FOLLOW ALONG WITH THE EXAMPLES…

```
$ git clone https://gitlab.com/jtfrey/unix-software-dev.git

$ cd unix-software-dev

$ git checkout tags/session1

$ ls -l
total 8
-rw-r--r--   1 frey  staff   350B Apr 21 15:25 README.md
drwxr-xr-x  15 frey  staff   510B Apr 21 15:23 src-1
drwxr-xr-x   8 frey  staff   272B Apr 21 15:23 src-2
```

https://gitlab.com/jtfrey/unix-software-dev.git

UNIX SOFTWARE DEVELOPMENT BASICS

PROJECT STRUCTURE AND BUILD MANAGEMENT WITH MAKE

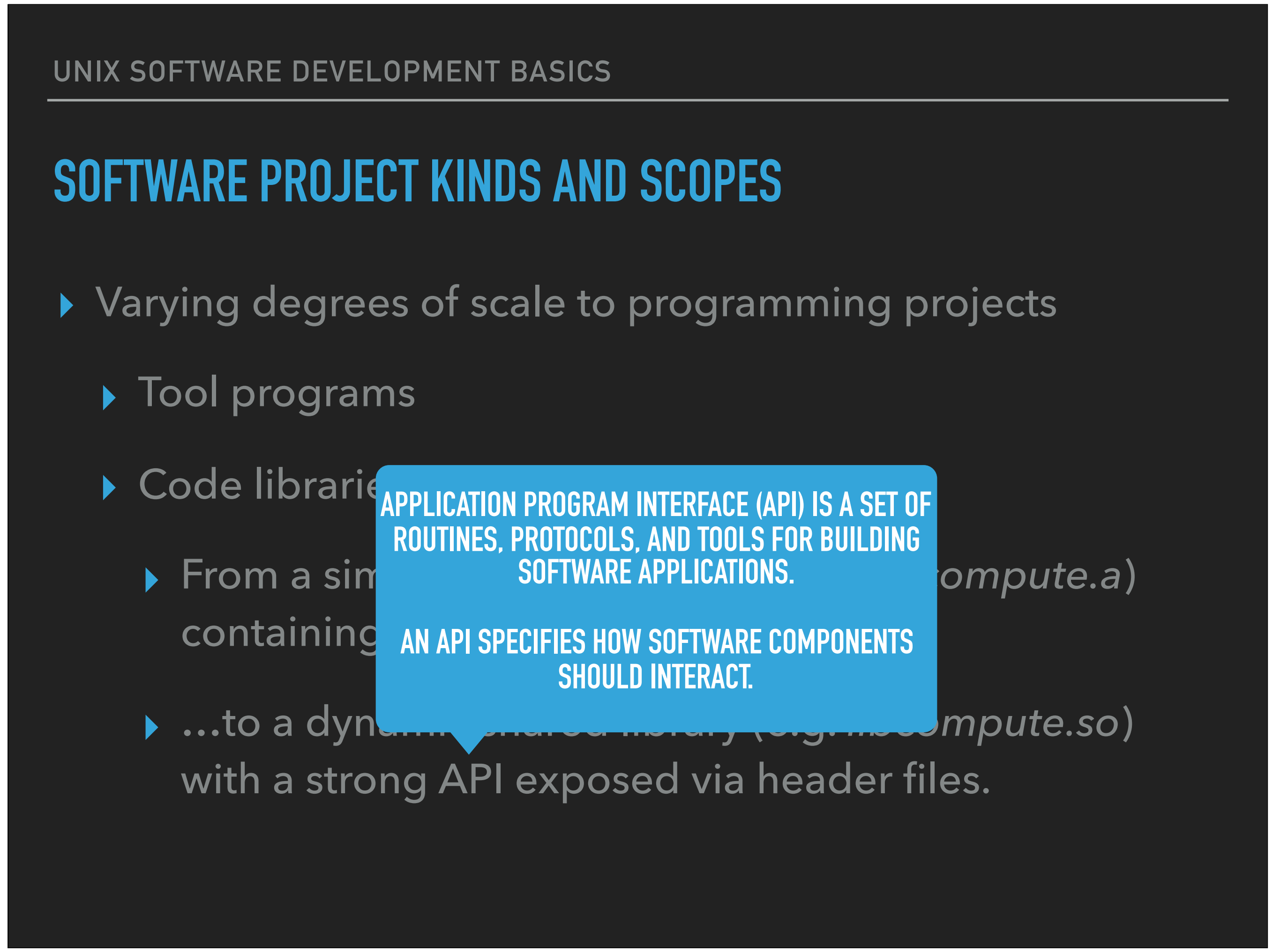
SOFTWARE PROJECT KINDS AND SCOPES

- Varying degrees of scale to programming projects
  - Tool programs
    - From a shell or Perl script to extract key data from an output file…
    - …to a multi-file Fortran or Python program that post-processes (via computation) data from an output file

SOFTWARE PROJECT KINDS AND SCOPES

- Varying degrees of scale to programming projects
  - Tool programs
  - Code libraries
    - From a simple Unix archive file (e.g. *libcompute.a*) containing compiled object code…
    - …to a dynamic shared library (e.g. *libcompute.so*) with a strong API exposed via header files.

- Many scripting (non-compiled) languages also have the concept of code libraries
  - Matlab .m files that add functions to the environment
  - Python modules (e.g. see /usr/lib64/python2.6/site-packages on Farber)
  - Perl modules (e.g. see /usr/lib64/perl5 on Farber)
- Note that creating an API demands more planning and structure to a project



- Many scripting (non-compiled) languages also have the concept of code libraries
  - Matlab .m files that add functions to the environment
  - Python modules (e.g. see /usr/lib64/python2.6/site-packages on Farber)
  - Perl modules (e.g. see /usr/lib64/perl5 on Farber)
- Note that creating an API demands more planning and structure to a project

SOFTWARE PROJECT KINDS AND SCOPES

- Varying degrees of scale to programming projects
    - Tool programs
    - Code libraries
    - Software suites
        - Containing a mix of tools and libraries

SOFTWARE PROJECT KINDS AND SCOPES

- Varying degrees of scale to programming projects
- A project that starts at the simpler end of the scale can evolve toward the complex end…
- …or it could move between the types
  - a collection of tool programs ⇒ library
  - a very complex tool program ⇒ software suite

STRUCTURING SOFTWARE PROJECTS

- Code is comprised of one or more files (be they source code or interpreted script, etc.)
- Any OS to which you've been exposed probably has the same solution to organizing files: a directory tree
    - Top-most directory is most generic, sub-directories levels become increasingly specific

STRUCTURING SOFTWARE PROJECTS

- Top directory is the project container
  - Typical software project contains source code, documentation, helper scripts, configuration samples, examples
  - Create a directory for each required

```
$ mkdir project

$ cd project

$ mkdir src docs examples

$ ls -l .
total 11
drwxr-xr-x 2 frey everyone 2 Feb 23 12:40 docs
drwxr-xr-x 2 frey everyone 2 Feb 23 12:40 examples
drwxr-xr-x 2 frey everyone 2 Feb 23 12:40 src

$
```

- On our HPC systems we like to use "s-r-c" as the directory containing source code
- Abbreviation for "source" and we always read it as "source"

STRUCTURING SOFTWARE PROJECTS

- Top directory is the project container
- Content (or presence) of *docs* and *examples* will depend on each individual project
- Let's focus on the *src* directory

STRUCTURING SOFTWARE PROJECTS

- Organization of the *src* directory should follow the component-wise structure of the project
  - A single tool program (or library) produced from one or more source files can exist as such

```
$ cd src

$ ls -l
total 5
-rw-r--r-- 1 frey everyone 108 Feb 23 12:51 my_program.c
-rw-r--r-- 1 frey everyone 195 Feb 23 12:50 printargv.c
-rw-r--r-- 1 frey everyone 105 Feb 23 12:51 printargv.h

$
```

STRUCTURING SOFTWARE PROJECTS

- Organization of the *src* directory should follow the component-wise structure of the project
  - A single tool program (or library) produced from one or more source files can exist as such
  - More complex projects should use directories to hold sub-projects

```
$ $ ls -lR .
.:
total 7
drwxr-xr-x 2 frey everyone 4 Feb 23 12:55 libprintargv
drwxr-xr-x 2 frey everyone 3 Feb 23 12:55 my_program

./libprintargv:
total 4
-rw-r--r-- 1 frey everyone 195 Feb 23 12:50 printargv.c
-rw-r--r-- 1 frey everyone 105 Feb 23 12:51 printargv.h

./my_program:
total 2
-rw-r--r-- 1 frey everyone 108 Feb 23 12:51 my_program.c
```

STRUCTURING SOFTWARE PROJECTS

- Organization of the *src* directory should follow the component-wise structure of the project
- Determining HOW to structure your source directory has many factors:
  - Size of project: the larger the code base, the more likely you can and should factor into distinct sub-projects
  - Reusability: if some of the code in the project could be used in other projects, make it a library and write an API

- I don't know HOW a particular OS implements the printf function, but the C standard I/O API tells me how to use that function. As long as I adhere to the API, my programs' use of printf doesn't care what OS I'm using.
- Creating an API usually implies a library and header files (C/C++/Fortran) or a Fortran module (mention restrictions to specific compilers, etc.)

STRUCTURING SOFTWARE PROJECTS

- Organization of the *src* directory should follow the component-wise structure of the project
- Determining HOW to structure your project has many factors
- Don't go overboard!
  - Putting every function in its own source file
  - Exposing too much in an API

- Compilers will do *inlining and interprocedural optimizations* — eliminating overhead of function calls by moving the statements into the context of the function call, reordering statements, etc. Functions not present in the same source file cannot have these optimizations performed.
- An API is meant to hide implementation details. As such, exposing any implementation details — internal organization of data structures, subroutines that may change — will break dependent software when/if the library changes its implementation.

STRUCTURING SOFTWARE PROJECTS

- What about interpreted languages like Python, Perl, PHP?
- Code modules are organized inside a directory hierarchy
  - Install in specific directory to automatically be available
- PYTHON /usr/lib64/python2.6/site-packages
- PERL /usr/lib64/perl5

STRUCTURING SOFTWARE PROJECTS

- What about interpreted languages like Python, Perl, PHP?
- Code modules are organized inside a directory hierarchy
  - Install in specific directory to automatically be available
  - Add additional search paths
- PYTHON export PYTHONPATH=~/pythonlib:${PYTHONPATH}
- PERL use lib "/home/1001/perl_lib";

STRUCTURING SOFTWARE PROJECTS

- What about interpreted languages like Python, Perl, PHP?
- Code modules are organized inside a directory hierarchy
- The three languages mentioned above include utilities to manage the distribution and installation of code modules

A BRIEF WORD RE: AN API

- An API should present an abstract interface to the world
  - Data structures or common blocks can change as a library evolves; code that uses them directly will break
    - Some are not likely to change: e.g. a point in 2D real space will always be two floating-point numbers

A FEW WORDS RE: AN API

- An API should present an abstract interface to the world
  - Data structures or common blocks can change as a library evolves; code that uses them directly will break
  - Interactions with such entities should be indirect, through *accessor* functions
    - Automate side-effects of changing a field's value
    - Internal changes don't affect consumers of the API

A FEW WORDS RE: AN API

- An API should present an abstract interface to the world
- Example: udunits
  - A C library for representation of units and conversions between them

```
typedef struct ut_system        ut_system;

EXTERNL ut_system*
ut_new_system(void);

EXTERNL void
ut_free_system(
    ut_system*  system);

EXTERNL ut_unit*
ut_get_unit_by_symbol(
    const ut_system* const      system,
    const char* const           symbol);

EXTERNL ut_status
ut_add_name_prefix(
    ut_system* const    system,
    const char* const   name,
    const double        value);

EXTERNL ut_unit*
ut_parse(
    const ut_system* const      system,
    const char* const           string,
    const ut_encoding           encoding);
```

- A rigorous naming scheme is defined and used for all types, functions, etc.
- The "ut_system" data structure isn't visible in the API
    - A function dynamically creates "ut_system" instances as opaque pointers
    - Another function dynamically destroys (frees) them
- If these details remain unchanged, a program that makes use of this API will not break when the library itself changes the internal implementation

BUILDING SOFTWARE

- You need compilers (of course)
  - A compiler's input is *source code* in language X
  - The compiler produces *object code* in machine language (numeric opcodes and operands to which the CPU responds)
  - Object code is an intermediary form. It becomes *executable code* when it is *linked*.

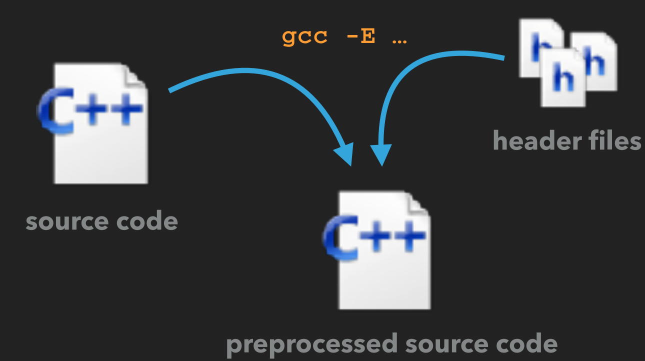
```
$ gcc -c printargv.c
$ gcc -c my_program.c
$ ls -l
total 13
-rw-r--r-- 1 frey everyone  113 Feb 23 13:19 my_program.c
-rw-r--r-- 1 frey everyone 1392 Feb 23 13:19 my_program.o
-rw-r--r-- 1 frey everyone  195 Feb 23 12:50 printargv.c
-rw-r--r-- 1 frey everyone  105 Feb 23 12:51 printargv.h
-rw-r--r-- 1 frey everyone 1560 Feb 23 13:19 printargv.o

$ gcc my_program.o printargv.o
$ ls -l
total 18
-rwxr-xr-x 1 frey everyone 6630 Feb 23 13:20 a.out
-rw-r--r-- 1 frey everyone  113 Feb 23 13:19 my_program.c
-rw-r--r-- 1 frey everyone 1392 Feb 23 13:19 my_program.o
-rw-r--r-- 1 frey everyone  195 Feb 23 12:50 printargv.c
-rw-r--r-- 1 frey everyone  105 Feb 23 12:51 printargv.h
-rw-r--r-- 1 frey everyone 1560 Feb 23 13:19 printargv.o

$ rm a.out
$ gcc -o my_program my_program.o printargv.o
$ ls -l
total 18
-rwxr-xr-x 1 frey everyone 6630 Feb 23 13:21 my_program
-rw-r--r-- 1 frey everyone  113 Feb 23 13:19 my_program.c
-rw-r--r-- 1 frey everyone 1392 Feb 23 13:19 my_program.o
-rw-r--r-- 1 frey everyone  195 Feb 23 12:50 printargv.c
-rw-r--r-- 1 frey everyone  105 Feb 23 12:51 printargv.h
-rw-r--r-- 1 frey everyone 1560 Feb 23 13:19 printargv.o
```

- The flag "-c" to the compiler emits the object code, NOT linked executable code
    - Default naming removes language-specific suffix, adds ".o"
- The executable code is produced by linking the object code
    - Default name of the linked executable is "a.out"
    - Use the "-o" option to explicitly name the output of the compiler

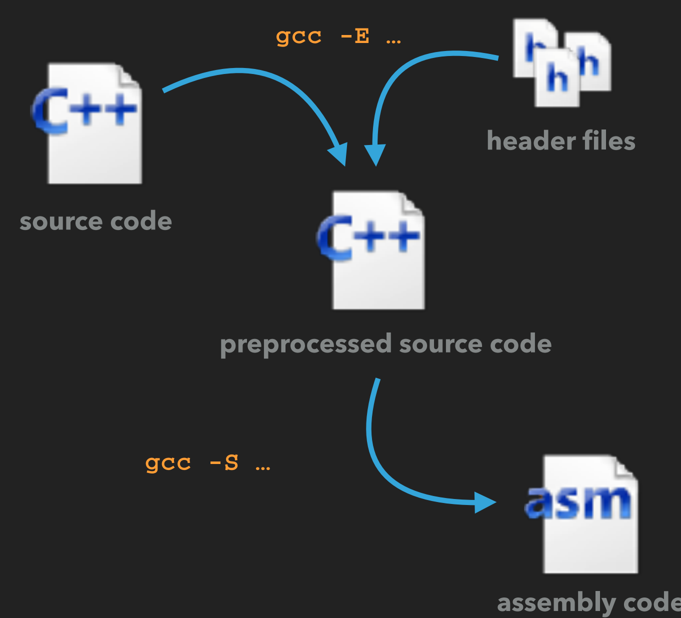
BUILDING SOFTWARE

source code

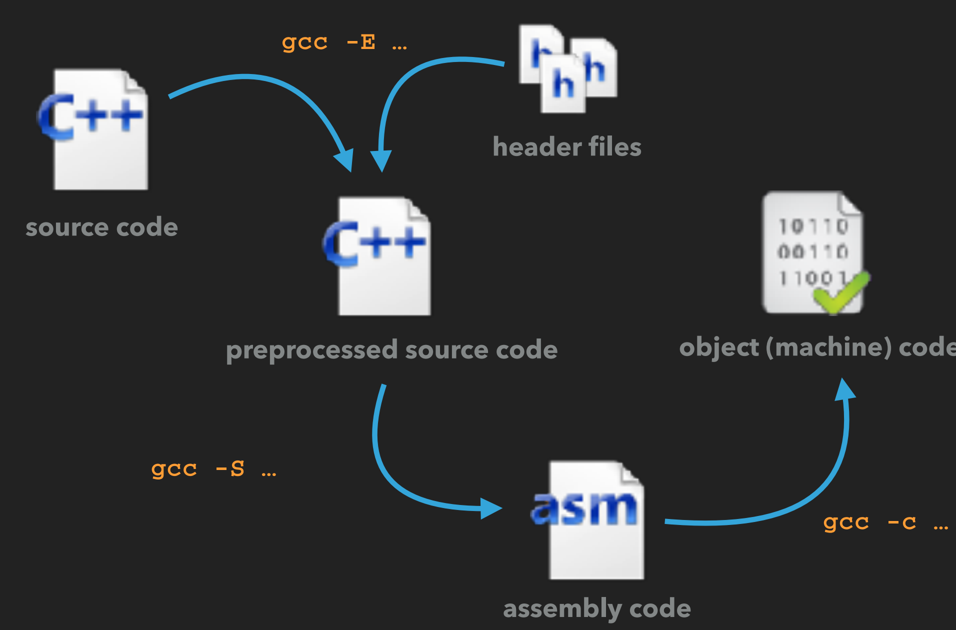
BUILDING SOFTWARE

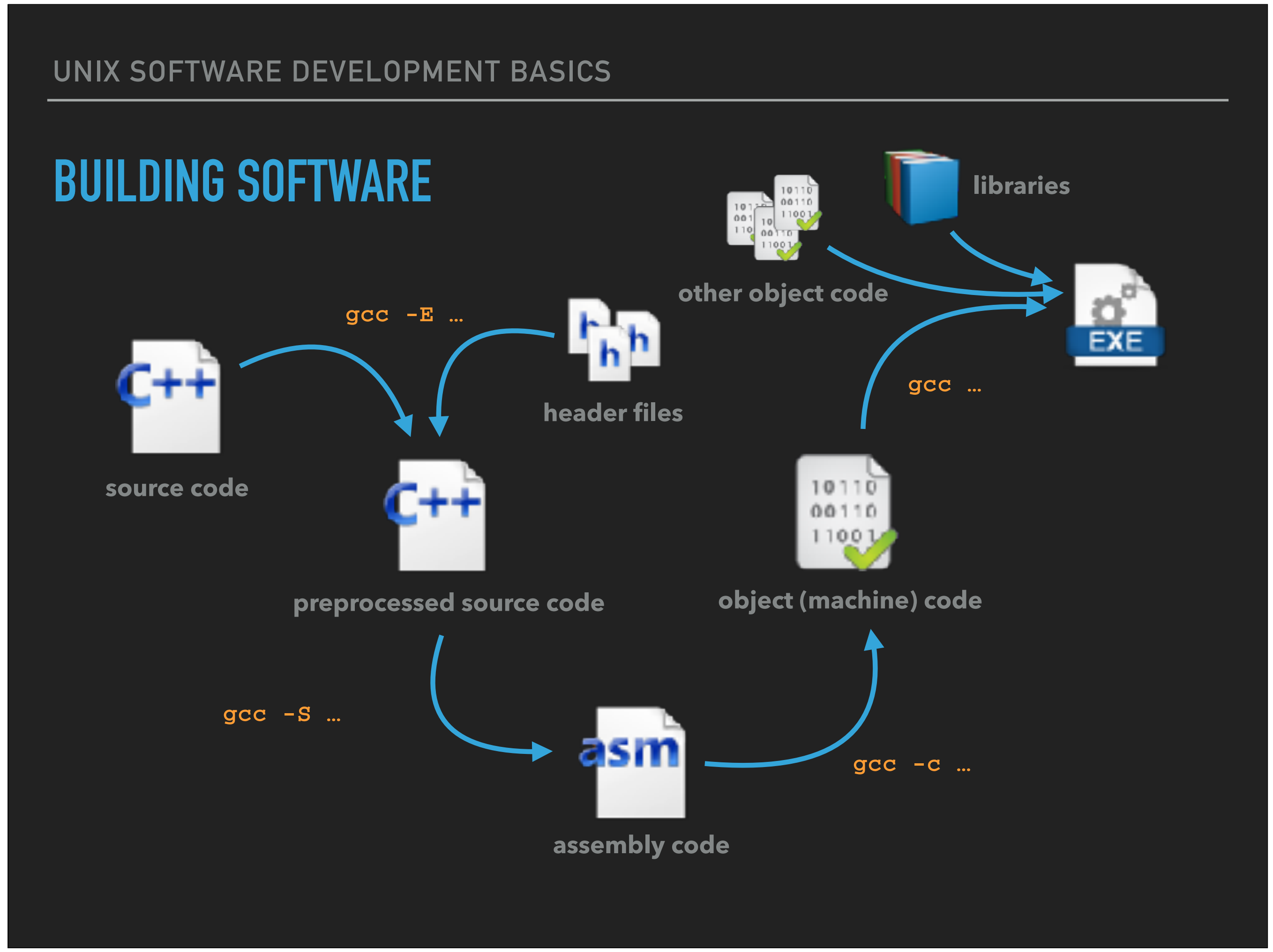


BUILDING SOFTWARE

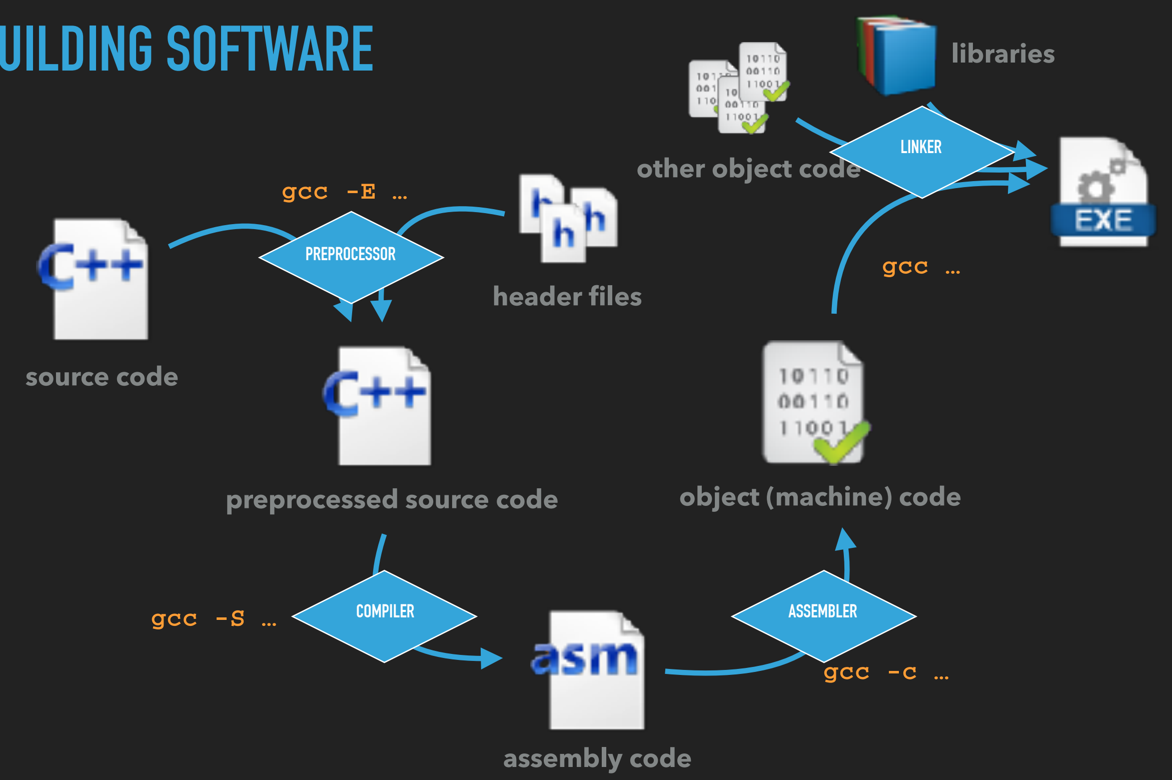


BUILDING SOFTWARE




UNIX SOFTWARE DEVELOPMENT BASICS
BUILDING SOFTWARE
libraries
other object code
gcc -E …
header files
source code
preprocessed source code
gcc …
object (machine) code
gcc -S …
asm
assembly code
gcc -c …
EXE

BUILDING SOFTWARE



BUILDING SOFTWARE

- You need compilers (of course)
- References to functions in libraries are resolved at *link*
  - My *printargv()* function calls *printf()*
  - The *printf()* function is NOT present in that source file…
  - …but the compiler happily produced the object code

```
$ cat printargv.c
    :
  while ( ++argn < argc ) {
    print("%-5d %s\n", argn, argv[argn]);
  }
    :

$ gcc -c printargv.c

$ gcc -o my_program printargv.o my_program.o
printargv.o: In function `printargv':
printargv.c:(.text+0x38): undefined reference to `print'
collect2: ld returned 1 exit status



$ gcc -o my_program my_program.o
my_program.o: In function `main':
my_program.c:(.text+0x1c): undefined reference to `printargv'
collect2: ld returned 1 exit status
```

- Let's change printargv() to call the print() function instead
- The source => object code works fine
- The link stage fails because object code for the print() function wasn't found
- Likewise, if we leave out the printargv.o object file, the link phase fails because printargv() itself cannot be found

BUILDING SOFTWARE

- You need compilers (of course)
- References to functions in libraries are resolved at *link*
  - **Q:** Wait a second, where did the compiler find the *printf()* function when it linked the executable the first time?

> **ANY SUFFICIENTLY ADVANCED TECHNOLOGY IS INDISTINGUISHABLE FROM MAGIC.**

Arthur C. Clarke's Third Law

FOR A SCIENTIST, ANY TECHNOLOGY THAT APPEARS MAGICAL SHOULD PROMPT CURIOSITY AND CAREFUL INVESTIGATION.

Frey's Corollary to the Third Law

BUILDING SOFTWARE

- You need compilers (of course)
- References to functions in libraries are resolved at *link*
  - **Q:** Wait a second, where did the compiler find the *printf()* function when it linked the executable the first time?
  - **A:** The system's C library, which the compiler implicitly added to the command line arguments I provided.

- Investigation in this case means asking the compiler to be "verbose" — print lots of information normally not seen

```
$ gcc --verbose -o my_program my_program.o printargv.o
Using built-in specs.
Target: x86_64-redhat-linux
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/
share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-bootstrap --
enable-shared --enable-threads=posix --enable-checking=release --with-system-zlib --
enable-__cxa_atexit --disable-libunwind-exceptions --enable-gnu-unique-object --
enable-languages=c,c++,objc,obj-c++,java,fortran,ada --enable-java-awt=gtk --
disable-dssi --with-java-home=/usr/lib/jvm/java-1.5.0-gcj-1.5.0.0/jre --enable-
libgcj-multifile --enable-java-maintainer-mode --with-ecj-jar=/usr/share/java/
eclipse-ecj.jar --disable-libjava-multilib --with-ppl --with-cloog --with-
tune=generic --with-arch_32=i686 --build=x86_64-redhat-linux
Thread model: posix
gcc version 4.4.7 20120313 (Red Hat 4.4.7-17) (GCC)
COMPILER_PATH=/usr/libexec/gcc/x86_64-redhat-linux/4.4.7/:/usr/libexec/gcc/x86_64-
redhat-linux/4.4.7/:/usr/libexec/gcc/x86_64-redhat-linux/:/usr/lib/gcc/x86_64-
redhat-linux/4.4.7/:/usr/lib/gcc/x86_64-redhat-linux/:/usr/libexec/gcc/x86_64-
redhat-linux/4.4.7/:/usr/libexec/gcc/x86_64-redhat-linux/:/usr/lib/gcc/x86_64-
redhat-linux/4.4.7/:/usr/lib/gcc/x86_64-redhat-linux/
LIBRARY_PATH=/usr/lib/gcc/x86_64-redhat-linux/4.4.7/:/usr/lib/gcc/x86_64-redhat-
linux/4.4.7/:/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/:/lib/../
lib64/:/usr/lib/../lib64/:/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../:/lib/:/
usr/lib/
COLLECT_GCC_OPTIONS='-v' '-o' 'my_program' '-mtune=generic'
 /usr/libexec/gcc/x86_64-redhat-linux/4.4.7/collect2 --eh-frame-hdr --build-id -m
elf_x86_64 --hash-style=gnu -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o
my_program /usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/crt1.o /usr/lib/
gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/crti.o /usr/lib/gcc/x86_64-redhat-
linux/4.4.7/crtbegin.o -L/usr/lib/gcc/x86_64-redhat-linux/4.4.7 -L/usr/lib/gcc/
x86_64-redhat-linux/4.4.7 -L/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64
-L/lib/../lib64 -L/usr/lib/../lib64 -L/usr/lib/gcc/x86_64-redhat-linux/
4.4.7/../../.. my_program.o printargv.o -lgcc --as-needed -lgcc_s --no-as-needed -lc
-lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-redhat-linux/4.4.7/
crtend.o /usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/crtn.o
```

- The "--verbose" option to gcc reveals a number of object code files and libraries that are added into my object code files to produce the executable:
  - crt1.o, crtbegin.o, crtend.o, crtn.o: standard prolog and epilog code to launch program
  - -lc, -lgcc: C runtime

BUILDING SOFTWARE

- You need compilers (of course)
- References to functions in libraries are resolved at *link*
- Libraries can be added by path, but preferable to use -l
  - -l<name> == {<path>/}lib<name>.<type>
  - The type can be:
    - "a": static object code (merged into executable)
    - "so": dynamic object code (loaded at runtime)

- Dynamic libraries are preferred, but use of static libraries can be indicated with compiler flags
- Answer to question: notice all those -L flags? They indicate paths in which the compiler looks for libraries that are referenced by -l flags
  - Mainly *standard library paths*, paths the OS knows to check: /lib, /usr/lib, /lib64
  - Any non-standard library paths will only be checked if you tell the OS to do so

BUILDING SOFTWARE

- You need compilers (of course)
- Refer **WITHOUT A FULL PATH, HOW DOES THE COMPILER FIND THE LIBRARY ASSOCIATED WITH THIS FLAG?** aries are resolved at *link*
- Librar th, but preferable to use -l
  - -l<name> == {<path>/}lib<name>.<type>
  - The type can be:
    - "a": static object code (merged into executable)
    - "so": dynamic object code (loaded at runtime)

- Dynamic libraries are preferred, but use of static libraries can be indicated with compiler flags
- Answer to question: notice all those -L flags? They indicate paths in which the compiler looks for libraries that are referenced by -l flags
  - Mainly *standard library paths*, paths the OS knows to check: /lib, /usr/lib, /lib64
  - Any non-standard library paths will only be checked if you tell the OS to do so

```
$ gcc --verbose -o my_program my_program.o printargv.o
Using built-in specs.
Target: x86_64-redhat-linux
Configured with: ../configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/
share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-bootstrap --
enable-shared --enable-threads=posix --enable-checking=release --with-system-zlib --
enable-__cxa_atexit --disable-libunwind-exceptions --enable-gnu-unique-object --
enable-languages=c,c++,objc,obj-c++,java,fortran,ada --enable-java-awt=gtk --
disable-dssi --with-java-home=/usr/lib/jvm/java-1.5.0-gcj-1.5.0.0/jre --enable-
libgcj-multifile --enable-java-maintainer-mode --with-ecj-jar=/usr/share/java/
eclipse-ecj.jar --disable-libjava-multilib --with-ppl --with-cloog --with-
tune=generic --with-arch_32=i686 --build=x86_64-redhat-linux
Thread model: posix
gcc version 4.4.7 20120313 (Red Hat 4.4.7-17) (GCC)
COMPILER_PATH=/usr/libexec/gcc/x86_64-redhat-linux/4.4.7/:/usr/libexec/gcc/x86_64-
redhat-linux/4.4.7/:/usr/libexec/gcc/x86_64-redhat-linux/:/usr/lib/gcc/x86_64-
redhat-linux/4.4.7/:/usr/lib/gcc/x86_64-redhat-linux/:/usr/libexec/gcc/x86_64-
redhat-linux/4.4.7/:/usr/libexec/gcc/x86_64-redhat-linux/:/usr/lib/gcc/x86_64-
redhat-linux/4.4.7/:/usr/lib/gcc/x86_64-redhat-linux/
LIBRARY_PATH=/usr/lib/gcc/x86_64-redhat-linux/4.4.7/:/usr/lib/gcc/x86_64-redhat-
linux/4.4.7/:/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/:/lib/../
lib64/:/usr/lib/../lib64/:/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../:/lib/:/
usr/lib/
COLLECT_GCC_OPTIONS='-v' '-o' 'my_program' '-mtune=generic'
 /usr/libexec/gcc/x86_64-redhat-linux/4.4.7/collect2 --eh-frame-hdr --build-id -m
elf_x86_64 --hash-style=gnu -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o
my_program /usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/crt1.o /usr/lib/
gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/crti.o /usr/lib/gcc/x86_64-redhat-
linux/4.4.7/crtbegin.o -L/usr/lib/gcc/x86_64-redhat-linux/4.4.7 -L/usr/lib/gcc/
x86_64-redhat-linux/4.4.7 -L/usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64
-L/lib/../lib64 -L/usr/lib/../lib64 -L/usr/lib/gcc/x86_64-redhat-linux/
4.4.7/../../.. my_program.o printargv.o -lgcc --as-needed -lgcc_s --no-as-needed -lc
-lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/x86_64-redhat-linux/4.4.7/
crtend.o /usr/lib/gcc/x86_64-redhat-linux/4.4.7/../../../../lib64/crtn.o
```

DYNAMIC VERSUS STATIC LIBRARIES

- Historically the use of static libraries and executables was preferred
  - Yields a larger executable file, but launching the program was typically faster
  - No external dependencies, executable could easily be copied from one system to another
- Use of dynamic libraries is more prevalent today
  - Much easier to replace a single dynamic library versus rebuilding every program that uses that library

- E.g. imagine the SSL library has a major security flaw
  - Many other libraries and programs use SSL routines
  - If statically linked, then all dependent software must be rebuilt, too
  - Patch the dynamic library, all dependent software is also patched

FINDING DYNAMIC LIBRARIES AT RUNTIME

- The OS needs to know where to find all of the dynamic (.so) libraries a program needs when it is run
  - By default, the OS checks standard paths: /lib, /usr/lib
  - Additional paths *can* be added into the executable itself (embedded runpaths) but they cannot be altered unless the executable is linked again
  - The LD_LIBRARY_PATH environment variable

- "Standard paths" are defined in /etc/ld.so.conf.d
- On UD clusters we prefer to use LD_LIBRARY_PATH
  - VALET alters the LD_LIBRARY_PATH for you

FROM SOURCE TO EXECUTABLE…

- "If I split my program into multiple files, I can't build the executable very easily – I need to type gcc so many times."
- Do not be tempted to write a script!
  - Serializing a sequence of gcc commands means you rebuild EVERYTHING no matter how minor the change to the source

```
$ cat build.sh
#!/bin/bash

set -x
gcc -c printargv.c
gcc -c my_program.c
gcc -o my_program printargv.o my_program.o

$ chmod +x build.sh
$ ./build.sh
+ gcc -c printargv.c
+ gcc -c my_program.c
+ gcc -o my_program printargv.o my_program.o

$ ls -lt
total 22
-rwxr-xr-x 1 frey everyone 6630 Feb 23 16:26 my_program
-rw-r--r-- 1 frey everyone 1392 Feb 23 16:26 my_program.o
-rw-r--r-- 1 frey everyone 1560 Feb 23 16:26 printargv.o
-rwxr-xr-x 1 frey everyone  103 Feb 23 16:26 build.sh
-rw-r--r-- 1 frey everyone  524 Feb 23 15:58 my_program.s
-rw-r--r-- 1 frey everyone  195 Feb 23 13:45 printargv.c
-rw-r--r-- 1 frey everyone  113 Feb 23 13:19 my_program.c
-rw-r--r-- 1 frey everyone  105 Feb 23 12:51 printargv.h
```

- What if there's an error when compiling printargv.c? Will the build stop there?

FROM SOURCE TO EXECUTABLE…

- "If I split my program into multiple files, I can't build the executable very easily – I need to type gcc so many times."
- Do not be tempted to write a script!
  - Serializing a sequence of gcc commands means you rebuild EVERYTHING no matter how minor the change to the source

```
$ #...edit printargv.c to introduce an error…

$ ./build.sh
+ gcc -c printargv.c
printargv.c: In function ‘printargv’:
printargv.c:16: error: expected declaration or statement at end of input
+ gcc -c my_program.c
+ gcc -o my_program printargv.o my_program.o

$ ls -lt
total 22
-rwxr-xr-x 1 frey everyone 6630 Feb 23 16:29 my_program
-rw-r--r-- 1 frey everyone 1392 Feb 23 16:29 my_program.o
-rw-r--r-- 1 frey everyone  191 Feb 23 16:29 printargv.c
-rw-r--r-- 1 frey everyone 1560 Feb 23 16:26 printargv.o
-rwxr-xr-x 1 frey everyone  103 Feb 23 16:26 build.sh
-rw-r--r-- 1 frey everyone  524 Feb 23 15:58 my_program.s
-rw-r--r-- 1 frey everyone  113 Feb 23 13:19 my_program.c
-rw-r--r-- 1 frey everyone  105 Feb 23 12:51 printargv.h
```

- Unless I add error checking into my script, it will rebuild everything despite encountering errors along the way

```
$ cat build.sh
#!/bin/bash

set -x

gcc -c printargv.c
if [ $? -ne 0 ]; then
  echo "ERROR:  unable to compile printargv.c"
  exit 1
fi

gcc -c my_program.c
if [ $? -ne 0 ]; then
  echo "ERROR:  unable to compile my_program.c"
  exit 1
fi

gcc -o my_program printargv.o my_program.o
if [ $? -ne 0 ]; then
  echo "ERROR:  unable to link program"
  exit 1
fi

exit 0
```

- E.g. augmented with error checking after each compile command

FROM SOURCE TO EXECUTABLE…

- Useful to have a mechanism to:
  - Describe what object code goes into an executable…
  - Describe what source code produces the object code…
  - Describe what dependencies exist between the source code, object code, and executable
- If only there were a program that given this information could just ***make*** the executable for you

FROM SOURCE TO EXECUTABLE…

YOU CAN JUST AS EASILY SUBSTITUTE "LIBRARY" FOR "EXECUTABLE" HERE

- Useful to have a mechanism t
  - Describe what object code goes into an executable…
  - Describe what source code produces the object code…
  - Describe what dependencies exist between the source code, object code, and executable
- If only there were a program that given this information could just ***make*** the executable for you

FROM SOURCE TO EXECUTABLE…

- The *make* utility uses a list of rules to transform data
  - Use the generic term "data" because *make* can be used for myriad purposes – it's not just for building software
  - *dependencies* model relationships between a product and its ingredients
    - when an ingredient is newer than a product, the product must be remade

FROM SOURCE TO EXECUTABLE…

- The *make* utility uses a list of rules to transform data
  - Use the generic term "data" because *make* can be used for myriad purposes – it's not just for building software
  - *dependencies* model relationships between a **product** and its **ingredients**
  - a **product** is "made" by means of commands associated with the rule – a **recipe**

- Recipe commands MUST be indented with a TAB character

FROM SOURCE TO EXECUTABLE…

- The
  - Us
    fo
  - *de*
    ar
  - a
    w

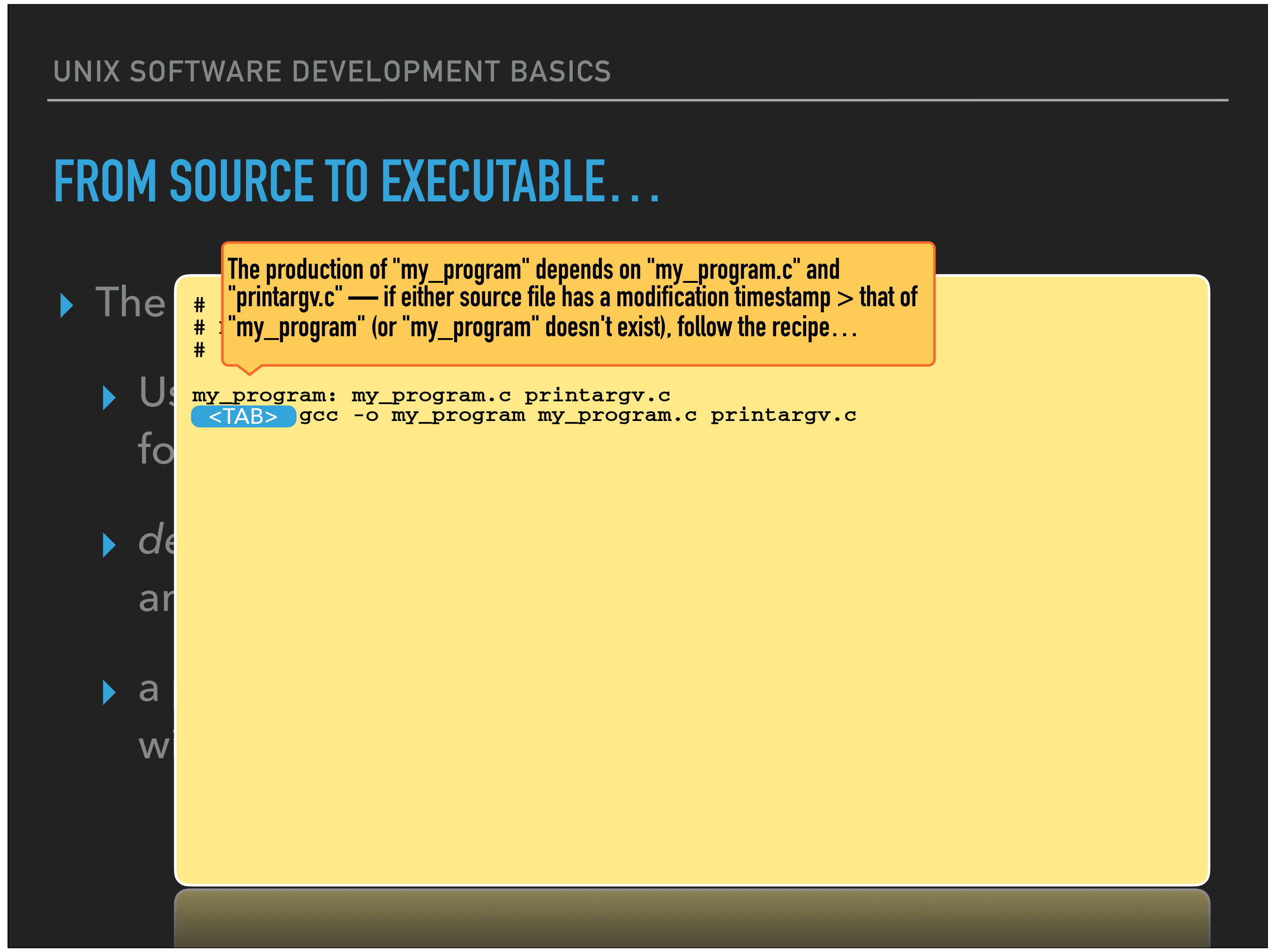
```
#
# Makefile for "my_program"
#

my_program: my_program.c printargv.c
	gcc -o my_program my_program.c printargv.c
```

- Recipe commands MUST be indented with a TAB character

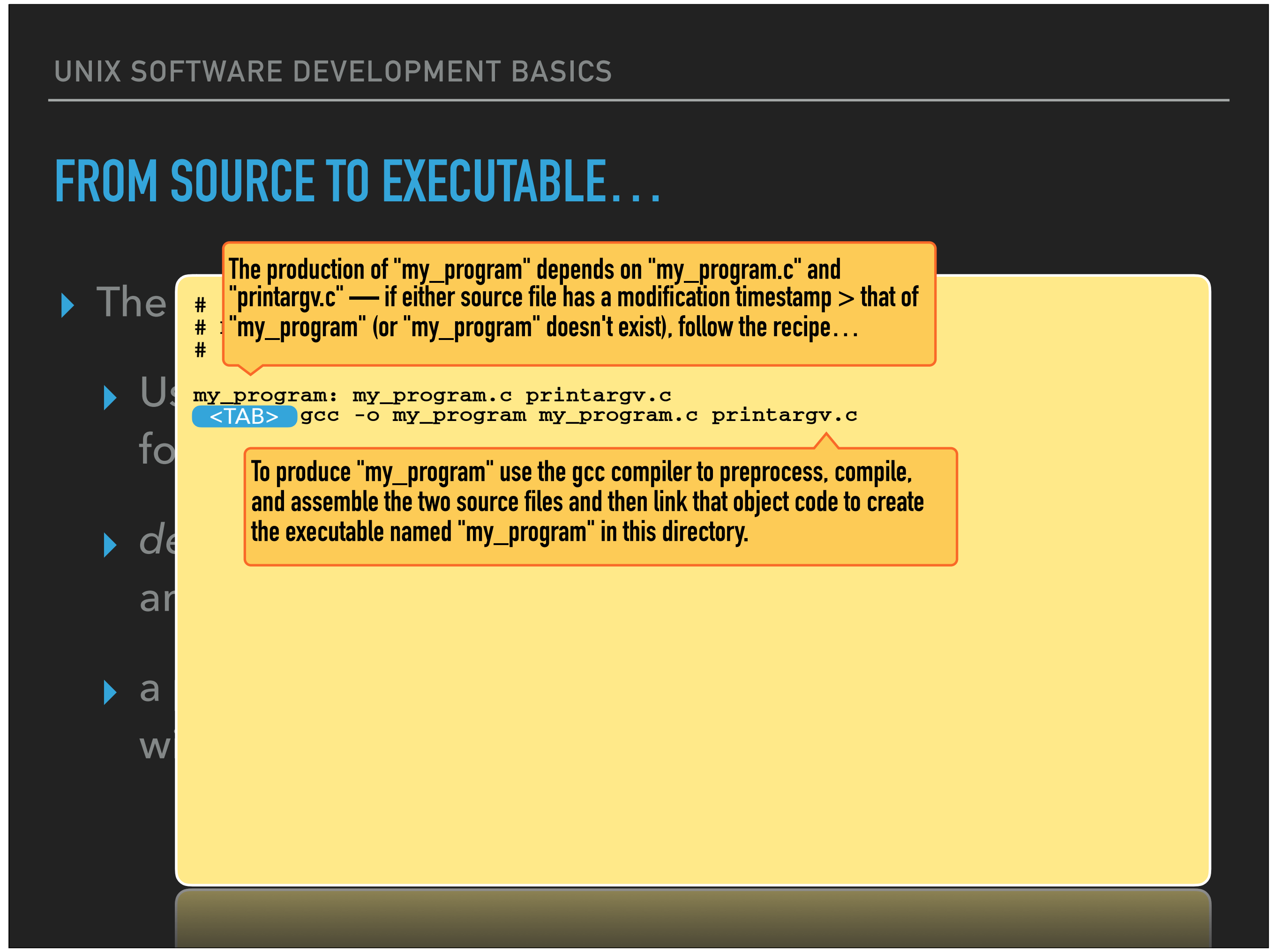
FROM SOURCE TO EXECUTABLE…

```
#
# Makefile for "my_program"
#

my_program: my_program.c printargv.c
<TAB> gcc -o my_program my_program.c printargv.c
```

- Recipe commands MUST be indented with a TAB character



- Recipe commands MUST be indented with a TAB character



- Recipe commands MUST be indented with a TAB character

FROM SOURCE TO EXECUTABLE…

- The
  - Us fo
  - *de* ar
  - a w

```
#
# Makefile for "my_program"
#

my_program: my_program.o printargv.o
        gcc -o my_program my_program.o printargv.o

my_program.o: my_program.c printargv.h
        gcc -c my_program.c

printargv.o: printargv.c printargv.h
        gcc -c printargv.c
```

- Henceforth it should be understood that file modification timestamps are used by make to determine when a product must be remade
- The first rule found in the file is the default rule

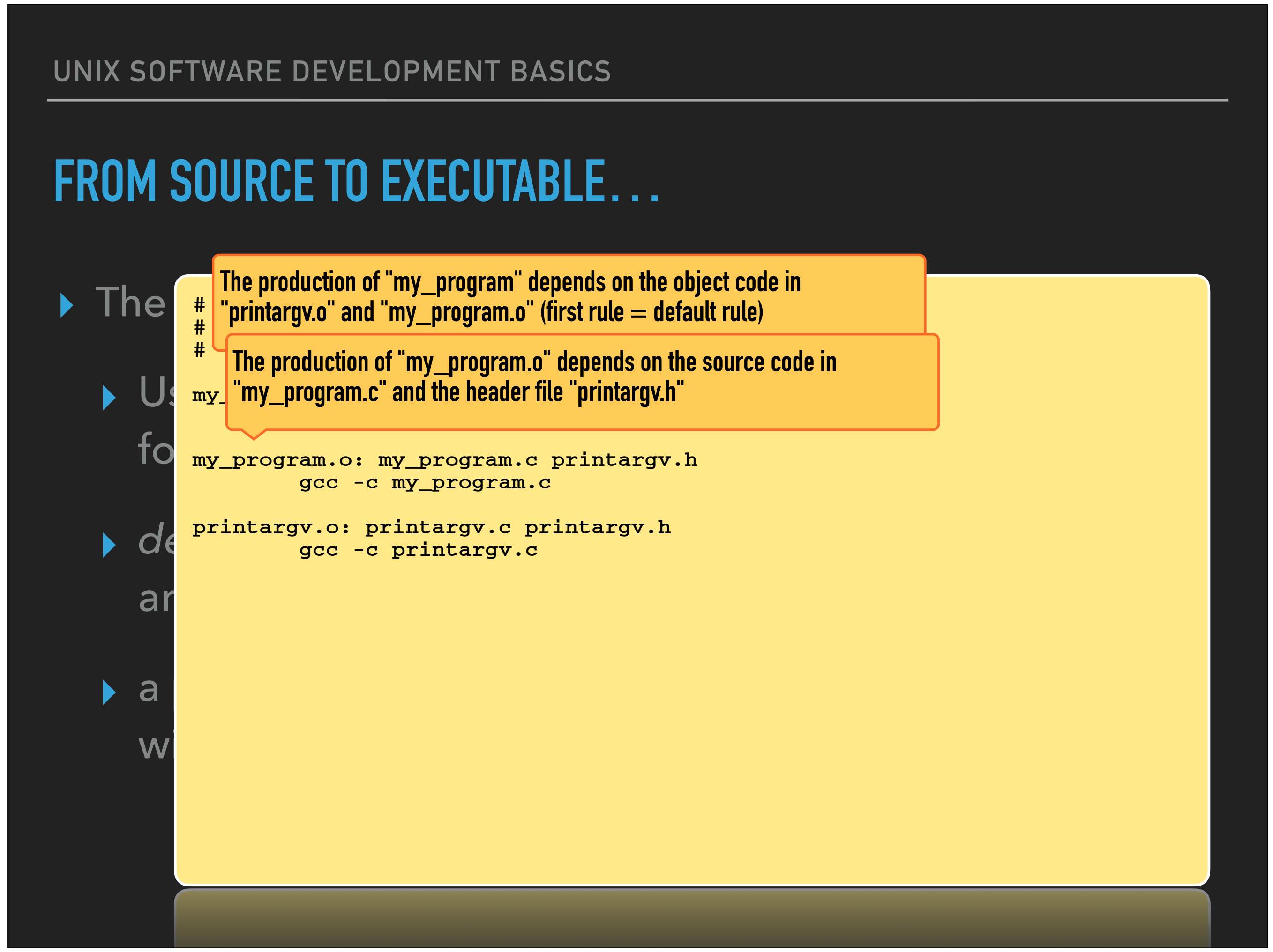
FROM SOURCE TO EXECUTABLE…

- The
  - Us fo
  - *de* ar
  - a w

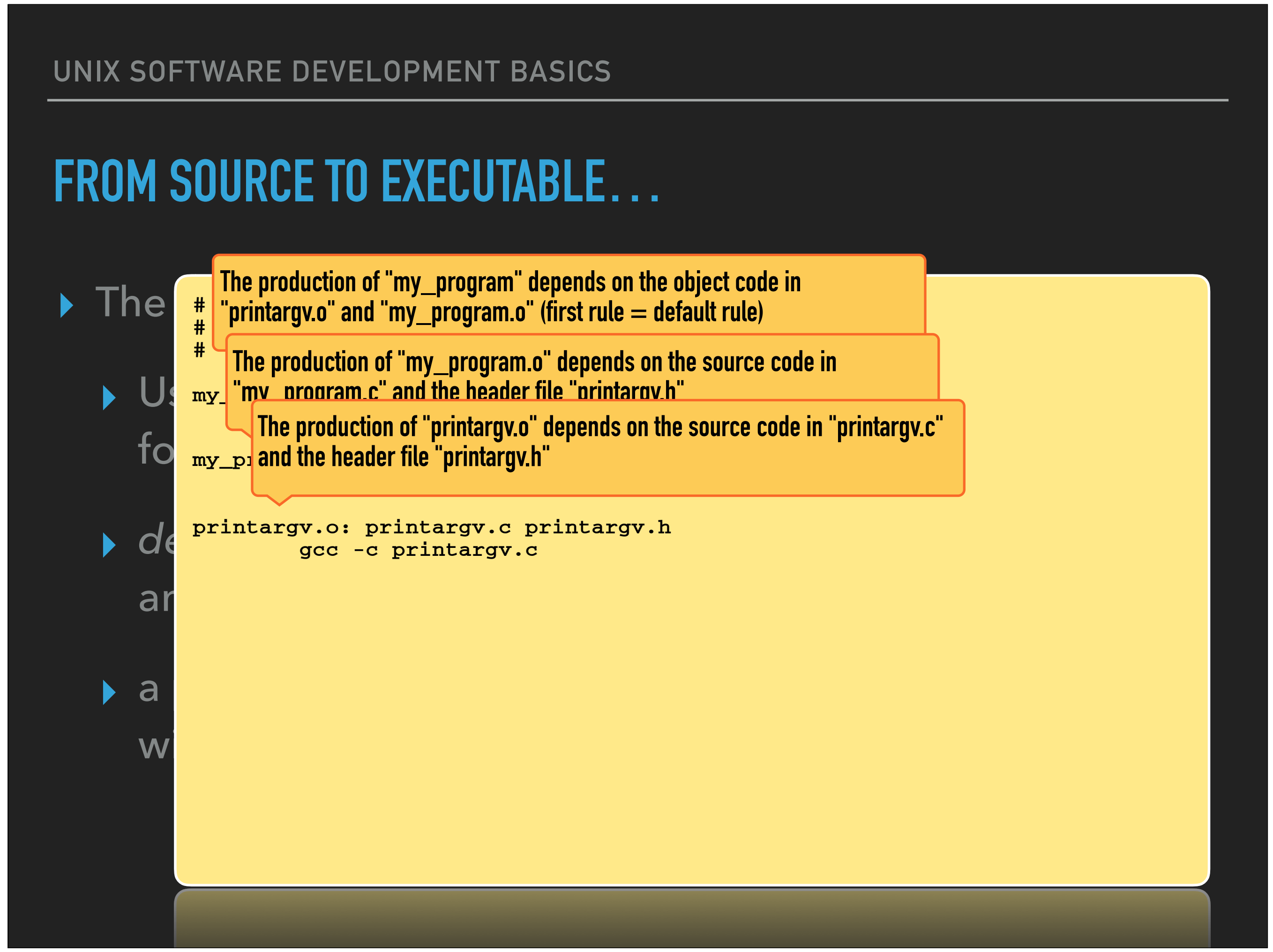
> The production of "my_program" depends on the object code in "printargv.o" and "my_program.o" (first rule = default rule)

```
#
#
#

my_program: my_program.o printargv.o
        gcc -o my_program my_program.o printargv.o

my_program.o: my_program.c printargv.h
        gcc -c my_program.c

printargv.o: printargv.c printargv.h
        gcc -c printargv.c
```

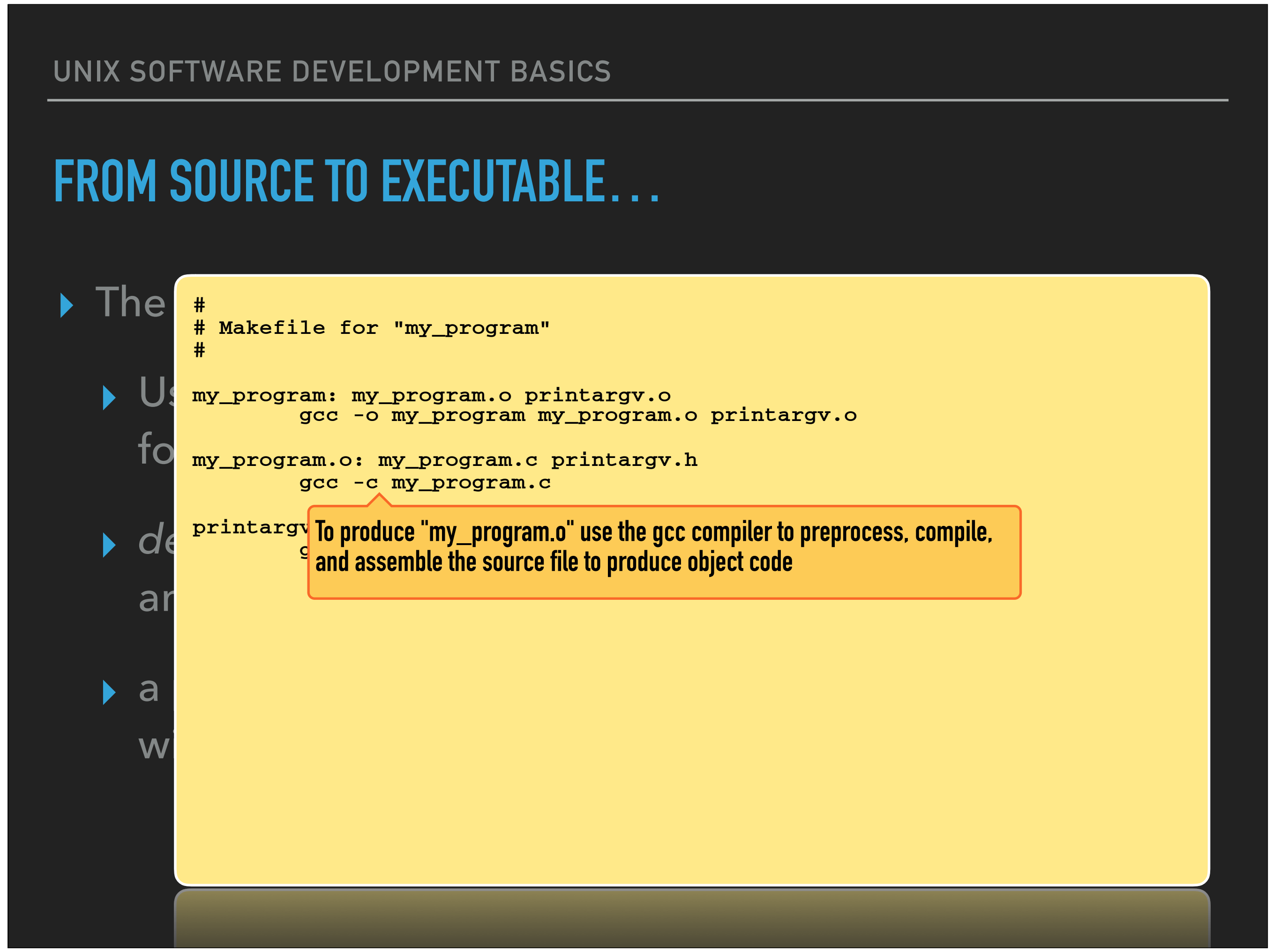
- Henceforth it should be understood that file modification timestamps are used by make to determine when a product must be remade
- The first rule found in the file is the default rule



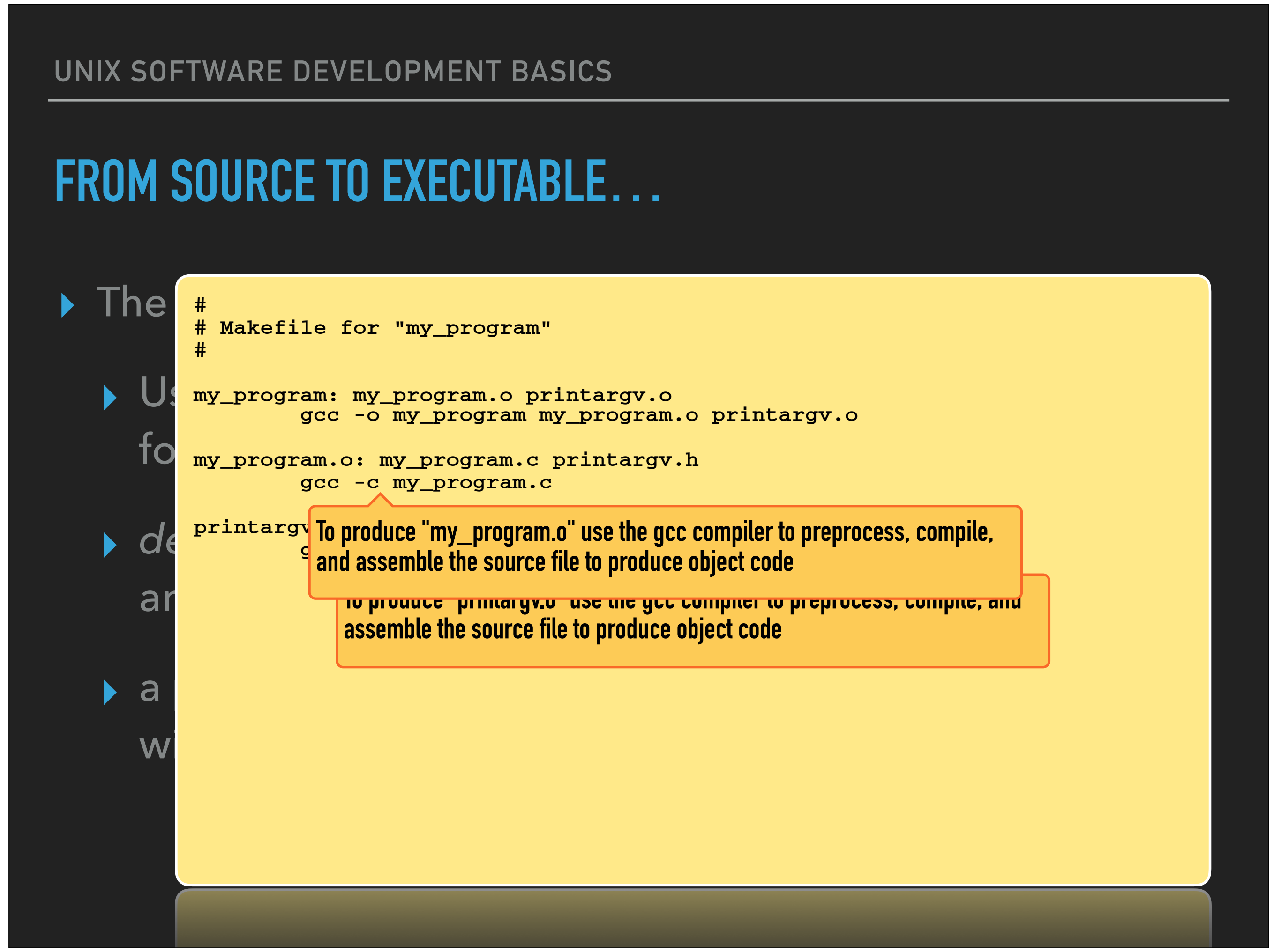
- Henceforth it should be understood that file modification timestamps are used by make to determine when a product must be remade
- The first rule found in the file is the default rule



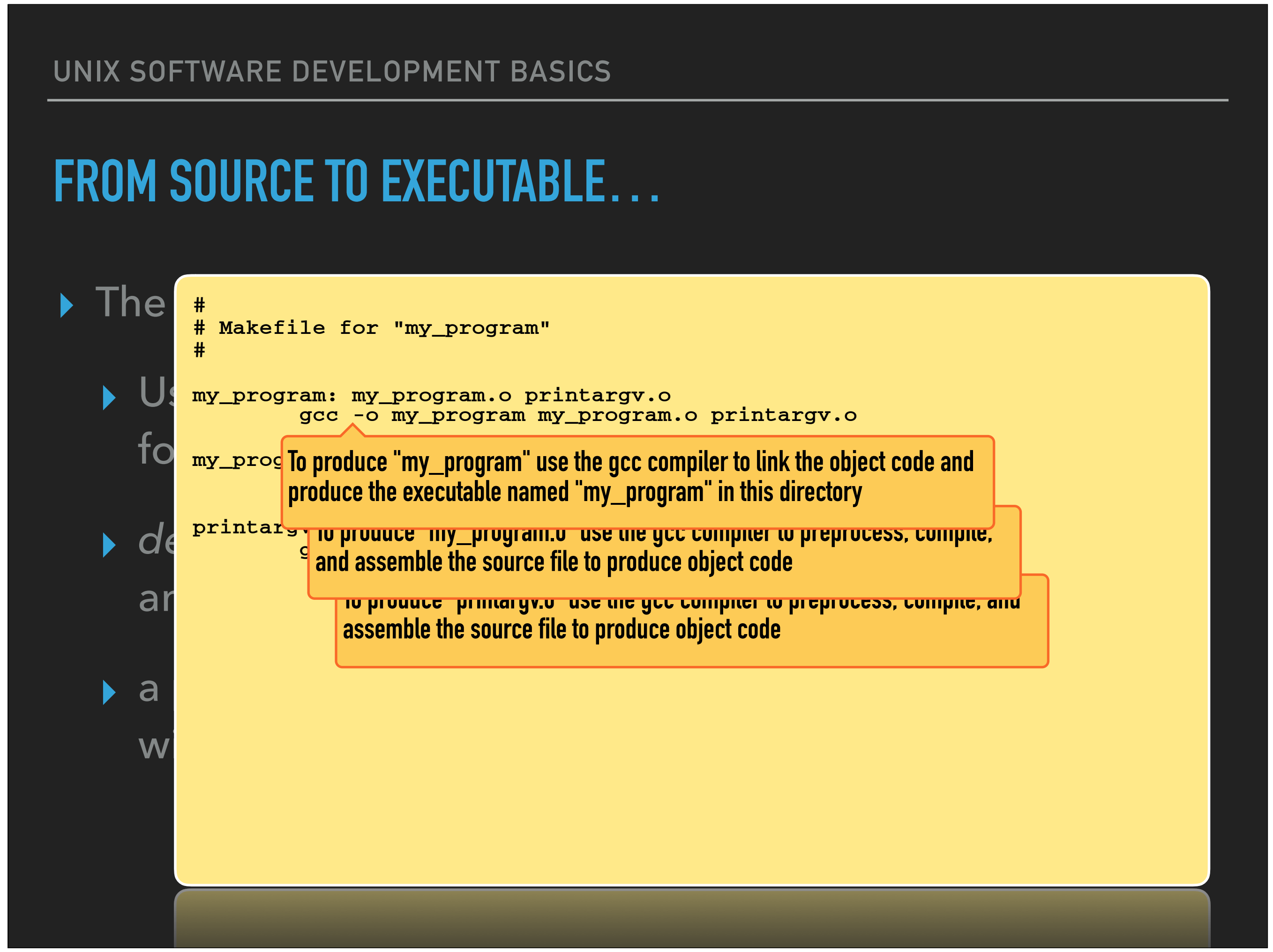
- Henceforth it should be understood that file modification timestamps are used by make to determine when a product must be remade
- The first rule found in the file is the default rule



- Henceforth it should be understood that file modification timestamps are used by make to determine when a product must be remade
- The first rule found in the file is the default rule



- Henceforth it should be understood that file modification timestamps are used by make to determine when a product must be remade
- The first rule found in the file is the default rule



- Henceforth it should be understood that file modification timestamps are used by make to determine when a product must be remade
- The first rule found in the file is the default rule

FROM SOURCE TO EXECUTABLE…

- The
  - Us
    fo
  - *de*
    ar
  - a
    w

```
#
# Makefile for "my_program"
#

TARGET        = my_program

OBJECTS       = printargv.o my_program.o

$(TARGET): $(OBJECTS)
        gcc -o $(TARGET) $(OBJECTS)

my_program.o: my_program.c printargv.h
        gcc -c my_program.c

printargv.o: printargv.c printargv.h
        gcc -c printargv.c
```

FROM SOURCE TO EXECUTABLE…

- The
  - Us fo
  - *de* ar
  - a w

```
#
# Makefile for "my
#

TARGET        = my_program

OBJECTS       = printargv.o my_program.o

$(TARGET): $(OBJECTS)
        gcc -o $(TARGET) $(OBJECTS)

my_program.o: my_program.c printargv.h
        gcc -c my_program.c

printargv.o: printargv.c printargv.h
        gcc -c printargv.c
```

Associate the string "my_program" with the variable named TARGET

FROM SOURCE TO EXECUTABLE…

- The
  - Us fo
  - *de* ar
  - a w

```
#
# Makefile for "my
#

TARGET        = my_p

OBJECTS       = printargv.o my_program.o

$(TARGET): $(OBJECTS)
        gcc -o $(TARGET) $(OBJECTS)

my_program.o: my_program.c printargv.h
        gcc -c my_program.c

printargv.o: printargv.c printargv.h
        gcc -c printargv.c
```

Associate the string "my_program" with the variable named TARGET

Associate a list of two strings with the variable named OBJECTS

FROM SOURCE TO EXECUTABLE…

- The
  - Us
    fo
  - *de*
    ar
  - a
    w

```
#
# Makefile for "my
#

TARGET        = my_p

OBJECTS       = printargv.o my_program.o

$(TARGET): $(OBJECTS)
        gcc -o $(TARGET) $(OBJECTS)

my
        gcc -c my_program.c

printargv.o: printargv.c printargv.h
        gcc -c printargv.c
```

Associate the string "my_program" with the variable named TARGET

Associate a list of two strings with the variable named OBJECTS

Evaluate the value of the variable TARGET, use that as the product side of this rule.

FROM SOURCE TO EXECUTABLE…

- The
  - Us
    fo
  - *de*
    ar
  - a
    w

```
#
# Makefile for "my
#

TARGET        = my_p

OBJECTS       = printargv.o my_program.o

$(TARGET): $(OBJECTS)
        gcc -o $(TARGET) $(OBJECTS)

my_program.o:
        gcc -c my_program.c

printargv.o: printargv.c printargv.h
        gcc -c printargv.c
```

Associate the string "my_program" with the variable named TARGET

Associate a list of two strings with the variable named OBJECTS

Evaluate the value of the variable OBJECTS, use that as the ingredient side of this rule.

FROM SOURCE TO EXECUTABLE…

- The
  - Us fo
  - *de* ar
  - a w

```
#
# Makefile for "my
#

TARGET        = my_p

OBJECTS       = printargv.o my_program.o

$(TARGET): $(OBJECTS)
        gcc -o $(TARGET) $(OBJECTS)

my_progra
        g

printargv
        g
```

Associate the string "my_program" with the variable named TARGET

Associate a list of two strings with the variable named OBJECTS

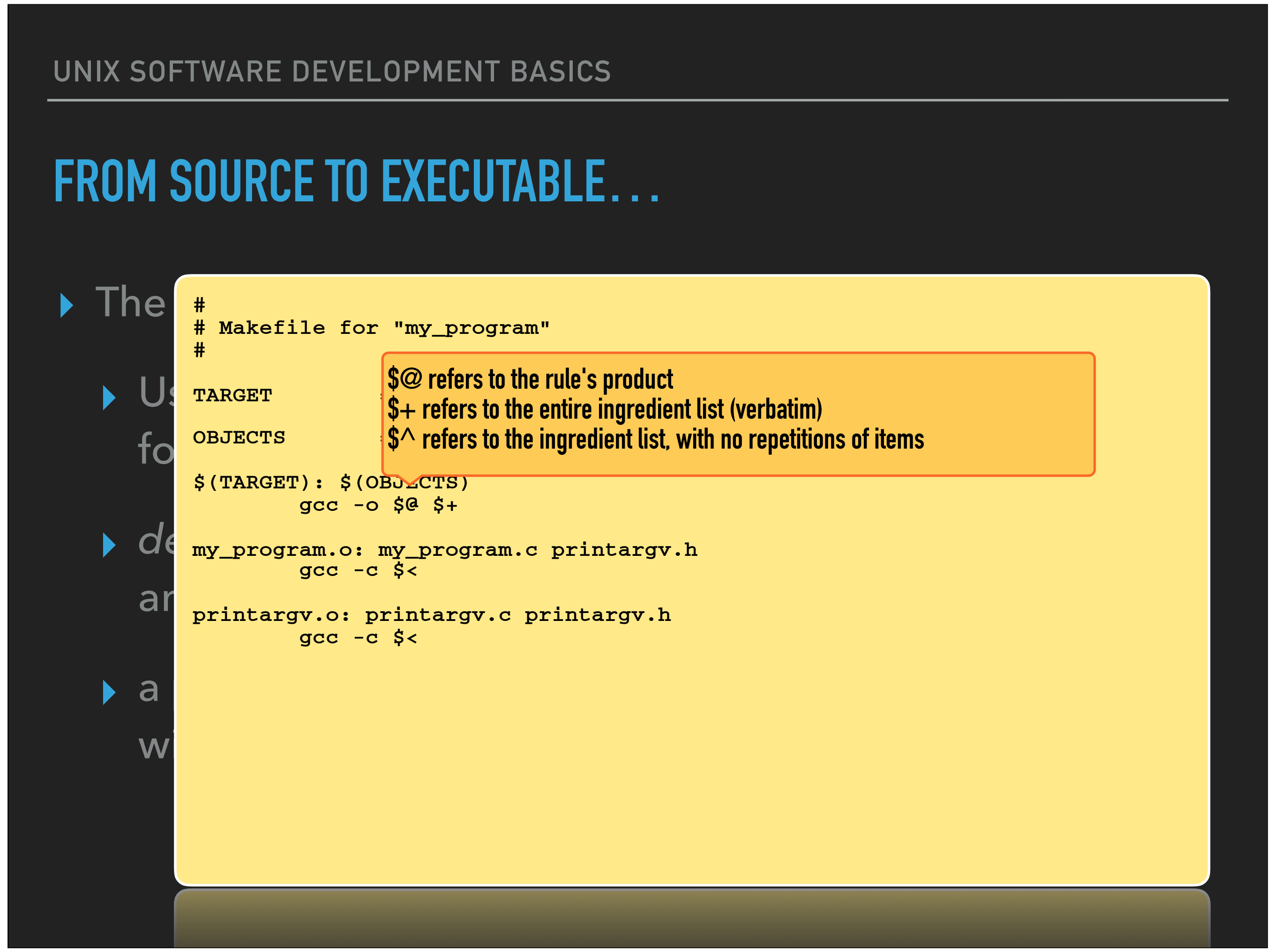
The recipe here uses the value of our two variables: if I modify the value assigned to TARGET or OBJECTS at the top of the makefile, I affect both the rules and the recipes associated with those rules. This makes it easy to add more source files to the project (additional object code names in OBJECTS) or rename the executable produced.

FROM SOURCE TO EXECUTABLE…

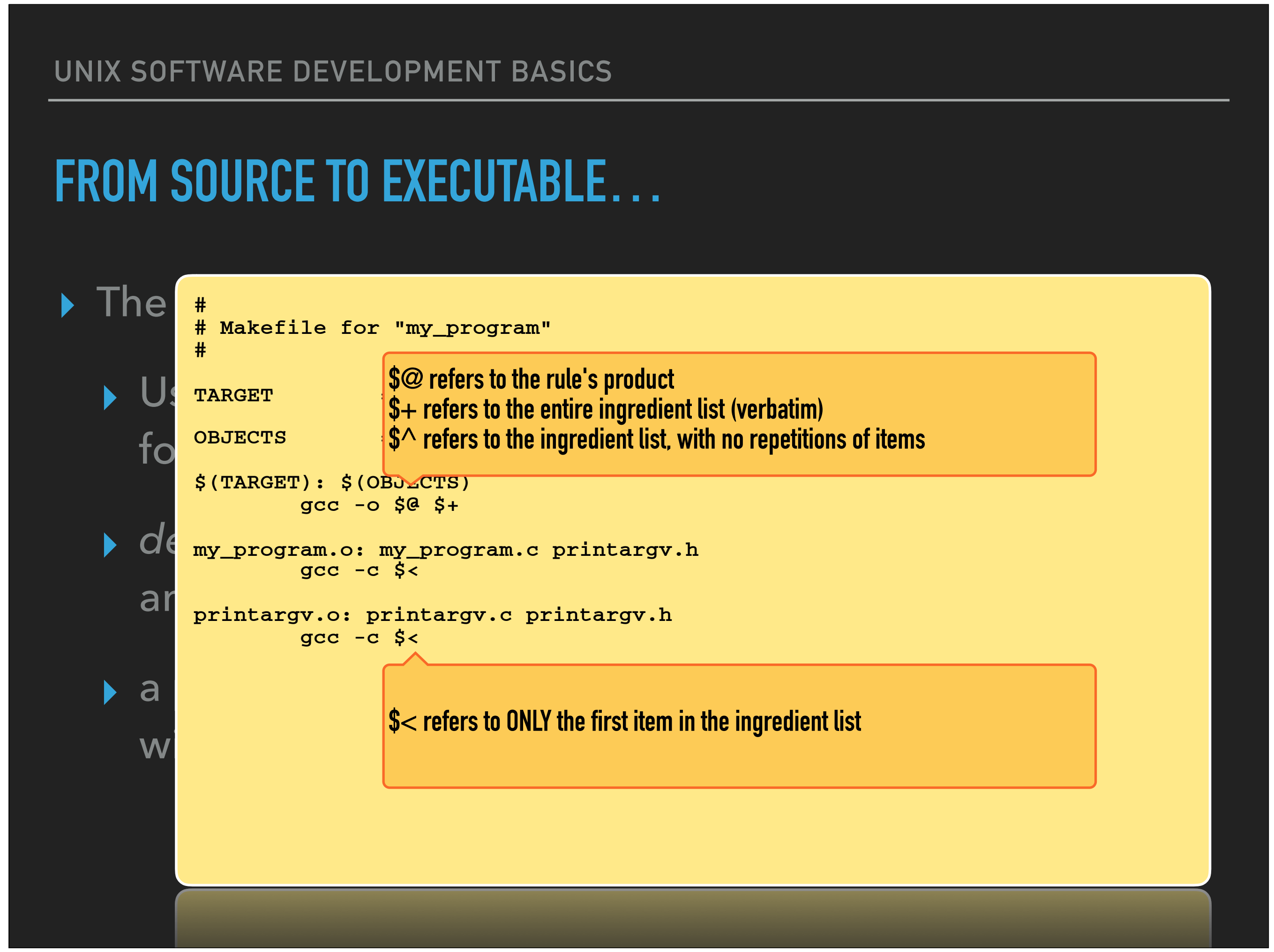
- The
  - Us fo
  - *de* ar
  - a w

```
#
# Makefile for "my_program"
#

TARGET        = my_program

OBJECTS       = printargv.o my_program.o

$(TARGET): $(OBJECTS)
        gcc -o $@ $+

my_program.o: my_program.c printargv.h
        gcc -c $<

printargv.o: printargv.c printargv.h
        gcc -c $<
```

- Automatic variables refer to the product and ingredient lists associated with a recipe



- Automatic variables refer to the product and ingredient lists associated with a recipe



- Automatic variables refer to the product and ingredient lists associated with a recipe

FROM SOURCE TO EXECUTABLE…

- The
  - Us fo
  - *de* ar
  - a w

```
#
# Makefile for "my_program"
#

TARGET        = my_program

OBJECTS       = printargv.o my_program.o

$(TARGET): $(OBJECTS)
        gcc -o $@ $+ $(LDFLAGS) $(LIBS)

my_program.o: my_program.c printargv.h

printargv.o: printargv.c printargv.h

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

- Rules with no recipe simply outline the dependencies
- Wildcard rules with a recipe actually create the product

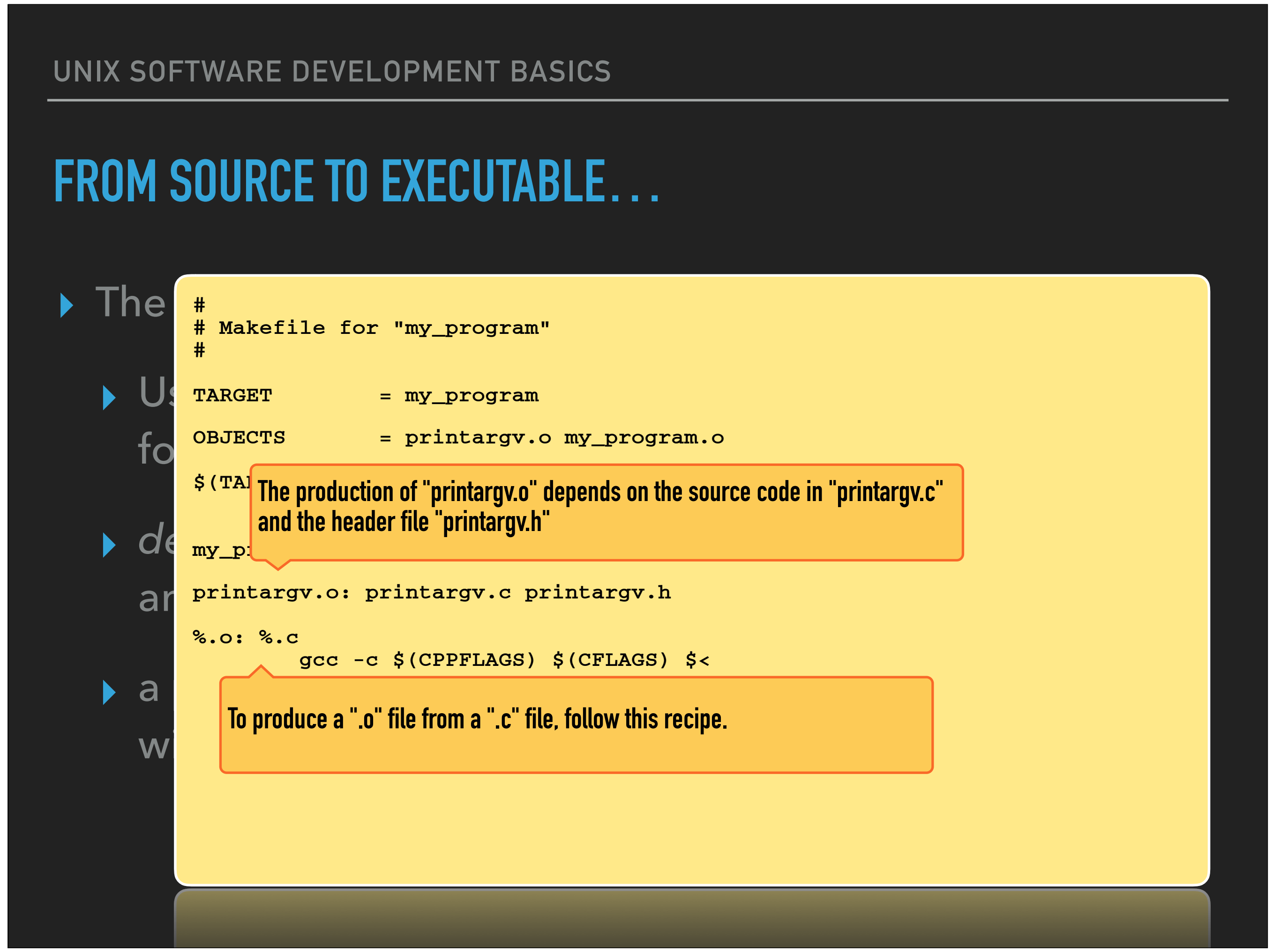
FROM SOURCE TO EXECUTABLE…

- The
  - Us
    fo
  - *de*
    ar
  - a
    w

```
#
# Makefile for "my_program"
#

TARGET        = my_program

OBJECTS       = printargv.o my_program.o

$(TA

my_pr

printargv.o: printargv.c printargv.h

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

The production of "printargv.o" depends on the source code in "printargv.c" and the header file "printargv.h"

- Rules with no recipe simply outline the dependencies
- Wildcard rules with a recipe actually create the product



- Rules with no recipe simply outline the dependencies
- Wildcard rules with a recipe actually create the product

FROM SOURCE TO EXECUTABLE…

- The
  - Us fo
  - *de* ar
  - a w

```
#
# Makefile for "my_program"
#

TARGET        = my_program

OBJECTS       = printargv.o my_program.o

default: $(TARGET)

clean:
	$(RM) $(TARGET) $(OBJECTS)

#

$(TARGET): $(OBJECTS)
	gcc -o $@ $+ $(LDFLAGS) $(LIBS)

my_program.o: my_program.c printargv.h

printargv.o: printargv.c printargv.h

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

- A "clean" rule is typically present to remove all intermediates and products
- A "default" rule can be useful to ensure the appropriate rule is indicated despite future editing of the rest of the makefile

FROM SOURCE TO EXECUTABLE…

- The
  - Us fo
  - *de* ar
  - a w

```
#
# Makefile for "my_program"
#

TAR
OBJ

default: $(TARGET)

clean:
        $(RM) $(TARGET) $(OBJECTS)

#

$(TARGET): $(OBJECTS)
        gcc -o $@ $+ $(LDFLAGS) $(LIBS)

my_program.o: my_program.c printargv.h

printargv.o: printargv.c printargv.h

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

Explicitly define what the default rule should be by placing this rule at the top of the file. Changes to the rest of the file will not interfere.

- A "clean" rule is typically present to remove all intermediates and products
- A "default" rule can be useful to ensure the appropriate rule is indicated despite future editing of the rest of the makefile

FROM SOURCE TO EXECUTABLE…

- The
  - U
    fo
  - *de*
    ar
  - a
    w

```
#
# Makefile for "my_program"
#

TAR

OBJ

defa

clean:
        $(RM) $(TARGET) $(OBJECTS)

#

$(TARGET): $(OBJECTS)
        gcc -o $@ $+ $(LDFLAGS) $(LIBS)

my_program.o: my_program.c printargv.h

printargv.o: printargv.c printargv.h

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

Explicitly define what the default rule should be by placing this rule at the

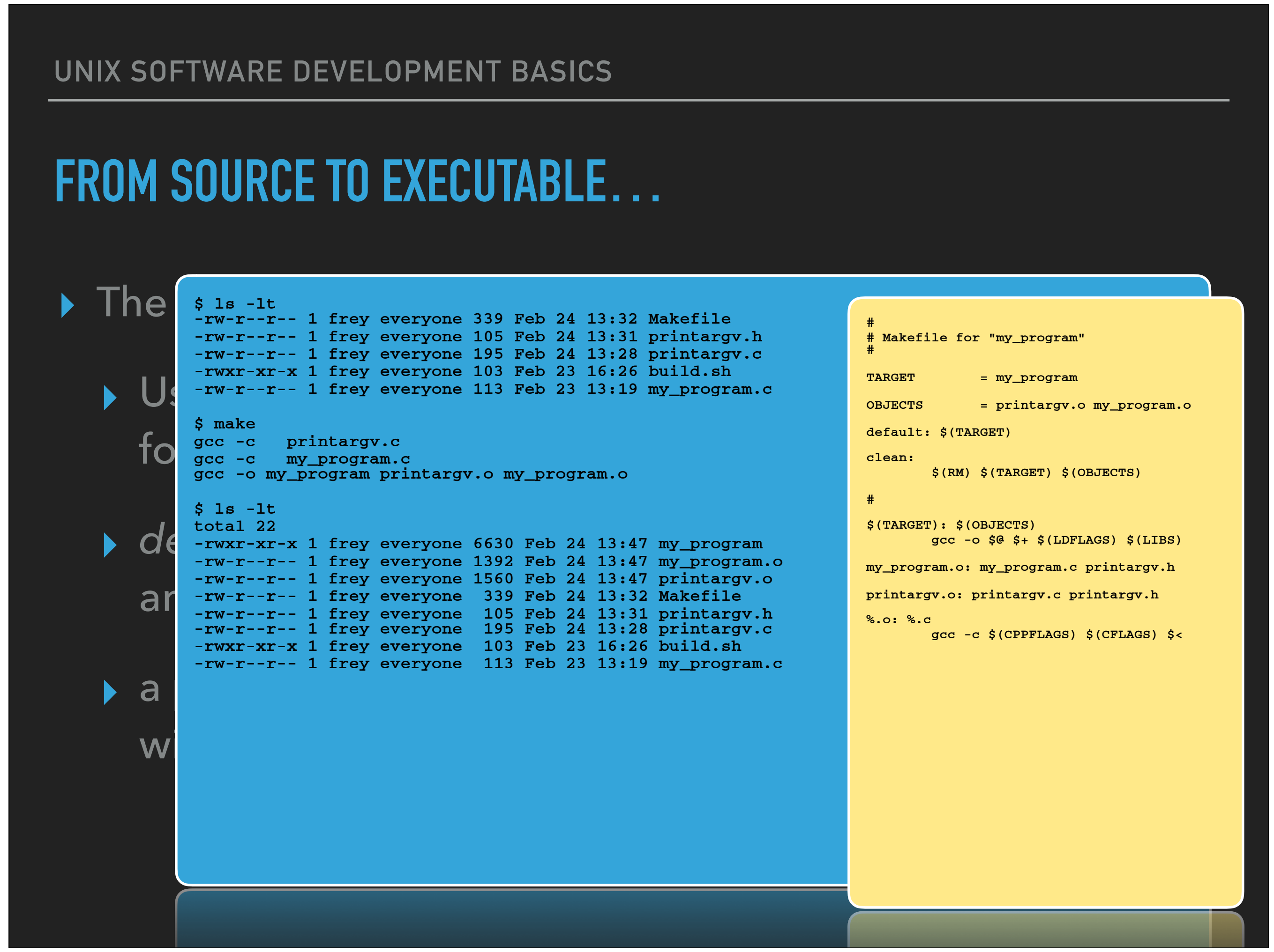
A rule is usually added that removes all intermediates (object code files) and the target itself. (Avoid naming your executable "clean" ...)

- A "clean" rule is typically present to remove all intermediates and products
- A "default" rule can be useful to ensure the appropriate rule is indicated despite future editing of the rest of the makefile

FROM SOURCE TO EXECUTABLE…

```
$ ls -lt
-rw-r--r-- 1 frey everyone 339 Feb 24 13:32 Makefile
-rw-r--r-- 1 frey everyone 105 Feb 24 13:31 printargv.h
-rw-r--r-- 1 frey everyone 195 Feb 24 13:28 printargv.c
-rwxr-xr-x 1 frey everyone 103 Feb 23 16:26 build.sh
-rw-r--r-- 1 frey everyone 113 Feb 23 13:19 my_program.c

$ make
gcc -c   printargv.c
gcc -c   my_program.c
gcc -o my_program printargv.o my_program.o

$ ls -lt
total 22
-rwxr-xr-x 1 frey everyone 6630 Feb 24 13:47 my_program
-rw-r--r-- 1 frey everyone 1392 Feb 24 13:47 my_program.o
-rw-r--r-- 1 frey everyone 1560 Feb 24 13:47 printargv.o
-rw-r--r-- 1 frey everyone  339 Feb 24 13:32 Makefile
-rw-r--r-- 1 frey everyone  105 Feb 24 13:31 printargv.h
-rw-r--r-- 1 frey everyone  195 Feb 24 13:28 printargv.c
-rwxr-xr-x 1 frey everyone  103 Feb 23 16:26 build.sh
-rw-r--r-- 1 frey everyone  113 Feb 23 13:19 my_program.c
```

- The default rule requires the two object code files; they aren't present, so they must be made
- printargv.o depends on printargv.c and printargv.h — but since the product doesn't exist, this doesn't matter this time
- The wildcard rule is a match: produce printargv.o using printargv.c
- Same for my_program.o
- Dependencies for default rule are ready, now do its recipe: produce the target (executable) my_program



- The default rule requires the two object code files; they aren't present, so they must be made
- printargv.o depends on printargv.c and printargv.h — but since the product doesn't exist, this doesn't matter this time
- The wildcard rule is a match: produce printargv.o using printargv.c
- Same for my_program.o
- Dependencies for default rule are ready, now do its recipe: produce the target (executable) my_program

FROM SOURCE TO EXECUTABLE…

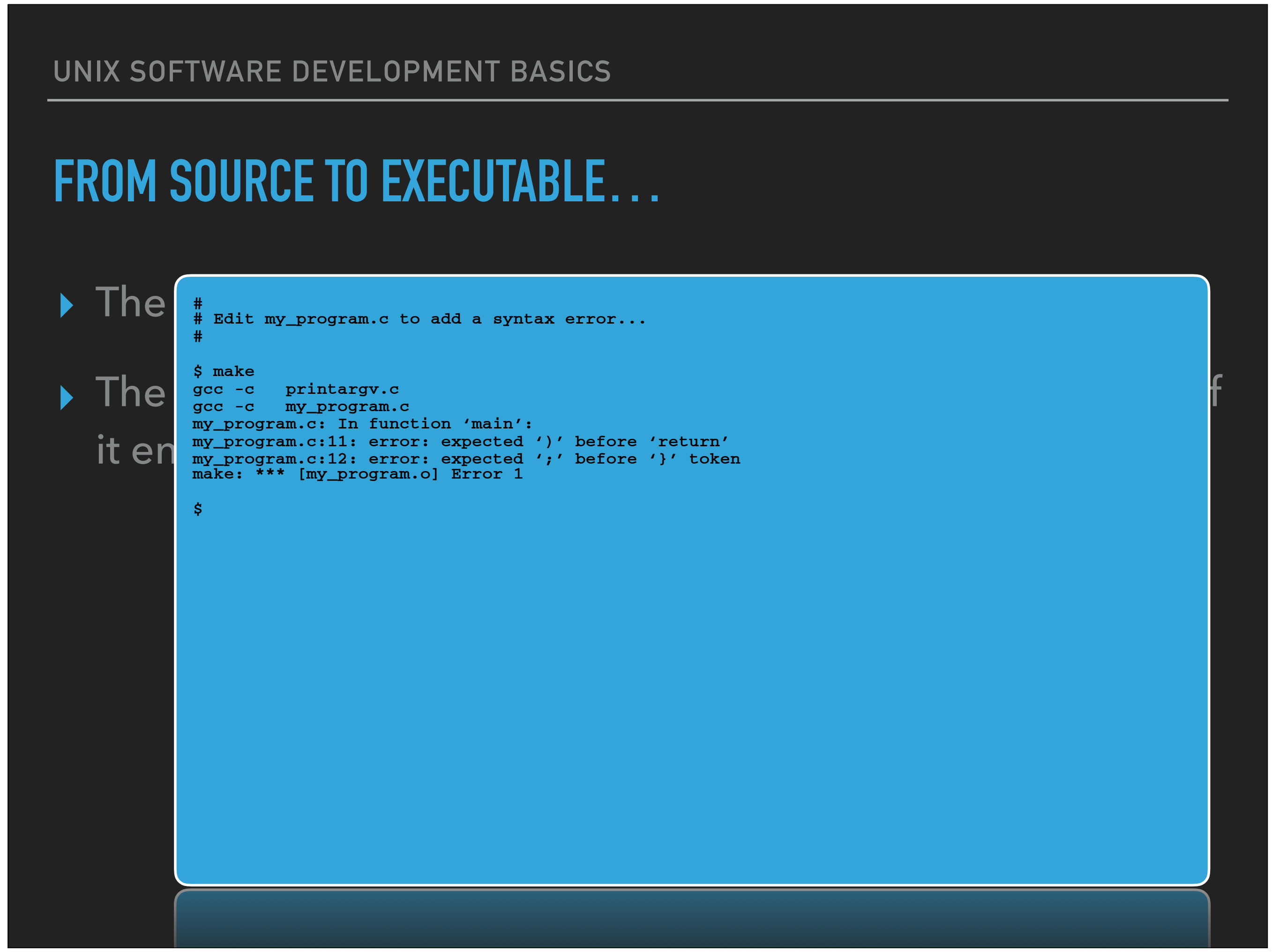
- The
  - Us
    fo
  - *de*
    ar
  - a
    w

```
$ ls -lt
-rw-r--r-- 1 frey everyone 339 Feb 24 13:32 Makefile
-rw-r--r-- 1 frey everyone 105 Feb 24 13:31 printargv.h
-rw-r--r-- 1 frey everyone 195 Feb 24 13:28 printargv.c
-rwxr-xr-x 1 frey everyone 103 Feb 23 16:26 build.sh
-rw-r--r-- 1 frey everyone 113 Feb 23 13:19 my_program.c

$ make
gcc -c   printargv.c
gcc -c   my_program.c
gcc -o my_program printargv.o my_program.o

$ ls -lt
total 22
-rwxr-xr-x 1 frey everyone 6630 Feb 24 13:47 my_program
-rw-r--r-- 1 frey everyone 1392 Feb 24 13:47 my_program.o
-rw-r--r-- 1 frey everyone 1560 Feb 24 13:47 printargv.o
-rw-r--r-- 1 frey everyone  339 Feb 24 13:32 Makefile
-rw-r--r-- 1 frey everyone  105 Feb 24 13:31 printargv.h
-rw-r--r-- 1 frey everyone  195 Feb 24 13:28 printargv.c
-rwxr-xr-x 1 frey everyone  103 Feb 23 16:26 build.sh
-rw-r--r-- 1 frey everyone  113 Feb 23 13:19 my_program.c

$ make clean
rm -f my_program printargv.o my_program.o

$ ls -lt
total 10
-rw-r--r-- 1 frey everyone 339 Feb 24 13:32 Makefile
-rw-r--r-- 1 frey everyone 105 Feb 24 13:31 printargv.h
-rw-r--r-- 1 frey everyone 195 Feb 24 13:28 printargv.c
-rwxr-xr-x 1 frey everyone 103 Feb 23 16:26 build.sh
-rw-r--r-- 1 frey everyone 113 Feb 23 13:19 my_program.c
```

FROM SOURCE TO EXECUTABLE…

- The *make* utility uses a list of rules to transform data
- The *make* utility handles error-checking for you, stopping if it encounters any errors

- make checks the return code from each command in a recipe; non-zero implies error, causes it to stop what it's doing



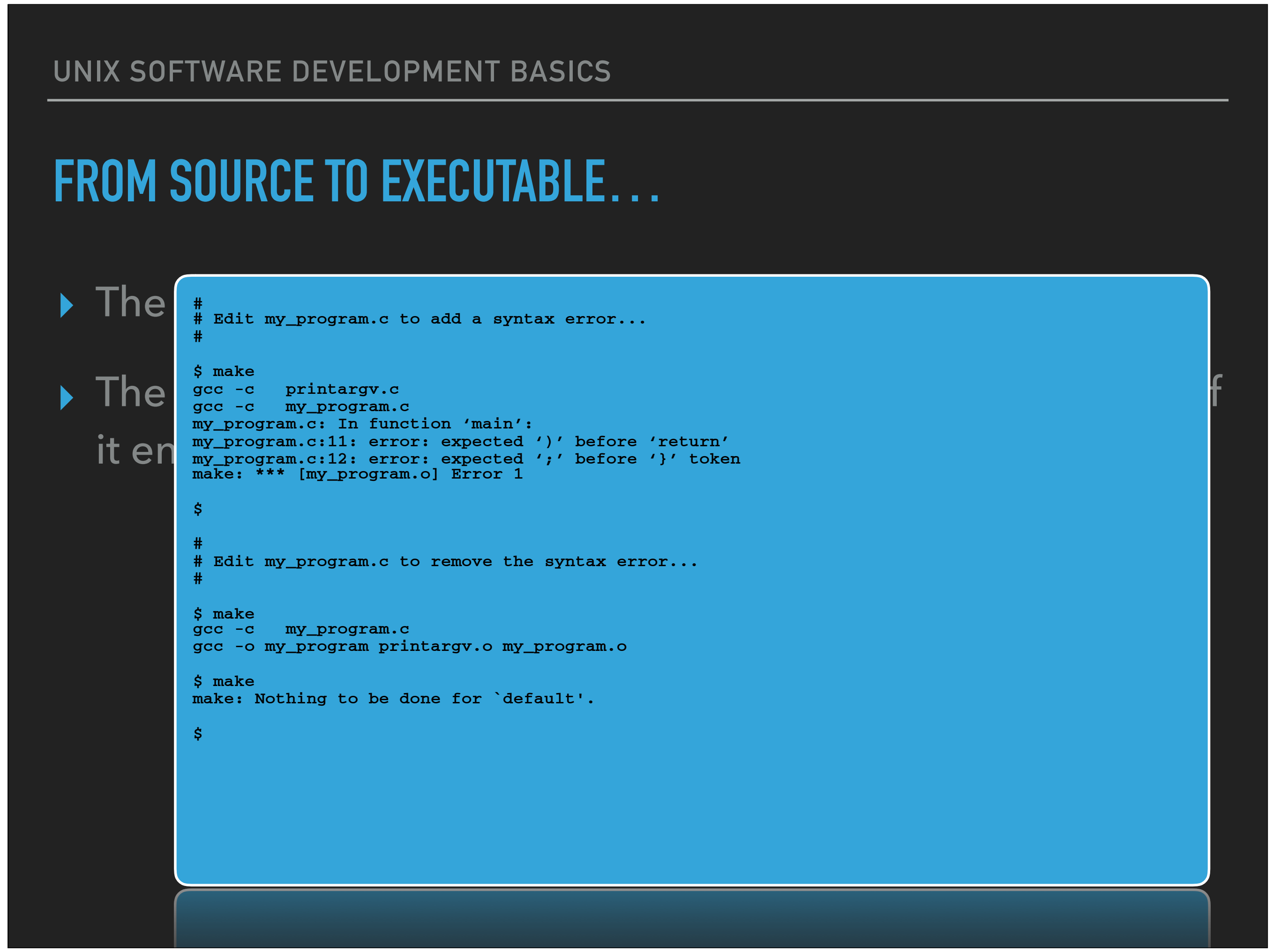
- make checks the return code from each command in a recipe; non-zero implies error, causes it to stop what it's doing

FROM SOURCE TO EXECUTABLE…

- The
- The … f
  it en

```
#
# Edit my_program.c to add a syntax error...
#

$ make
gcc -c   printargv.c
gcc -c   my_program.c
my_program.c: In function 'main':
my_program.c:11: error: expected ')' before 'return'
my_program.c:12: error: expected ';' before '}' token
make: *** [my_program.o] Error 1

$

#
# Edit my_program.c to remove the syntax error...
#

$ make
gcc -c   my_program.c
gcc -o my_program printargv.o my_program.o

$
```

- notice that after I fixed the syntax error, make did NOT recompile printargv
  - it had already done that — successfully — in the previous make and none of its dependencies were newer than the .o file
  -



- If I do make again, nothing has changed so nothing needs to be done

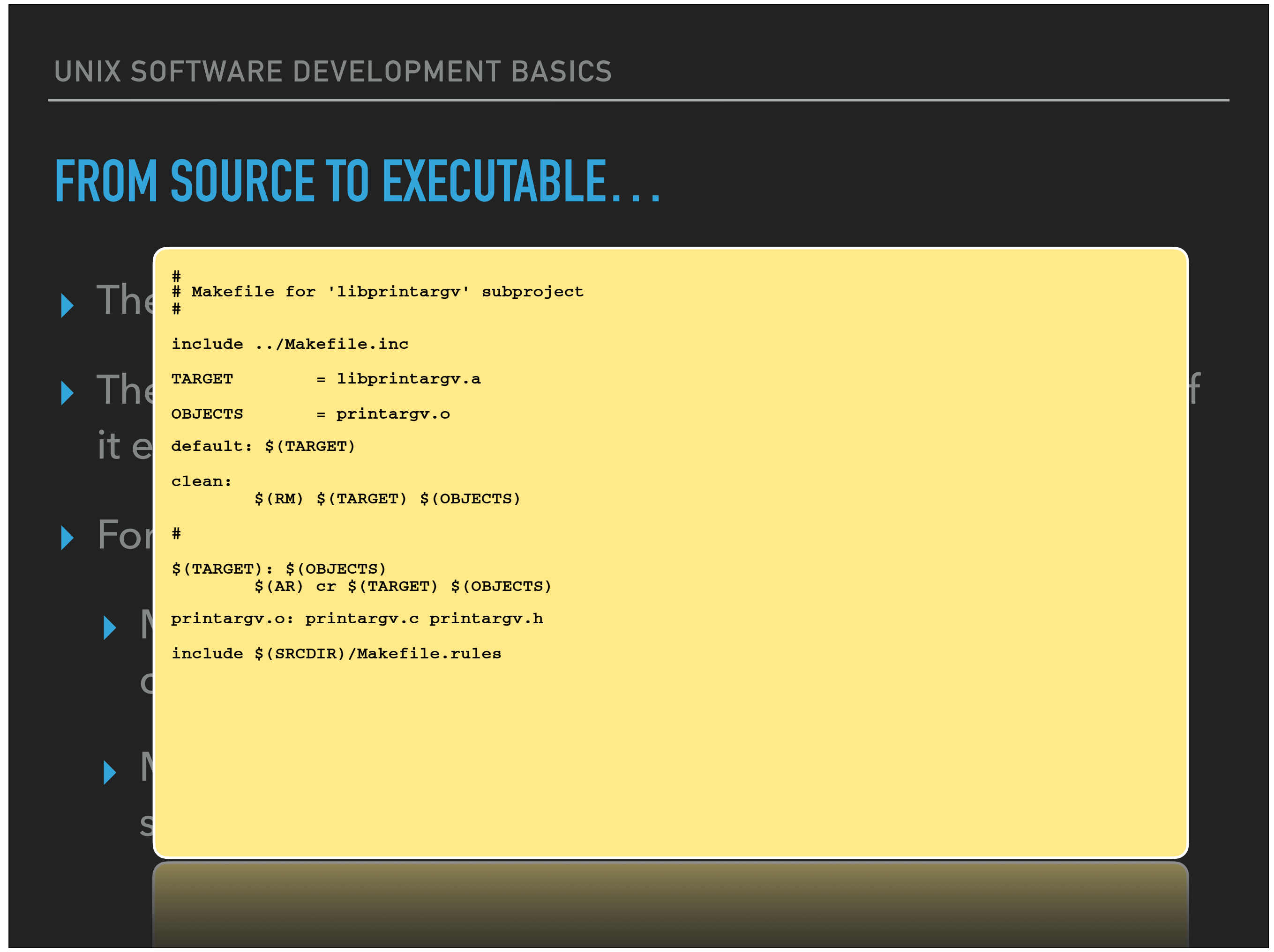
FROM SOURCE TO EXECUTABLE…

- The *make* utility uses a list of rules to transform data
- The *make* utility handles error-checking for you, stopping if it encounters any errors
- For complex projects, organization is again key
  - Makefile in each subdirectory to handle build of that component of the project
  - Makefile in the top directory that recurses into subdirectories

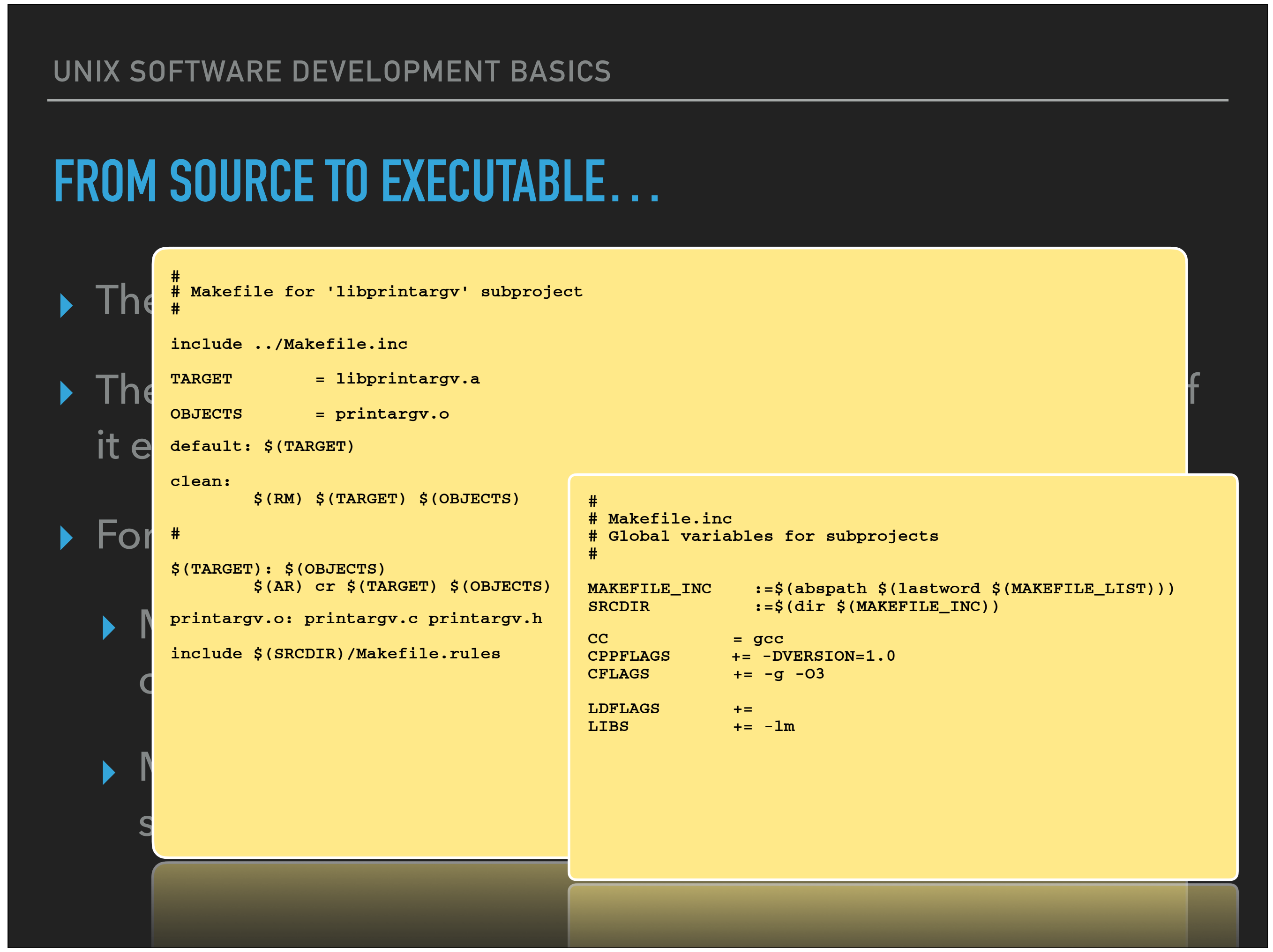
FROM SOURCE TO EXECUTABLE…

- The
- The … f
  it e
- For
  - M
    c
  - M
    s

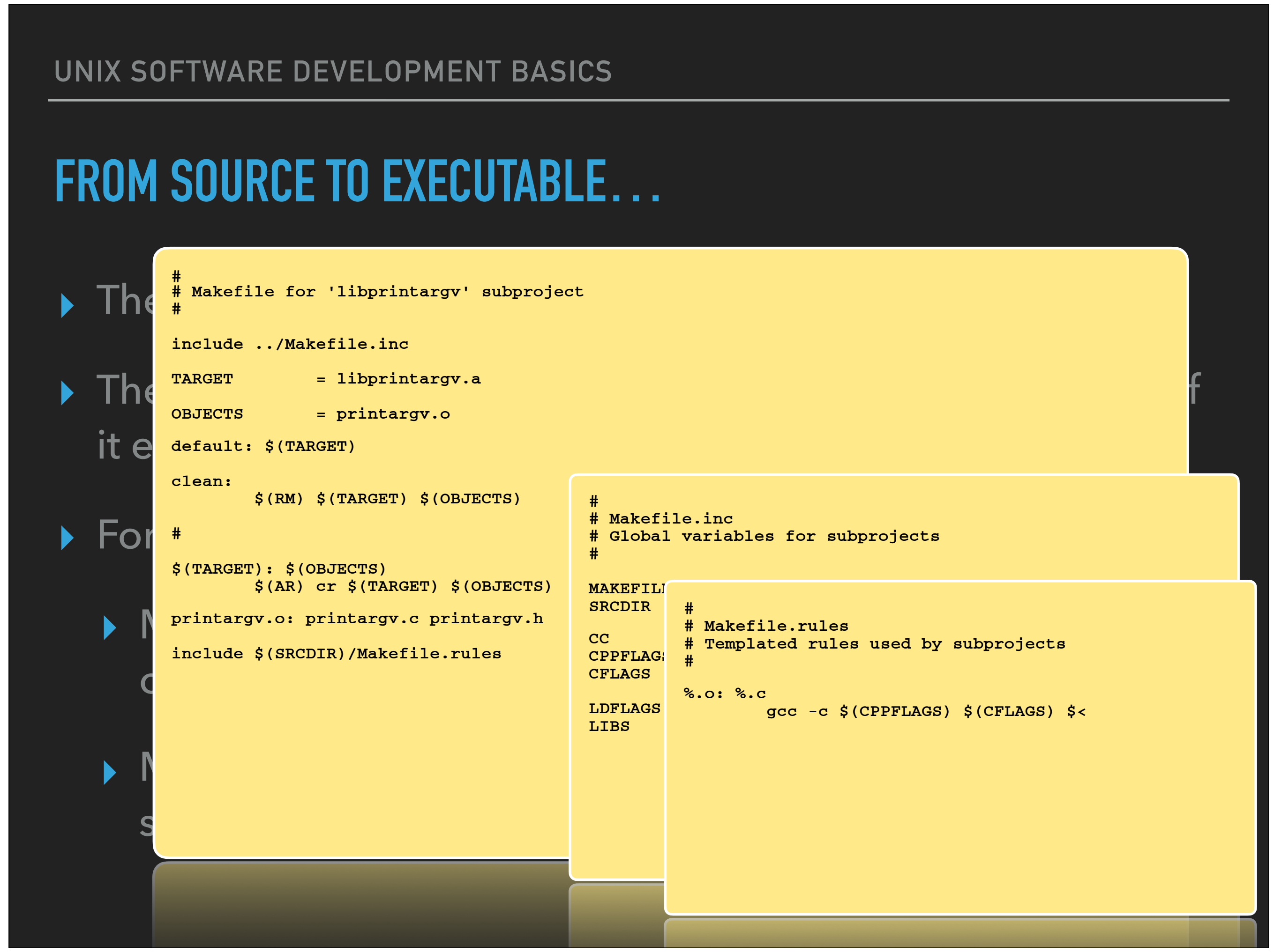
```
#
# Top-level Makefile for project
#

SUBPROJS        = libprintargv my_program

default:
        @for SUBPROJ in $(SUBPROJS); do make -C $$SUBPROJ; done

clean:
        @for SUBPROJ in $(SUBPROJS); do make -C $$SUBPROJ clean; done
```



- A makefile can import the contents of other files using the "include" statement
- E.g. single file containing all variable definitions, import at the start of each subproject's makefile
    - Compare against having to edit each subproject makefile to make changes
- E.g. file containing wildcard rules that are common to all subproject makefiles



- A makefile can import the contents of other files using the "include" statement
- E.g. single file containing all variable definitions, import at the start of each subproject's makefile
    - Compare against having to edit each subproject makefile to make changes
- E.g. file containing wildcard rules that are common to all subproject makefiles

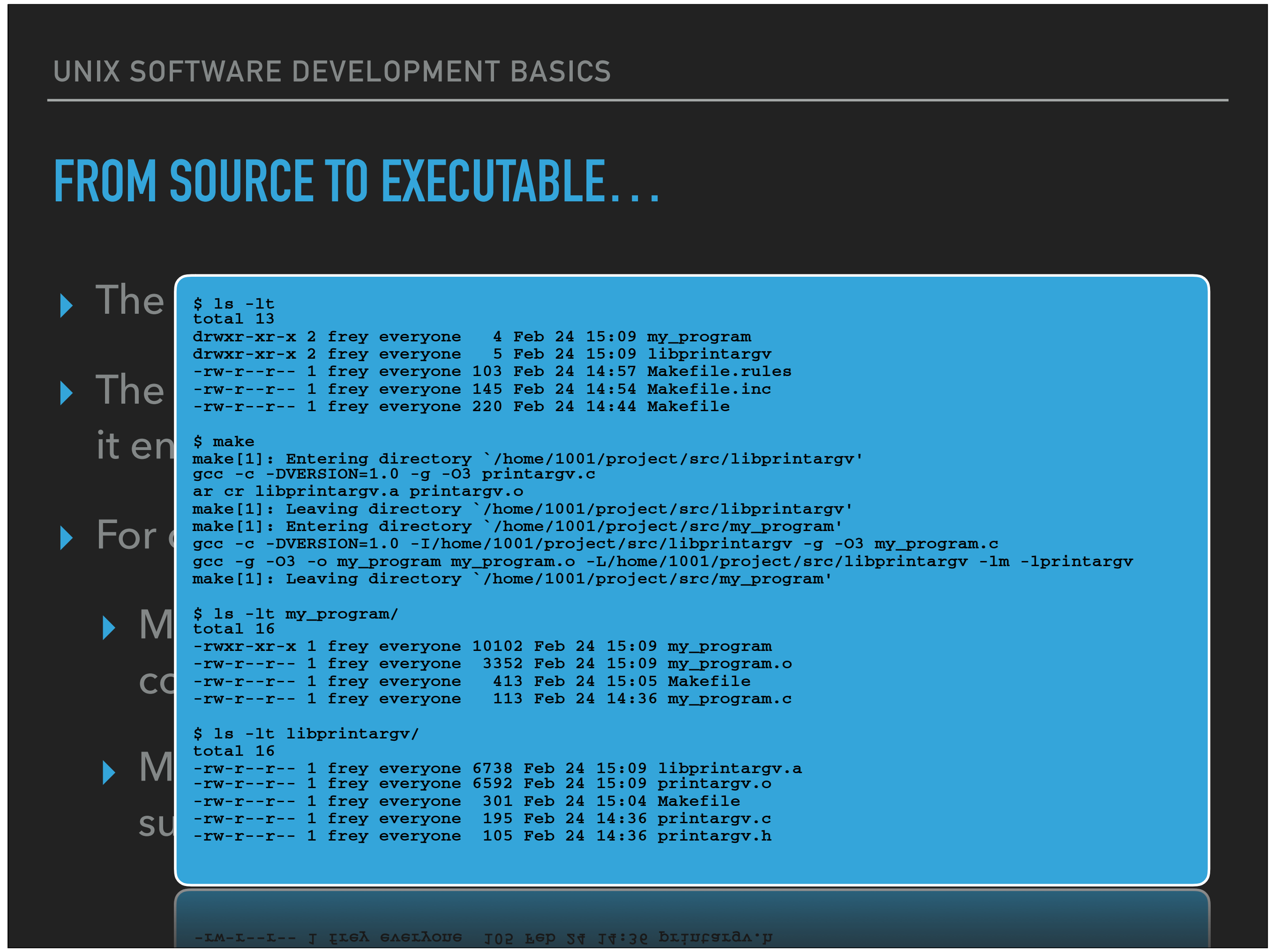


- A makefile can import the contents of other files using the "include" statement
- E.g. single file containing all variable definitions, import at the start of each subproject's makefile
    - Compare against having to edit each subproject makefile to make changes
- E.g. file containing wildcard rules that are common to all subproject makefiles

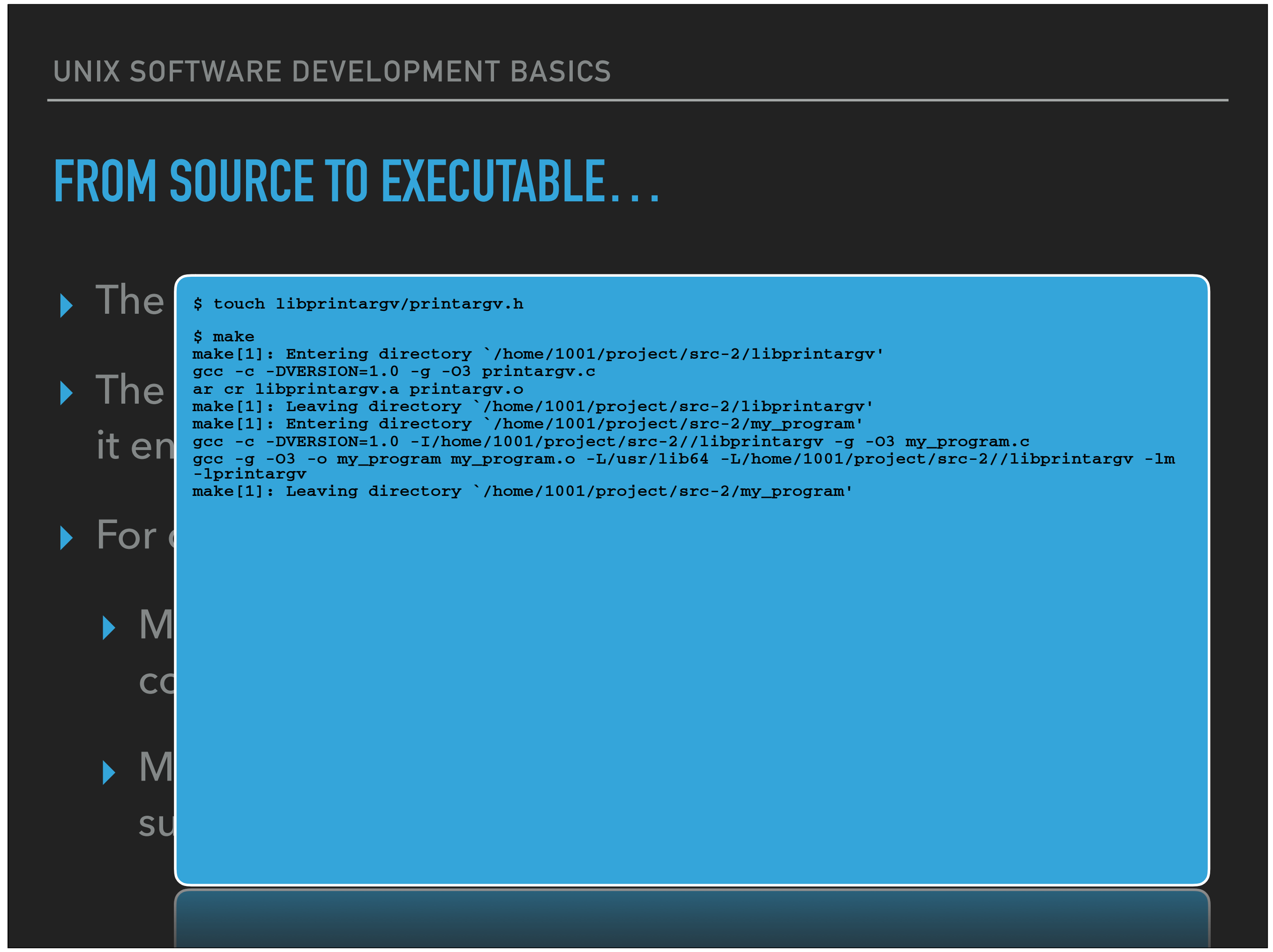
FROM SOURCE TO EXECUTABLE…

```
#
# Makefile for 'my_program' subproject
#

include ../Makefile.inc

TARGET        = my_program

OBJECTS       = my_program.o

# Augment values from ../Makefile.inc:
CPPFLAGS      += -I$(SRCDIR)/libprintargv
LDFLAGS       += -L$(SRCDIR)/libprintargv
LIBS          += -lprintargv

TARGET        = libprintargv.a

OBJECTS       = printargv.o

default: $(TARGET)

clean:
	$(RM) $(TARGET) $(OBJECTS)

#

$(TARGET): $(OBJECTS)
	$(CC) $(CFLAGS) -o $@ $+ $(LDFLAGS) $(LIBS)

my_program.o: my_program.c $(SRCDIR)/libprintargv

include $(SRCDIR)/Makefile.rules
```

- Similar to the makefile for the "libprintargv" subproject:
  - Add the path to libprintargv to CPPFLAGS for finding header files
  - Add the path to libprintargv to LDFLAGS for finding the library itself
  - Add the library reference to LIBS
- Produces my_program from the my_program.c source file and the static library generated by the libprintargv subproject
- Note the dependency on the libprintargv directory — any change to file(s) inside it implies my_program.c must be recompiled



- Similar to the makefile for the "libprintargv" subproject:
  - Add the path to libprintargv to CPPFLAGS for finding header files
  - Add the path to libprintargv to LDFLAGS for finding the library itself
  - Add the library reference to LIBS
- Produces my_program from the my_program.c source file and the static library generated by the libprintargv subproject



- The touch command updates the timestamp on the file(s) listed
- Since printargv.h is newer than printargv.o, printargv.c is recompiled
- printargv.o is newer than libprintargv.a, library is rebuilt
- Timestamp on libprintargv directory is now newer than my_program.o, so my_program.c is recompiled
- my_program.o is newer than my_program, executable is rebuilt

ADDITIONAL RESOURCES ON COMMUNITY CLUSTERS

- VALET's *vpkg_devrequire* command can set CPPFLAGS and LDFLAGS for you
  - If a version of a package contains header files, libraries, the appropriate "-I" and "-L" flags are added

ADDITIONAL RESOURCES ON COMMUNITY CLUSTERS

- VALET's *vpkg_devrequire* command can set CPPFLAGS and LDFLAGS for you
- A few programming project templates for C/C++/Fortran
  - /opt/templates/dev-projects
  - All integrate VALET dev environment setup within the *make* recipe