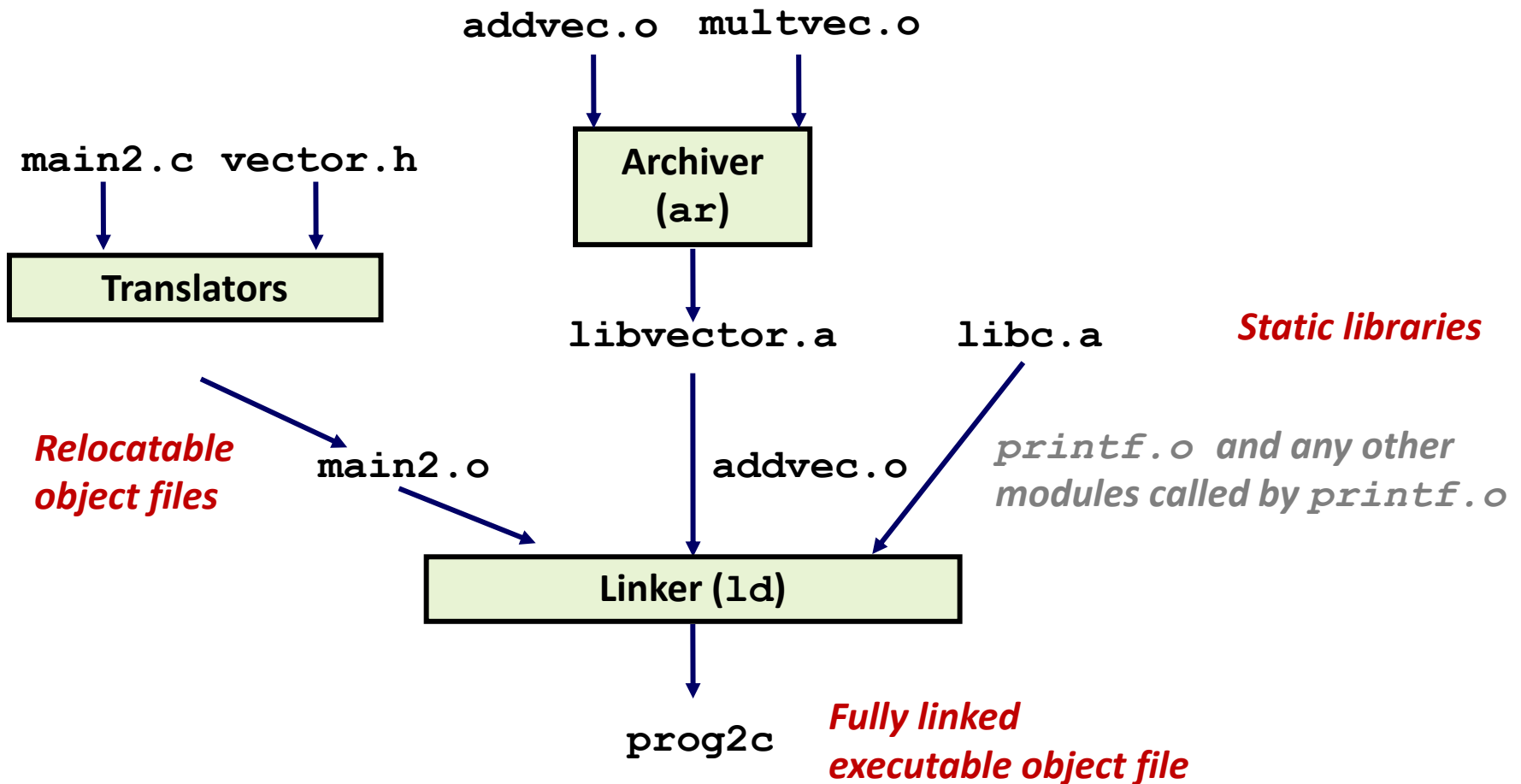
```
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

multvec.c



Linking with Static Libraries



"c" for "compile-time"



Sample make file with static library and linking

```
OBJS = LinkedList.o HashTable.o Assert257.o
HEADERS = LinkedList.h HashTable.h Assert257.h
```

```
AR = ar
ARFLAGS = rcs
```

```
libpr2.a: $(OBJS) $(HEADERS) FORCE
    $(AR) $(ARFLAGS) libpr2.a $(OBJS)
```

```
example_program_ll: example_program_ll.o libpr2.a $(HEADERS) FORCE
    $(CC) $(CFLAGS) -o example_program_ll example_program_ll.o $(LDFLAGS)
```

```
CFLAGS += -g -Wall -I. -I.. -O0
LDFLAGS += -L. -lpr2
```

R - replace existing code with the objects passed

C - create the library if needed

S - create an index for “relocatable code”

-all targets depending on FORCE will always have their recipe run.

-L searchdir: Look in directory for library file

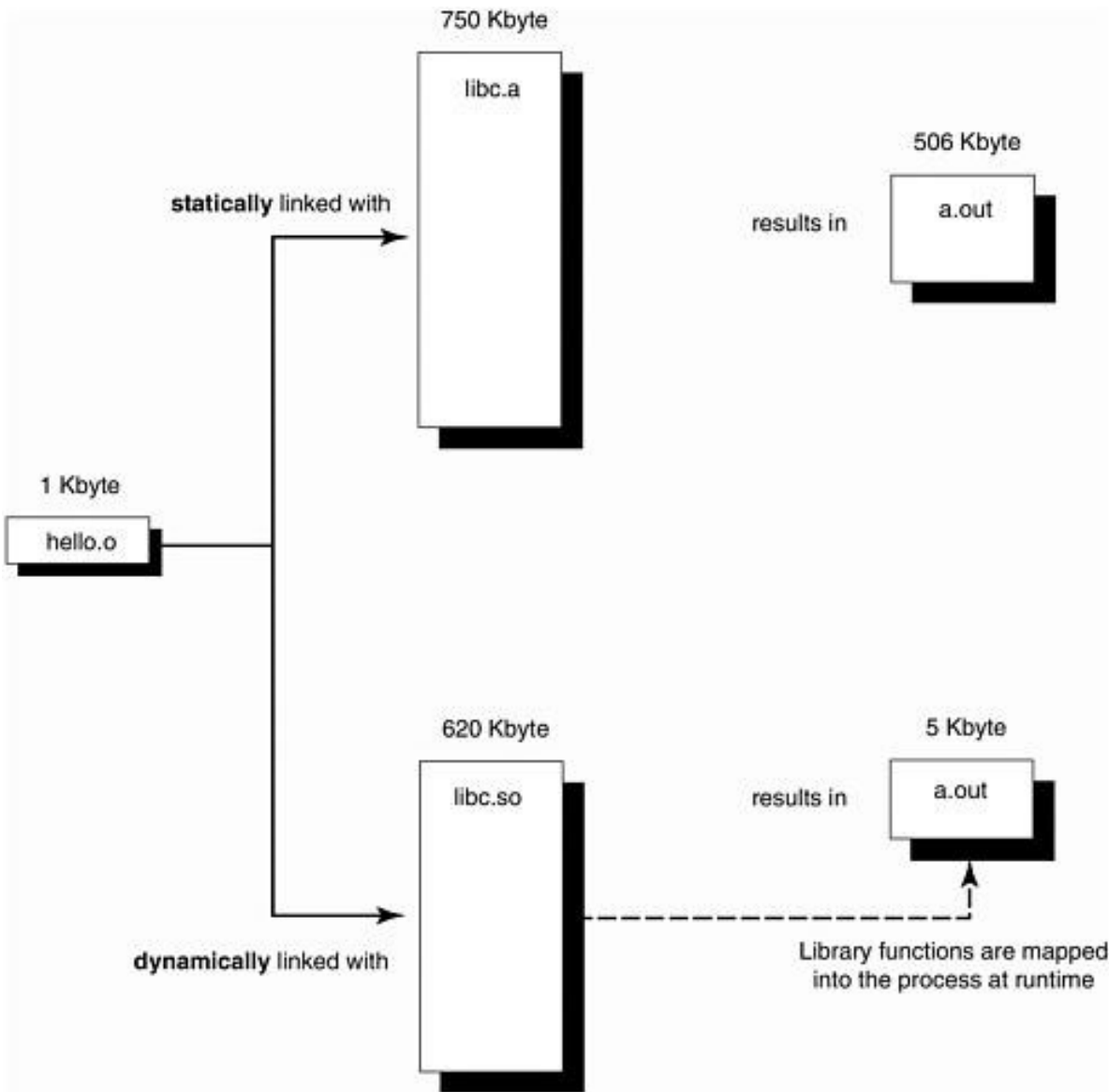
-l namespec: link with library file (lpr2 will link libpr2.a or libpr2.so)

Modern Solution: Shared Libraries

- ❖ Static libraries have the following disadvantages:
 - Minor bug fixes of system libraries require each application to explicitly relink
 - Larger file size.
 - Each process will have it's own copy of function in memory.

- ❖ Modern solution: Shared Libraries
 - Object files that contain code and data that are loaded and linked into an application *dynamically*, at either *load-time* or *run-time*
 - Also called: dynamic link libraries, DLLs, `.so` files
 - Small file size
 - Shared code section in main memory.





Note: sizes in diagram are for illustrative purposes; your mileage may vary.

<https://medium.com/@meghamohan/everything-you-need-to-know-about-libraries-in-c-e8ad6138cbb4>

Shared Libraries (cont.)

- ❖ Dynamic linking can occur when executable is first loaded and run (load-time linking).
 - Standard C library (**libc.so**) usually dynamically linked.
- ❖ Dynamic linking can also occur after program has begun (run-time linking).
 - In Linux, this is done by calls to the **dlopen()** interface.
 - Distributing software. (More in next slide)
 - High-performance web servers. (Dynamic linking based solution)
 - Shared library routines can be shared by multiple processes.



Uses of dynamic libraries

❖ Distributing software:

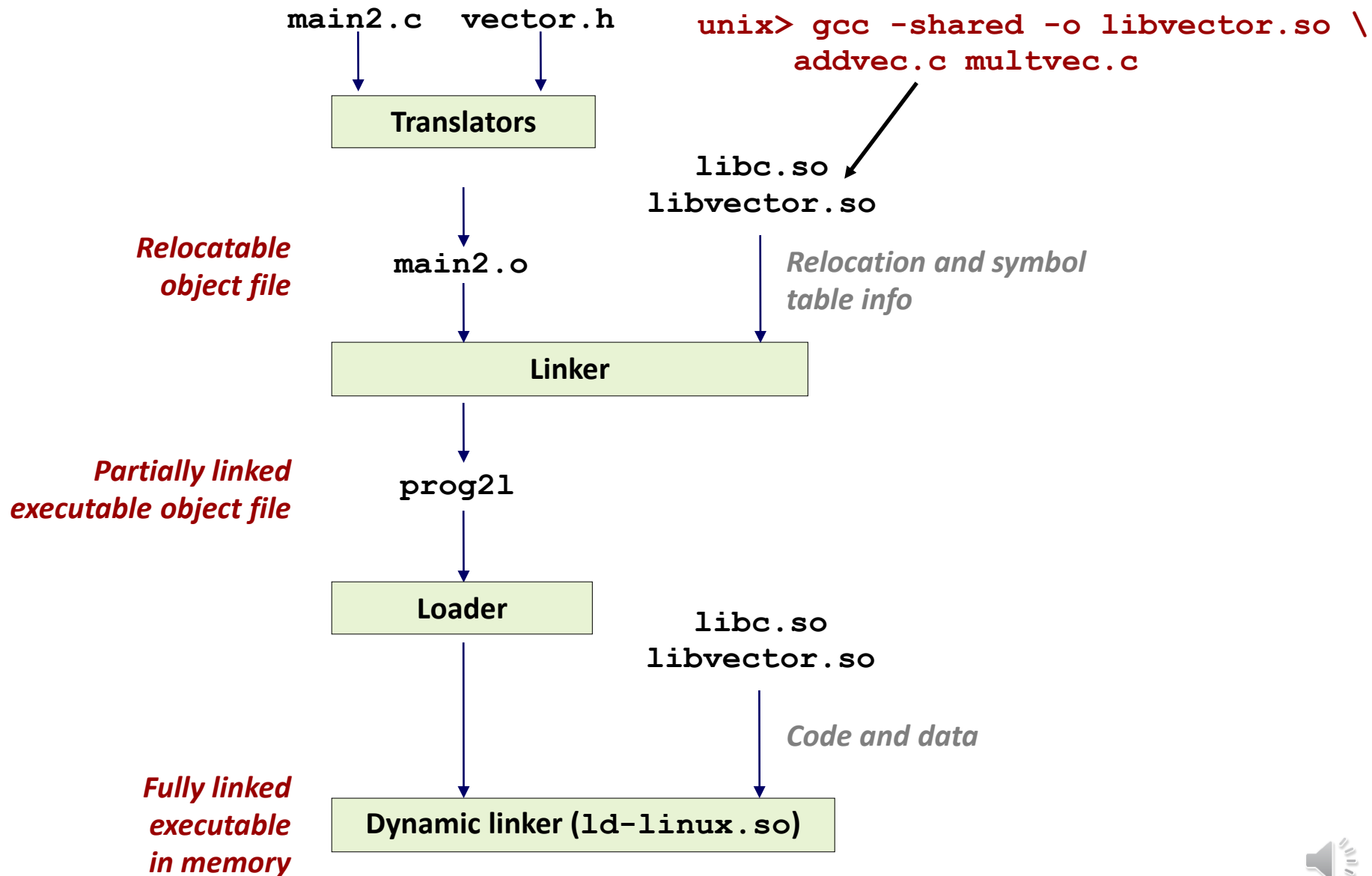
- Developers of Microsoft Windows applications frequently use shared libraries to distribute software updates.
- New copy of shared library is downloaded by users, and used as a replacement for the current version.
- The next time they run the application, it will automatically link and load the new shared library.

Building a dynamic library

- A **dynamically linked** library produces object code that is inserted into program at *execution time*.
 - You are building loadable versions of the library which the loader uses to launch the application
 - To run the command, pass to gcc using “-shared”, e.g.,
`gcc -shared -o libmyexample.so a.o b.o c.o d.o`
- Important: all object files to be placed in library must have been compiled to *position-independent code (PIC)*
`gcc a.c -fpic -c -Wall -g -o a.o`
- Naming: same as before, only with `.so` extension



Dynamic Linking at Load-time



Checking which dynamic libraries loaded

```
cs257@cs257-VirtualBox:~/Desktop/lab4$ ls
lab4  lab4.c
cs257@cs257-VirtualBox:~/Desktop/lab4$ ldd lab4
        linux-gate.so.1 (0xb7ef8000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7cff000)
        /lib/ld-linux.so.2 (0xb7ef9000)
```

Check here for more examples:

<http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html>

Dynamic Linking at Run-time

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

```
int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];
```

```
int main()
{
```

```
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;
```

```
    /* Dynamically load the shared library that contains addvec()
    */
```

```
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

dll.c

```
#include <dlfcn.h>
```

```
void *dlopen(const char *filename, int flag)
```

returns pointer to handle if OK, NULL on error

<https://pubs.opengroup.org/onlinepubs/009695399/functions/dlopen.html>



Dynamic Linking at Run-time

...

```
/* Get a pointer to the addvec() function we just loaded */
```

```
addvec = dlsym(handle, "addvec");
```

```
if ((error = dlerror()) != NULL) {
```

```
    fprintf(stderr, "%s\n", error);
```

```
    exit(1);
```

```
}
```

```
/* Now we can call addvec() just like any other function */
```

```
addvec(x, y, z, 2);
```

```
printf("z = [%d %d]\n", z[0], z[1]);
```

```
/* Unload the shared library */
```

```
if (dlclose(handle) < 0) {
```

```
    fprintf(stderr, "%s\n", dlerror());
```

```
    exit(1);
```

```
}
```

```
return 0;
```

```
}
```

d11.c



Linking Summary

- ❖ Linking is a technique that allows programs to be constructed from multiple object files.
- ❖ Linking can happen at different times in a program's lifetime:
 - Compile time (when a program is compiled)
 - Load time (when a program is loaded into memory)
 - Run time (while a program is executing)
- ❖ Understanding linking can help you avoid nasty errors and make you a better programmer.

