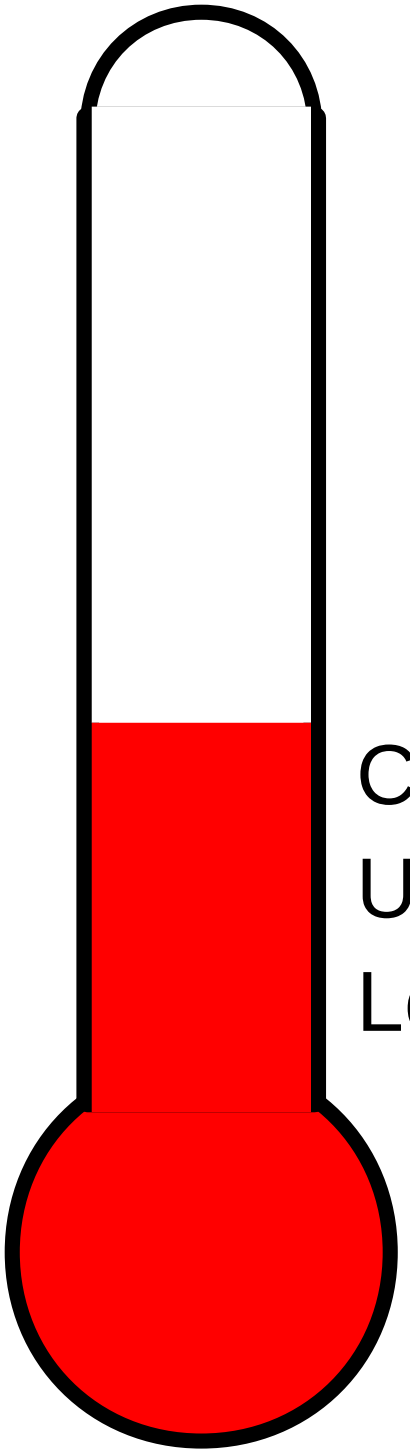
Building, installing and running software

Day two

Bob Dowling
University Computing Service

http://www-uxsup.csx.cam.ac.uk/courses/
http://training.csx.cam.ac.uk/

```
${HOME}/sw

PATH
MANPATH
PKG_CONFIG_PATH

tar   unzip

configure
make
make install
```



Configured builds
Unpacking
Location

UCS

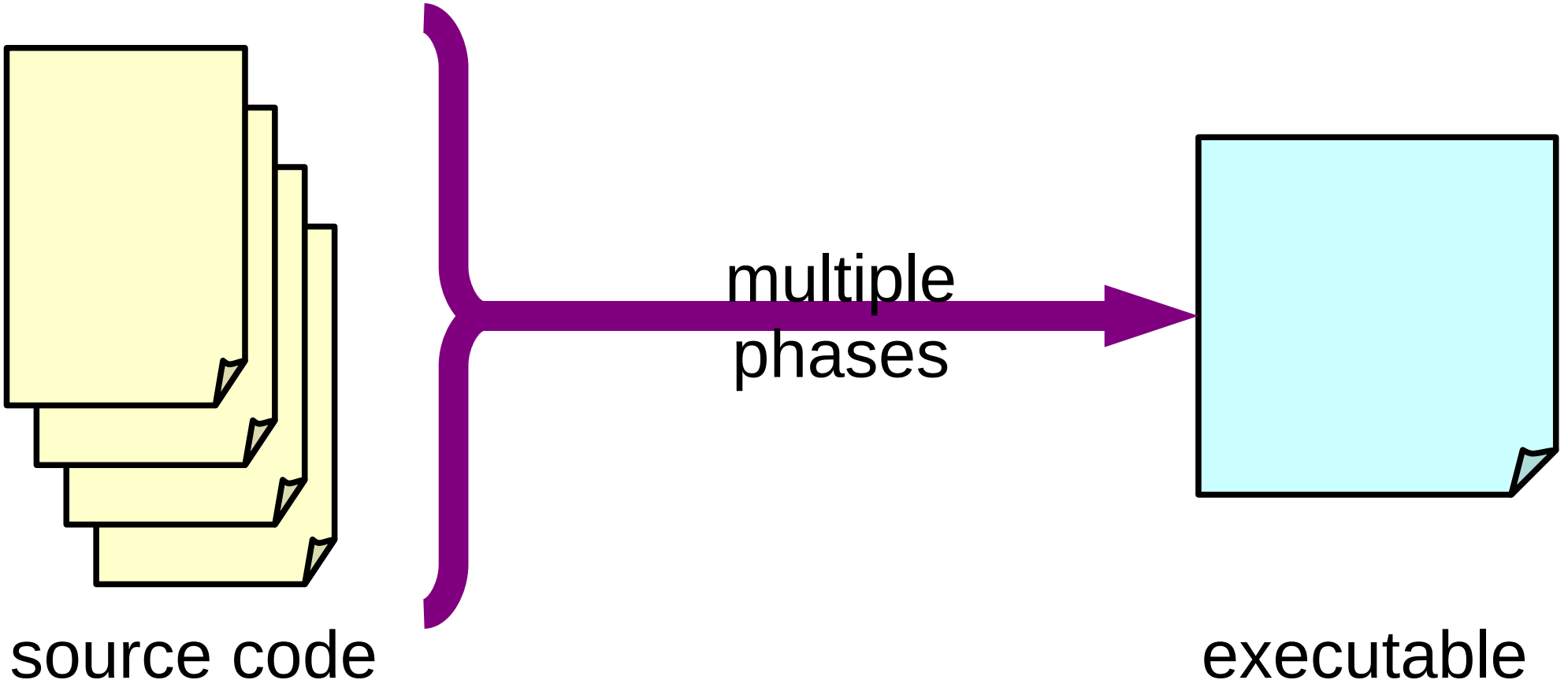
Course outline

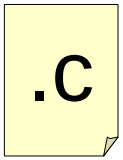
1. Location — 1st afternoon
2. Unpacking
3. “Configured” builds

4. make and Makefiles — 2nd afternoon
5. Software libraries

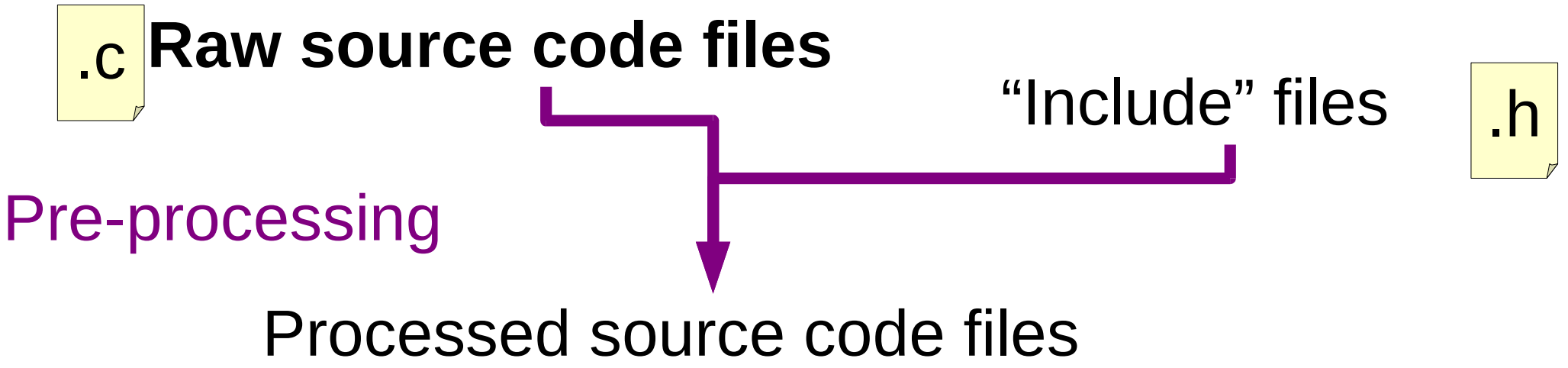
6. “Real world” example — 3rd afternoon
7. Recursive make

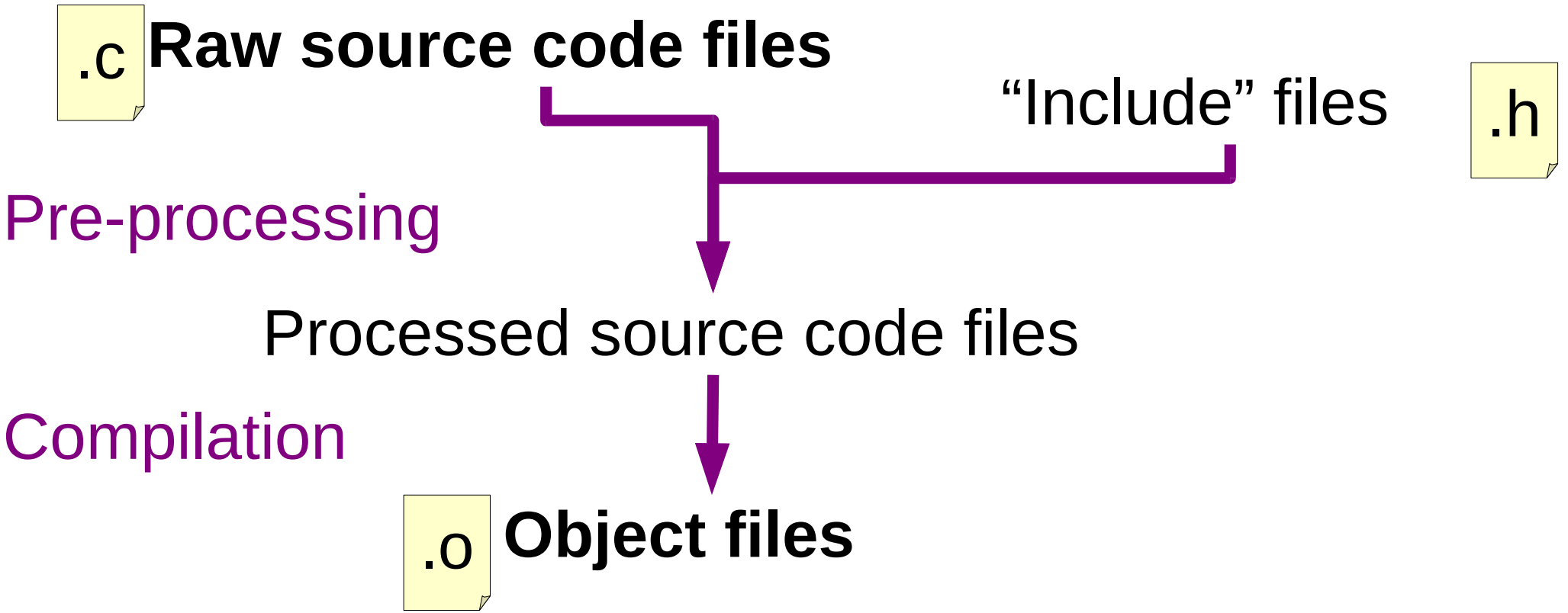
The build process

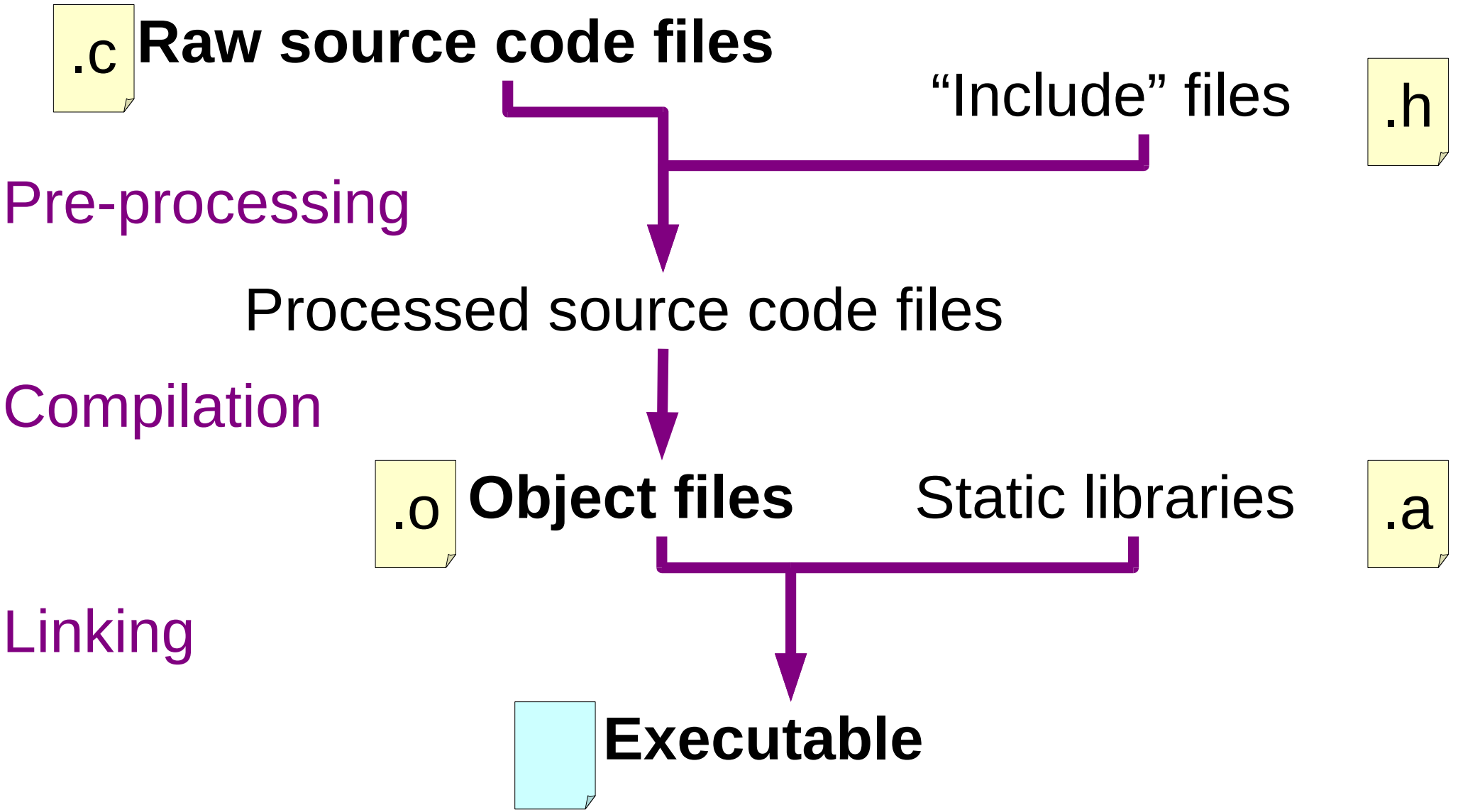


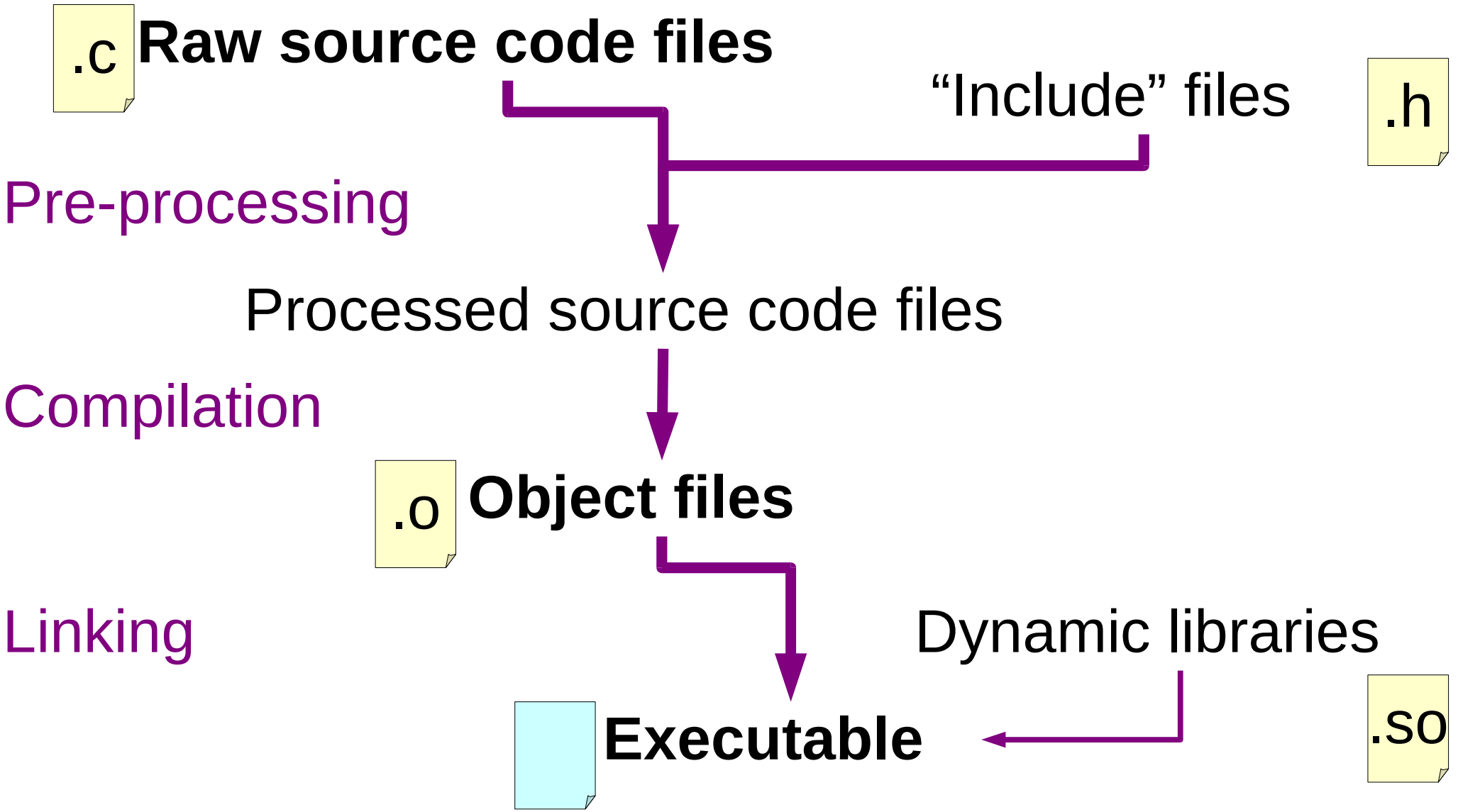


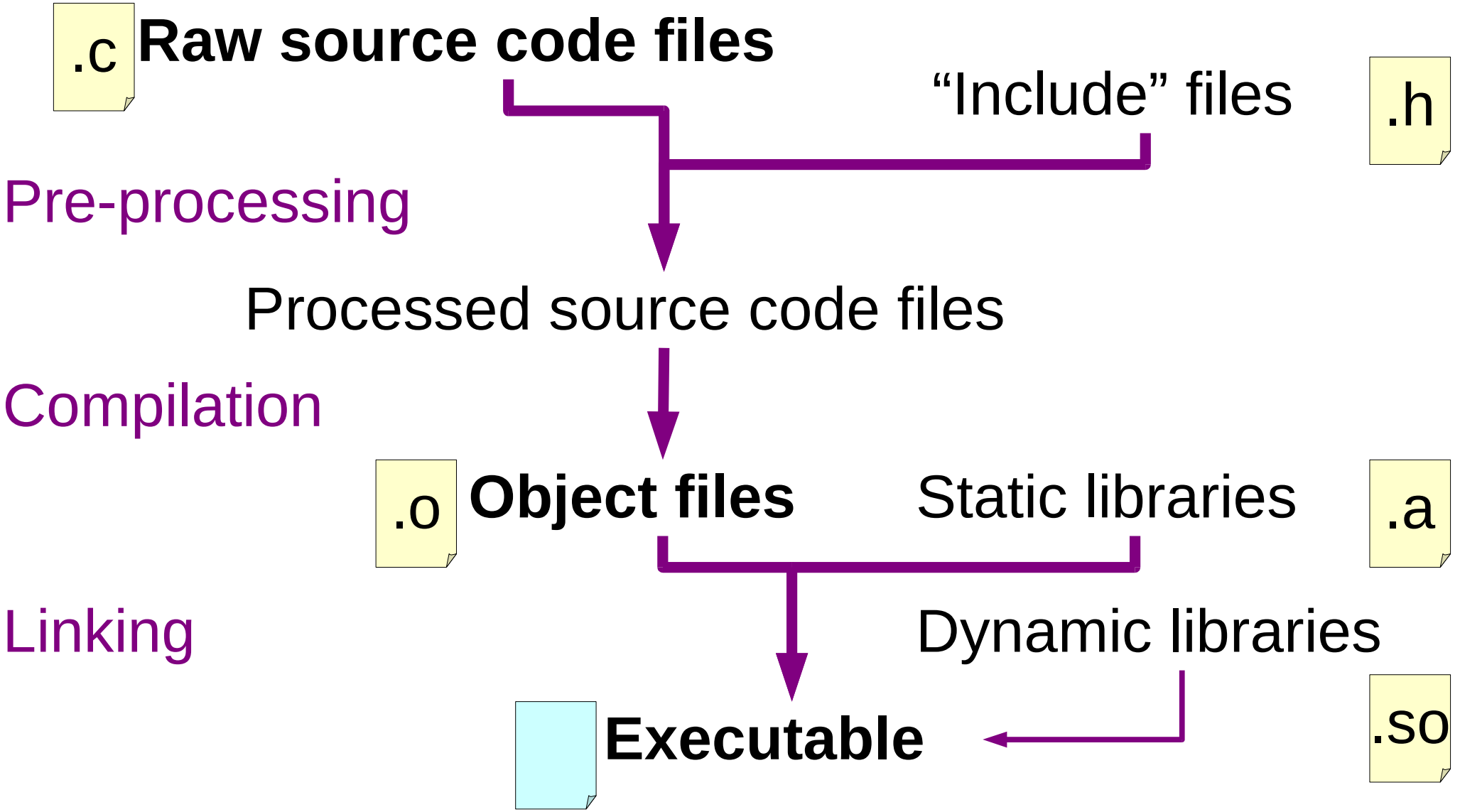
Raw source code files

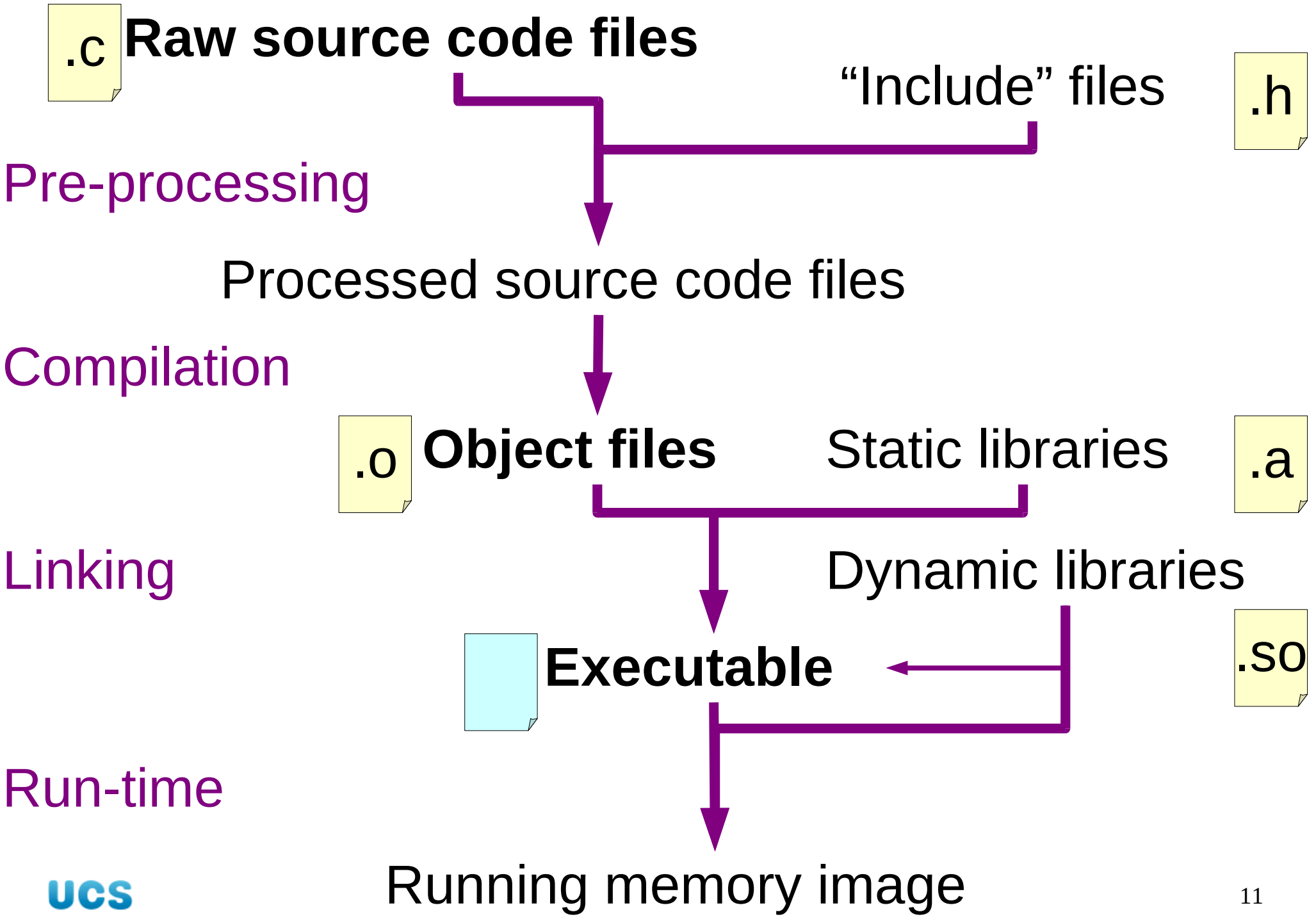

.c
Raw source code files
“Include” files
.h
Pre-processing
Processed source code files


.c
Raw source code files
“Include” files
.h
Pre-processing
Processed source code files
Compilation
.o
Object files


.c
Raw source code files
“Include” files
.h
Pre-processing
Processed source code files
Compilation
.o
Object files
Static libraries
.a
Linking
Executable

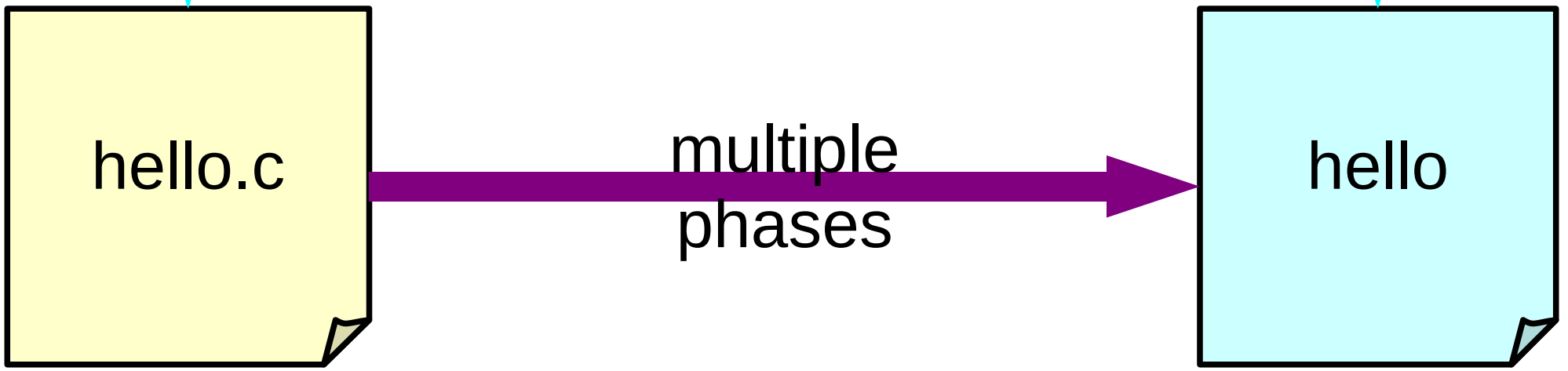

.c
Raw source code files
“Include” files
.h
Pre-processing
Processed source code files
Compilation
.o
Object files
Linking
Dynamic libraries
.so
Executable


.c
Raw source code files
"Include" files
.h
Pre-processing
Processed source code files
Compilation
.o
Object files
Static libraries
.a
Linking
Dynamic libraries
Executable
.so


.c
Raw source code files
“Include” files
.h
Pre-processing
Processed source code files
Compilation
.o
Object files
Static libraries
.a
Linking
Dynamic libraries
Executable
.so
Run-time
Running memory image

The “compiler”

```
$ cc hello.c -o hello
```



single source code file

1. pre-process
2. compile
3. link

executable

Always automate!

```
make
```

Worked example

1. `/ux/Lessons/Building` → `/tmp/building`
   `hello.c`

2. `$ make hello`

```
cc   hello.c  -o hello
```

make knows how to build “hello”

3. `$ make hello`

```
make: `hello' is up to date.
```

make won't rebuild files unnecessarily

Worked example

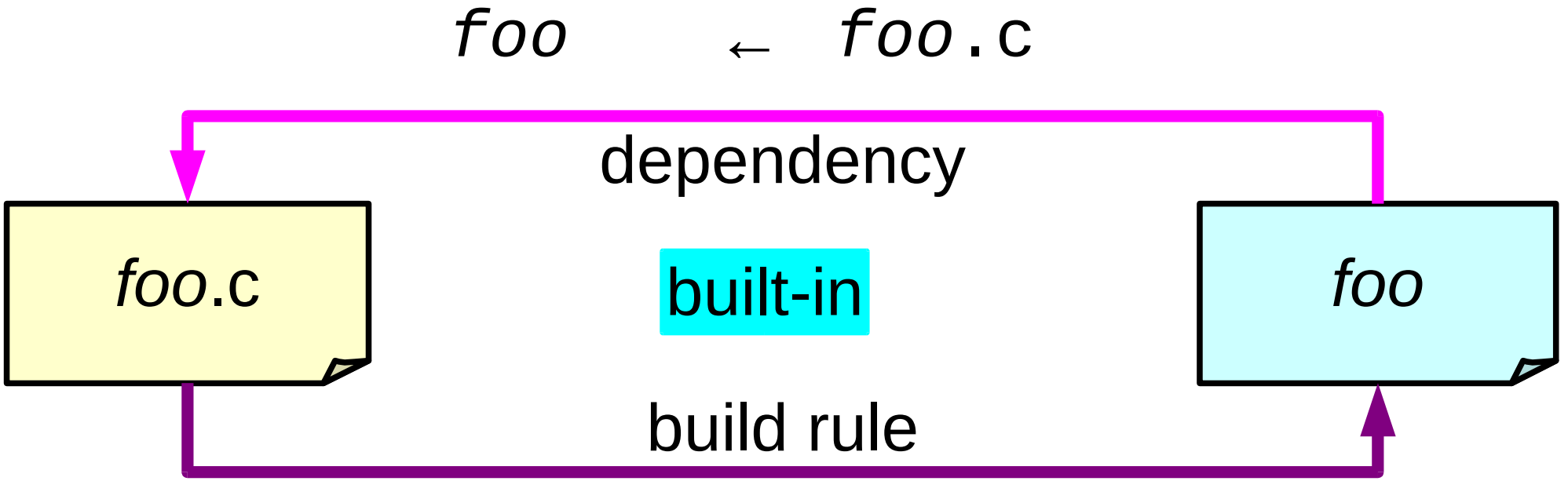
4. Update `hello.c`

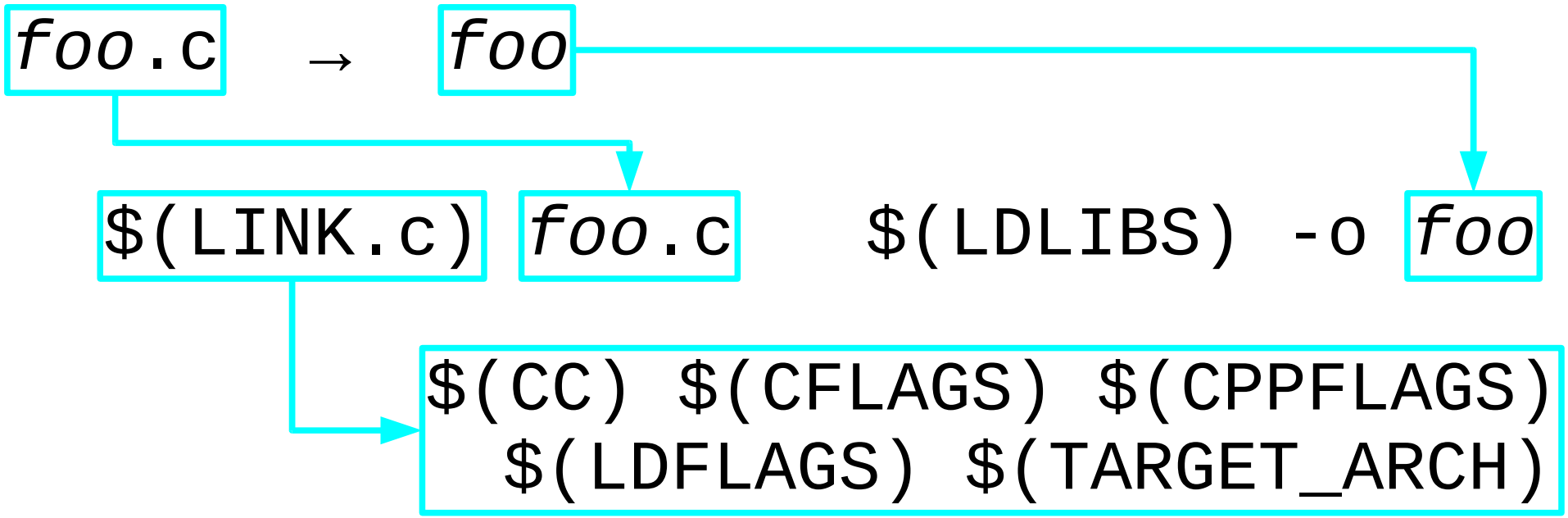
   `"Hello, world!"` → `"Goodbye, world!"`

5. `$ make hello`

```
cc   hello.c  -o hello
```

make rebuilds the executable this time


foo ← foo.c
dependency
foo.c
built-in
foo
build rule


foo.c → foo
$(LINK.c) foo.c $(LDLIBS) -o foo
$(CC) $(CFLAGS) $(CPPFLAGS)
$(LDFLAGS) $(TARGET_ARCH)

Common macros **cc**

C compiler

Default value `cc`

Possible other values:

GNU C compiler `gcc`

POSIX C compiler `c99`

Common macros **`CFLAGS`**

C compiler options

Default value *none*

Possible other values:

Optimisation level `-O3`

Debugging level `-g3`

Common macros **CPPFLAGS**

C pre-processor options

Default value *none*

Possible other values:

Extra include directories `-I/opt/include`

Derived from pkg-config:

```
$ pkg-config --cflags alsa
-I/usr/include/alsa
```

Common macros **LDFLAGS**

Linking options

Default value *none*

Possible other values:

Require static linking `-static`

Common macros **LDLIBS**

Extra libraries to use

Default value *none*

Possible other values:

Use system **m**aths library `-lm`

Derived from pkg-config:

```
$ pkg-config --libs alsa
-lasound
```

Common macros **TARGET_ARCH**

Select specific architecture

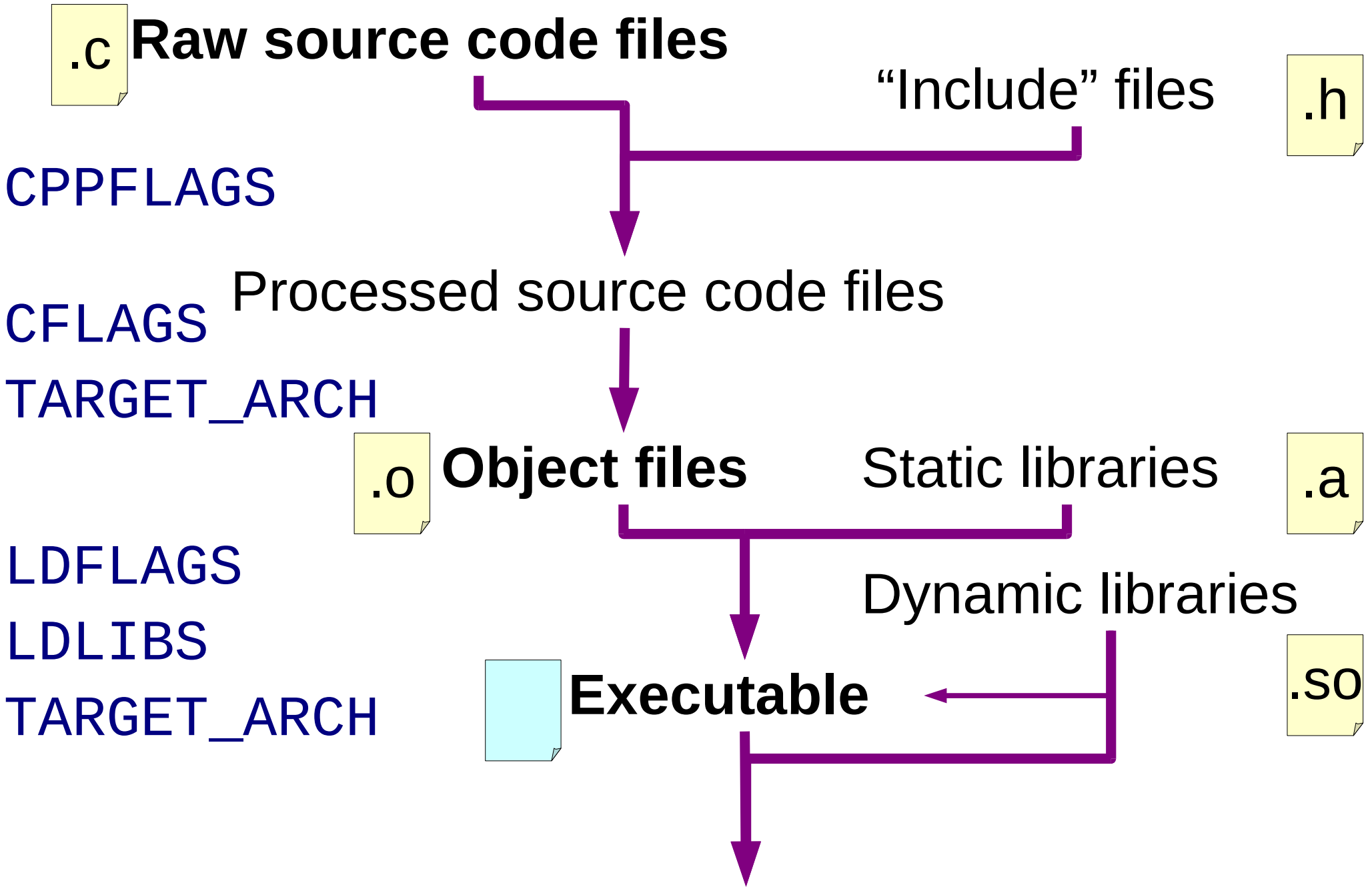
Default value *none*

Possible other values:

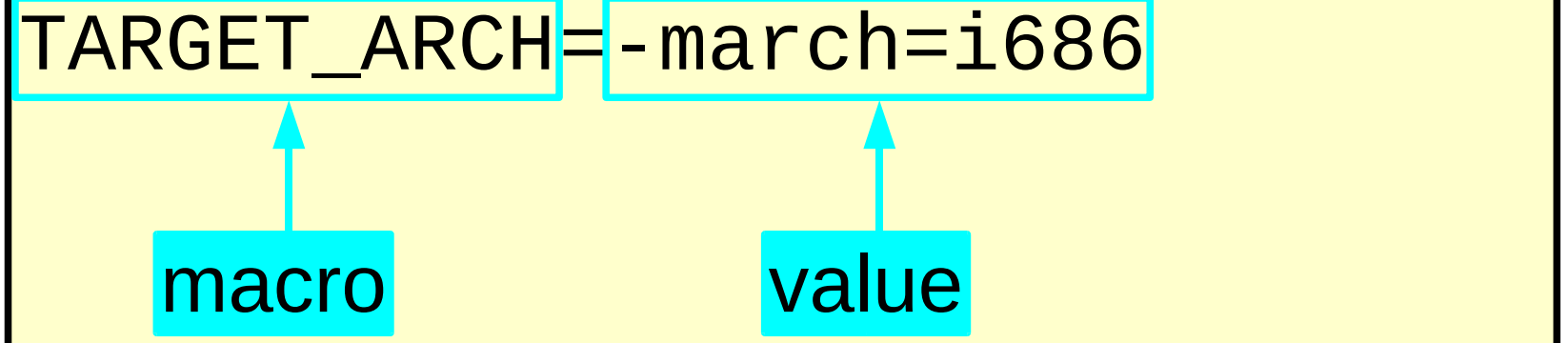
Build for Pentium II `-march=i686`

Derived from arch command:

```
$ arch
i686
```


.c
Raw source code files
“Include” files
.h
CPPFLAGS
Processed source code files
CFLAGS
TARGET_ARCH
.o
Object files
Static libraries
.a
LDFLAGS
LDLIBS
TARGET_ARCH
Dynamic libraries
Executable
.so
Running memory image

```
# Makefile – the configuration
# file for the "make" program.

# Macros are set like this:

CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686
```

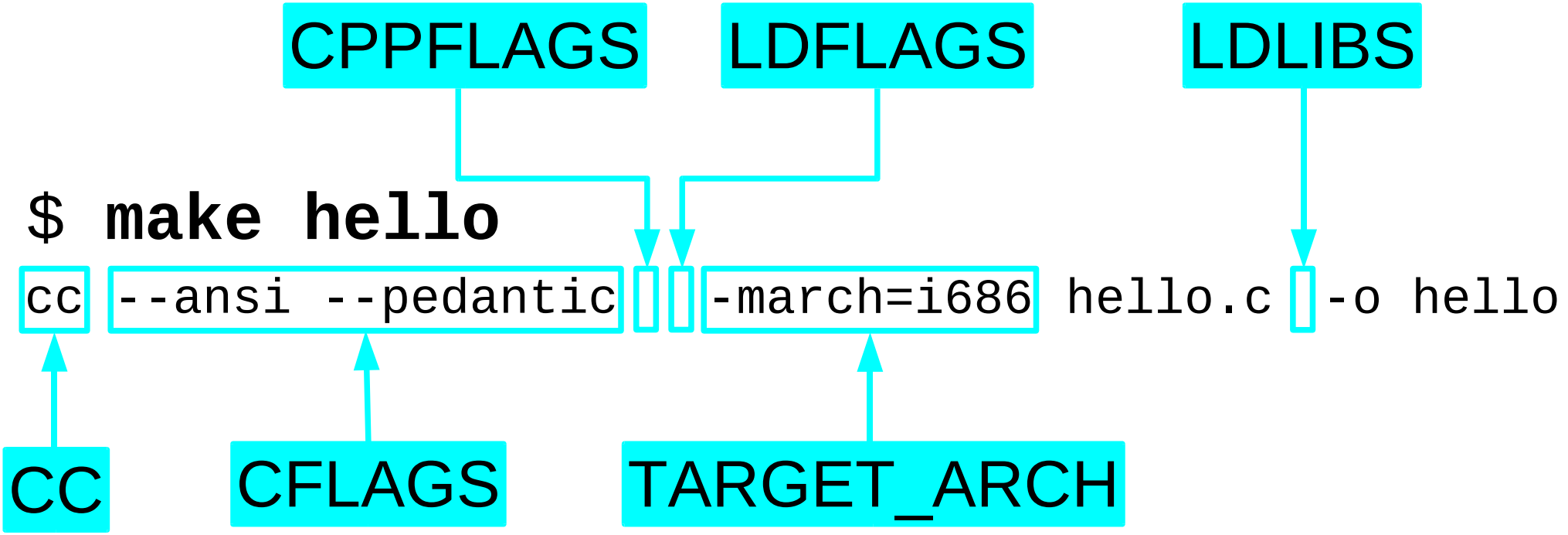


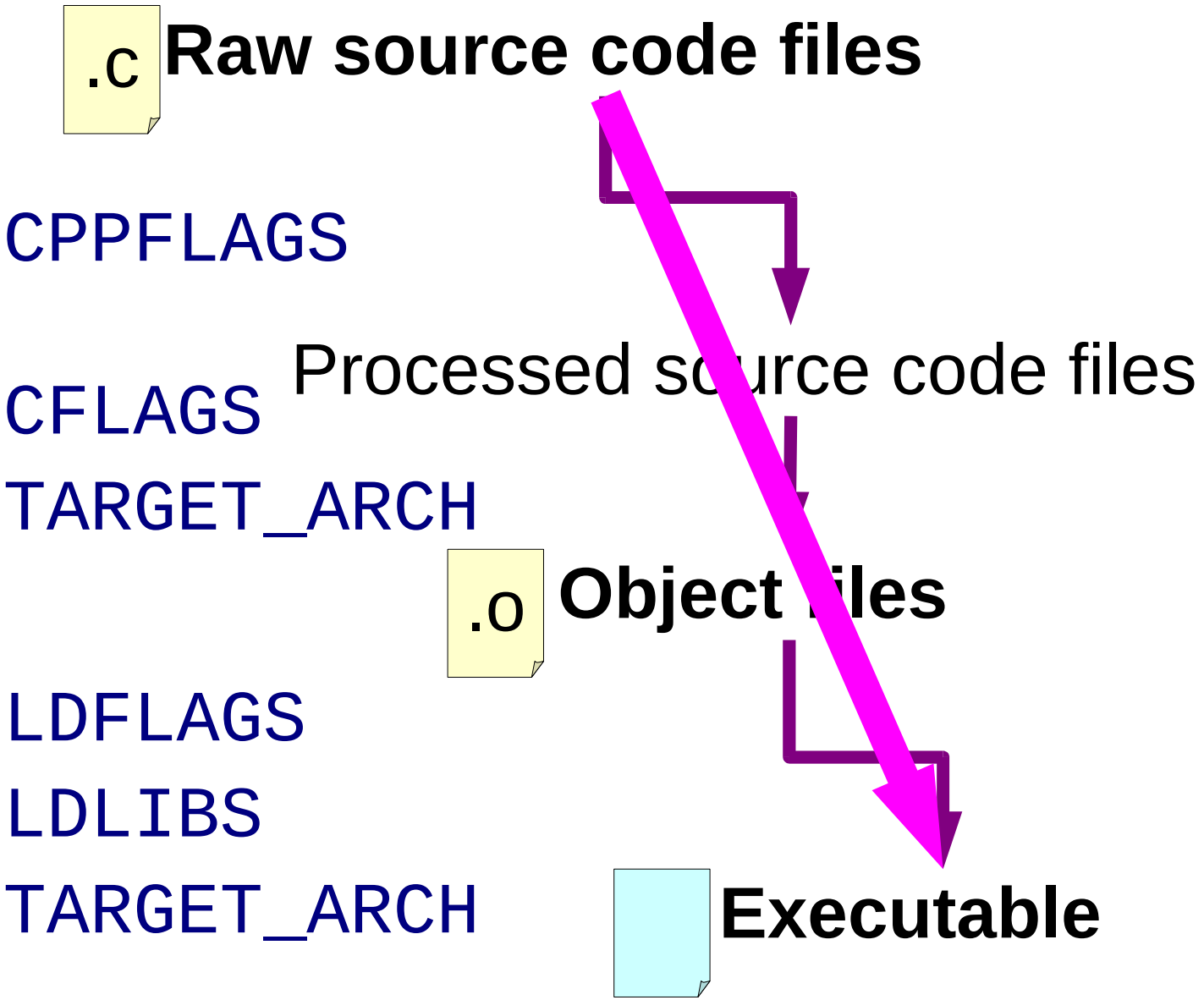
Makefile

Worked example

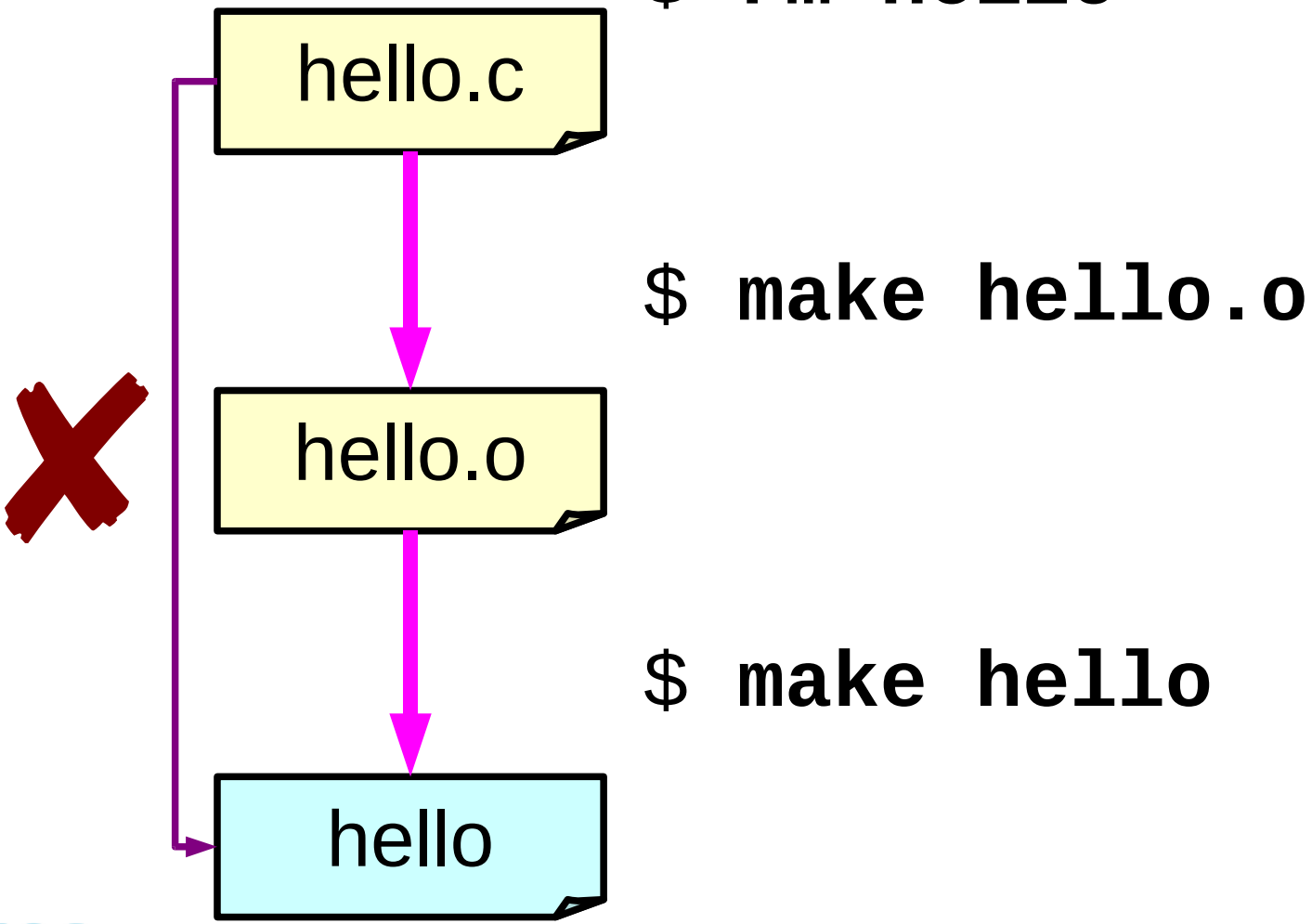
1. `$ cd /tmp/building`
2.
```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686

                    Makefile
```
3. `$ rm hello`
4. `$ make hello`

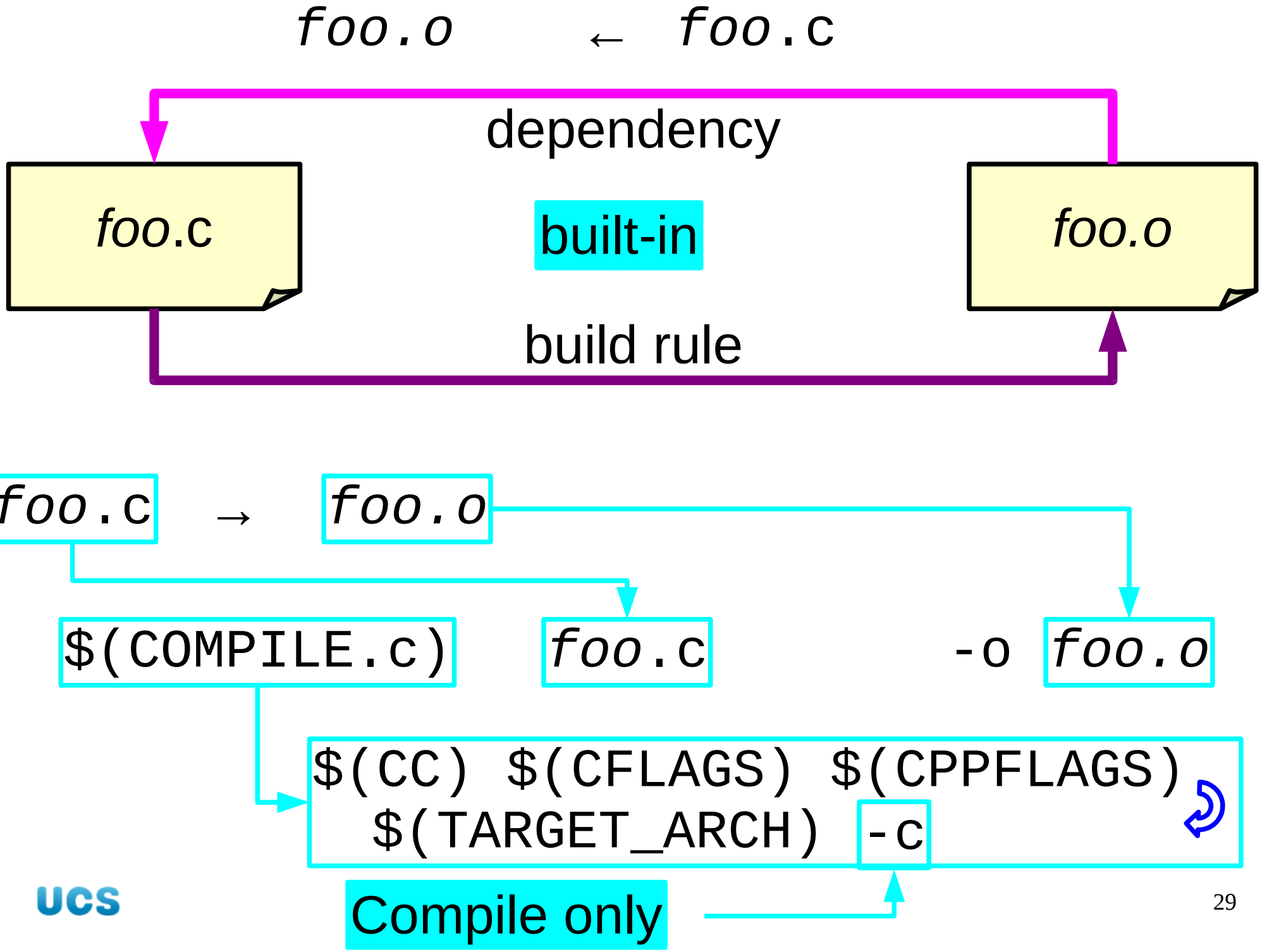
```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686

                    Makefile
```

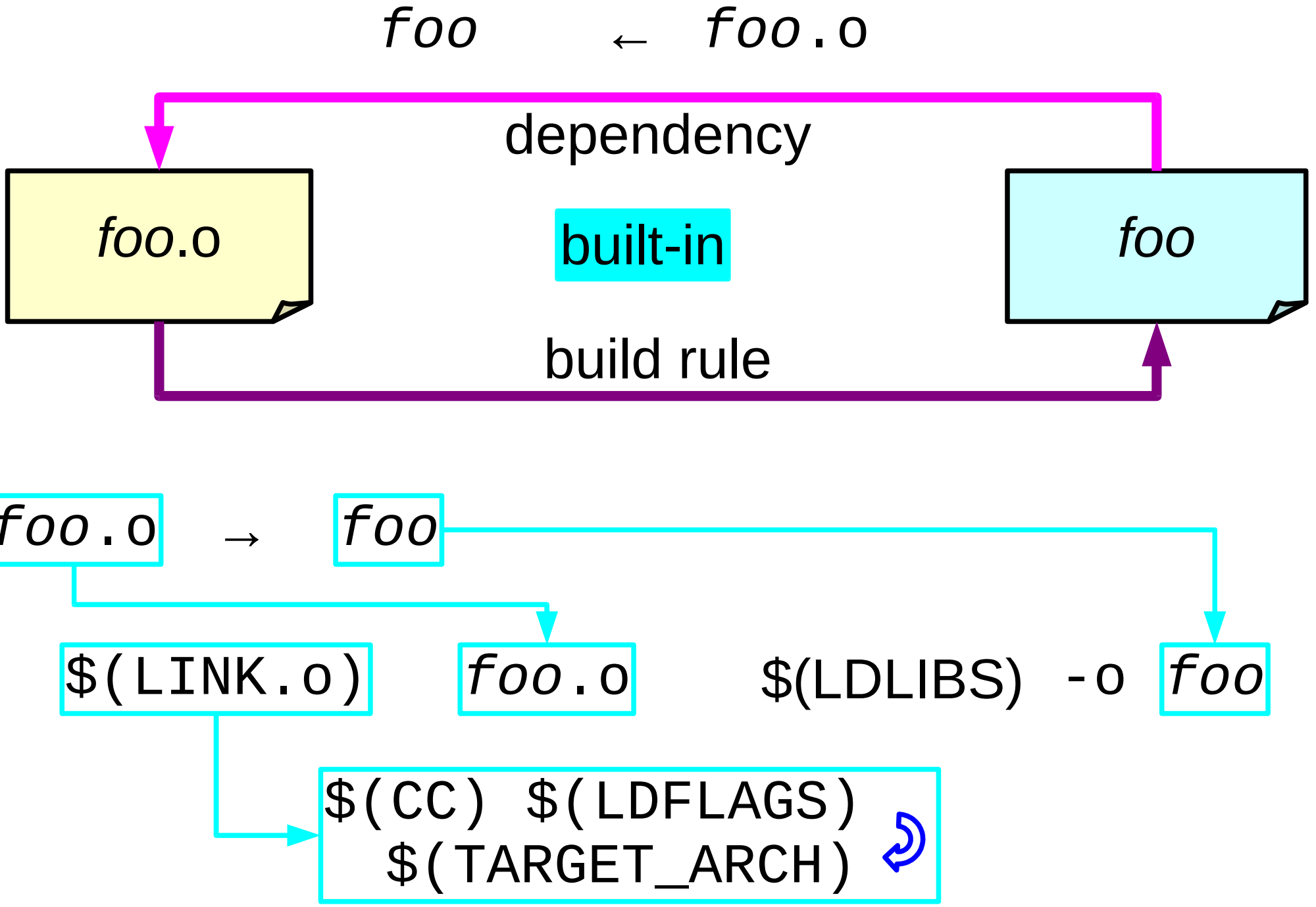



.c
Raw source code files
CPPFLAGS
Processed source code files
CFLAGS
TARGET_ARCH
.o
Object files
LDFLAGS
LDLIBS
TARGET_ARCH
Executable

One step at a time




foo.o ← foo.c
dependency
foo.c
built-in
foo.o
build rule
foo.c → foo.o
$(COMPILE.c)
foo.c
-o foo.o
$(CC) $(CFLAGS) $(CPPFLAGS)
$(TARGET_ARCH) -c
Compile only


foo ← foo.o
dependency
foo.o
built-in
foo
build rule
foo.o → foo
$(LINK.o)
foo.o
$(LDLIBS) -o foo
$(CC) $(LDFLAGS)
$(TARGET_ARCH)

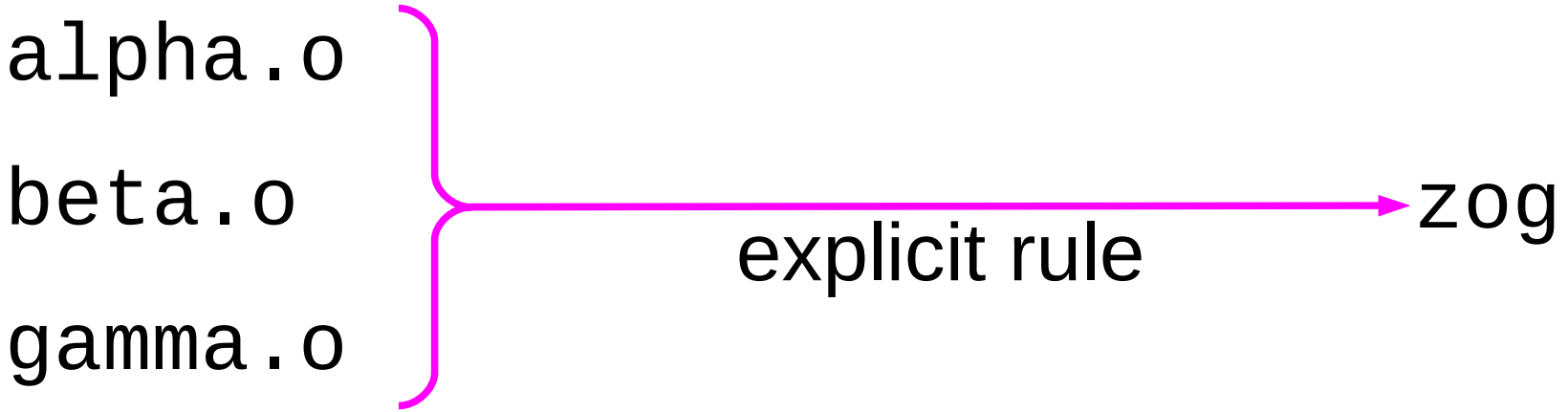
make coffee

Dependencies:
1. 45p
2. coffee machine

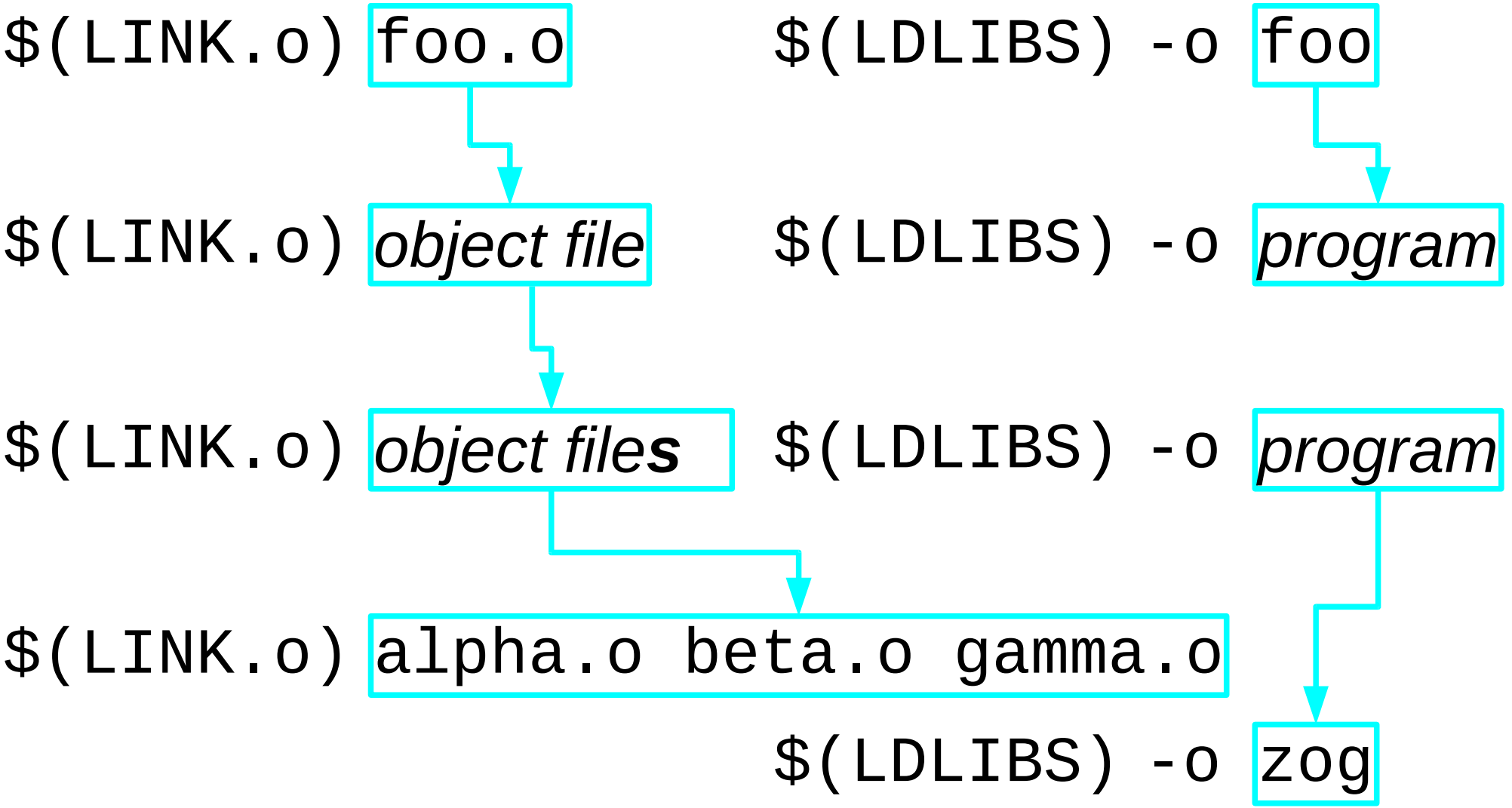
Ten minutes break

UCS

foo.o
built-in rule
foo

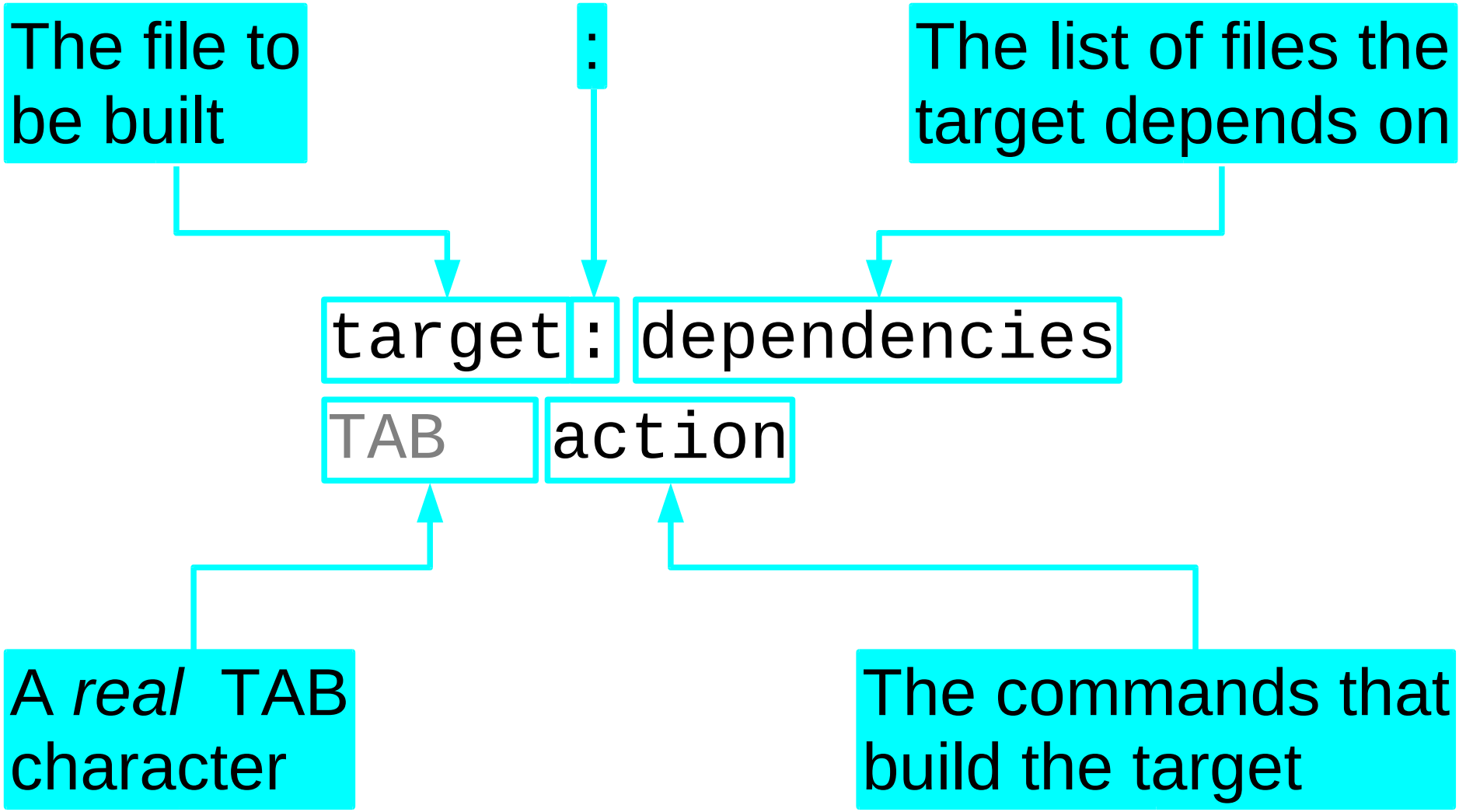

alpha.o
beta.o
gamma.o
explicit rule
zog

built-in action →

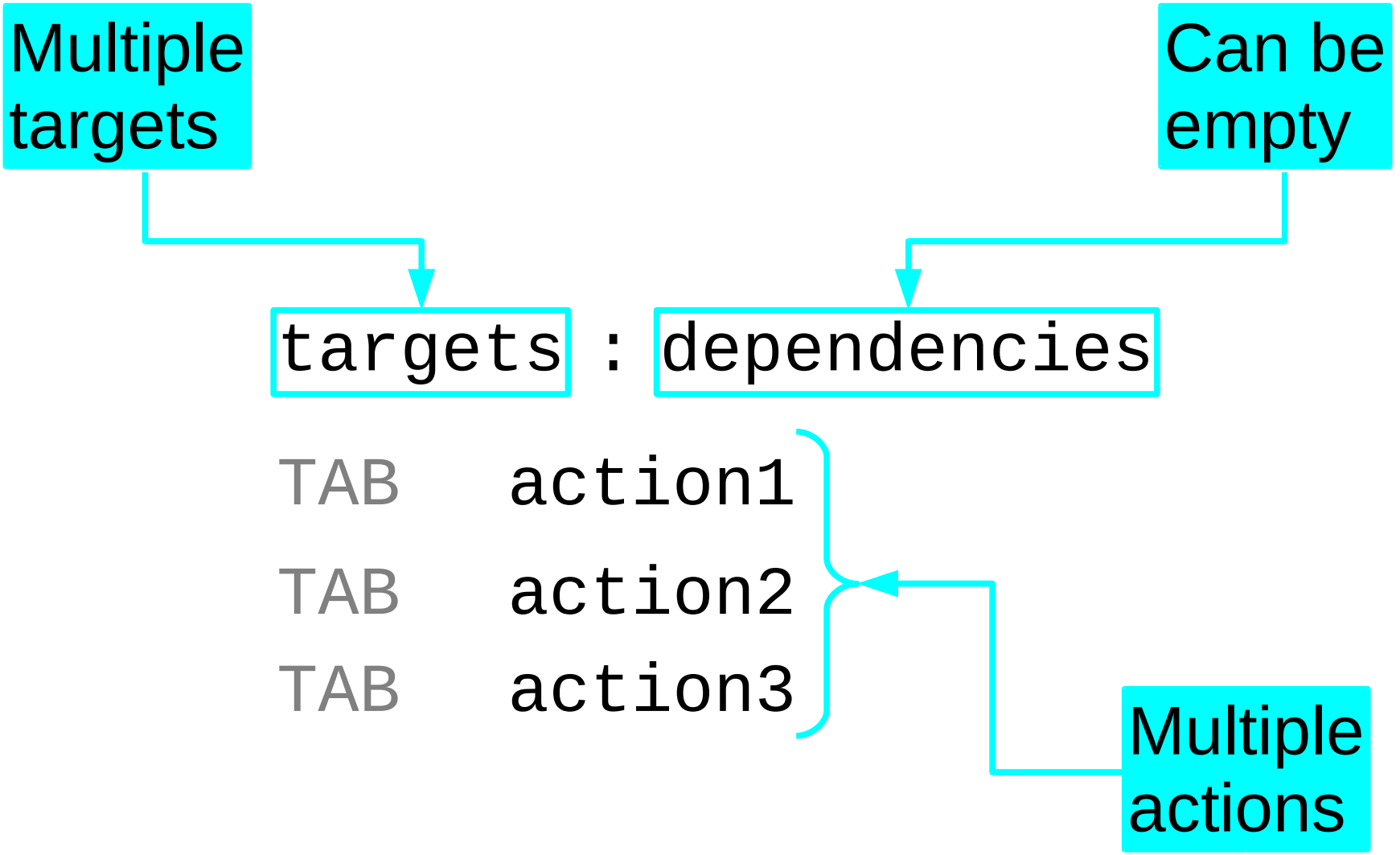


→ added action

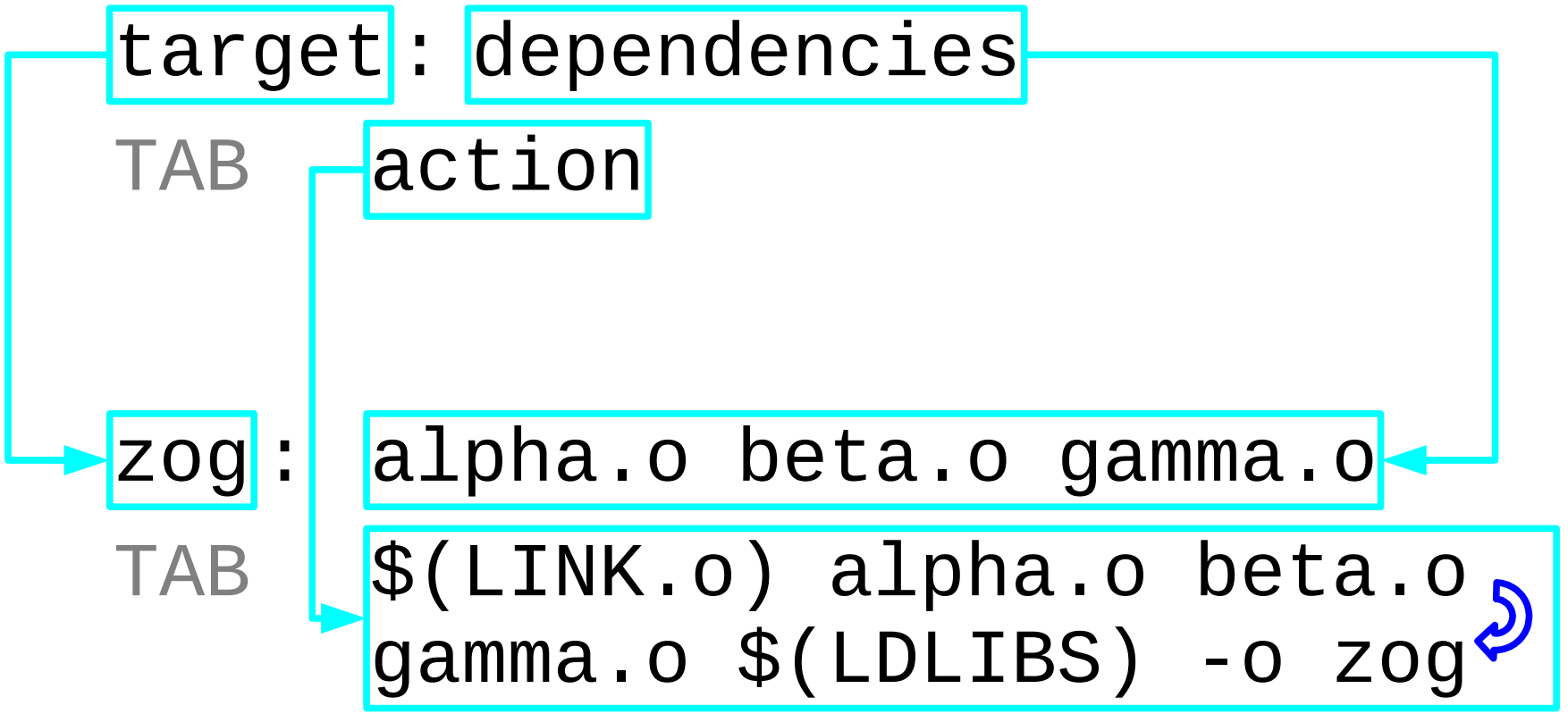
Makefile rules



General Makefile rules



Our rule


target : dependencies
TAB action
zog : alpha.o beta.o gamma.o
TAB $(LINK.o) alpha.o beta.o
gamma.o $(LDLIBS) -o zog

The Makefile

```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686

zog: alpha.o beta.o gamma.o
	$(LINK.o) alpha.o beta.o \
	gamma.o $(LDLIBS) -o zog
```

TAB

TAB

Line wrap

Worked example

1. `$ cd /ux/Lessons/Building`

2. `$ cp  alpha.c beta.c gamma.c  /tmp/building`

3. `$ cd /tmp/building`

Worked example

4. Update Makefile

```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686

zog: alpha.o beta.o gamma.o
TAB $(LINK.o) alpha.o beta.o \
    gamma.o $(LDLIBS) -o zog
```

Makefile

5. `$ make zog`

Improving our rule

```
zog: alpha.o beta.o gamma.o
TAB   $(LINK.o) alpha.o beta.o gamma.o \
      $(LDLIBS) -o zog
```

“Automatic macros”

```
zog: alpha.o beta.o gamma.o
TAB  $(LINK.o) alpha.o beta.o gamma.o \
     $(LDLIBS) -o zog

zog: alpha.o beta.o gamma.o
TAB  $(LINK.o)                $^          \
     $(LDLIBS) -o $@
```

dependencies

target

```
zog: alpha.o beta.o gamma.o
TAB   $(LINK.o) $^  $(LDLIBS) -o  $@
```

Values defined rule-by-rule

```
zog: alpha.o beta.o gamma.o
TAB   $(LINK.o)  $^  $(LDLIBS) -o  $@
```

alpha.o beta.o ...

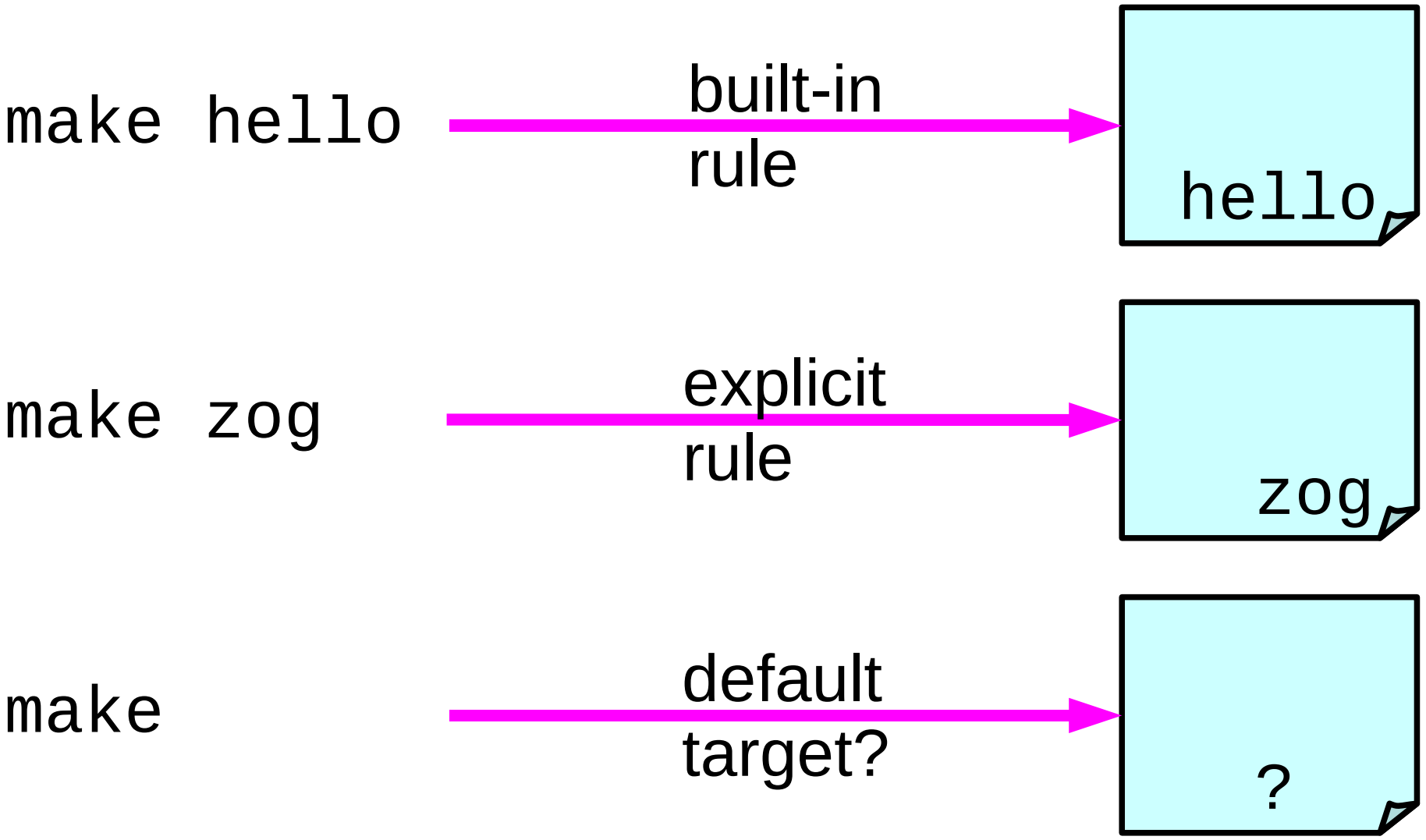
zog

```
flub: one.o two.o three.o four.o
TAB   $(LINK.o)  $^  $(LDLIBS) -o  $@
```

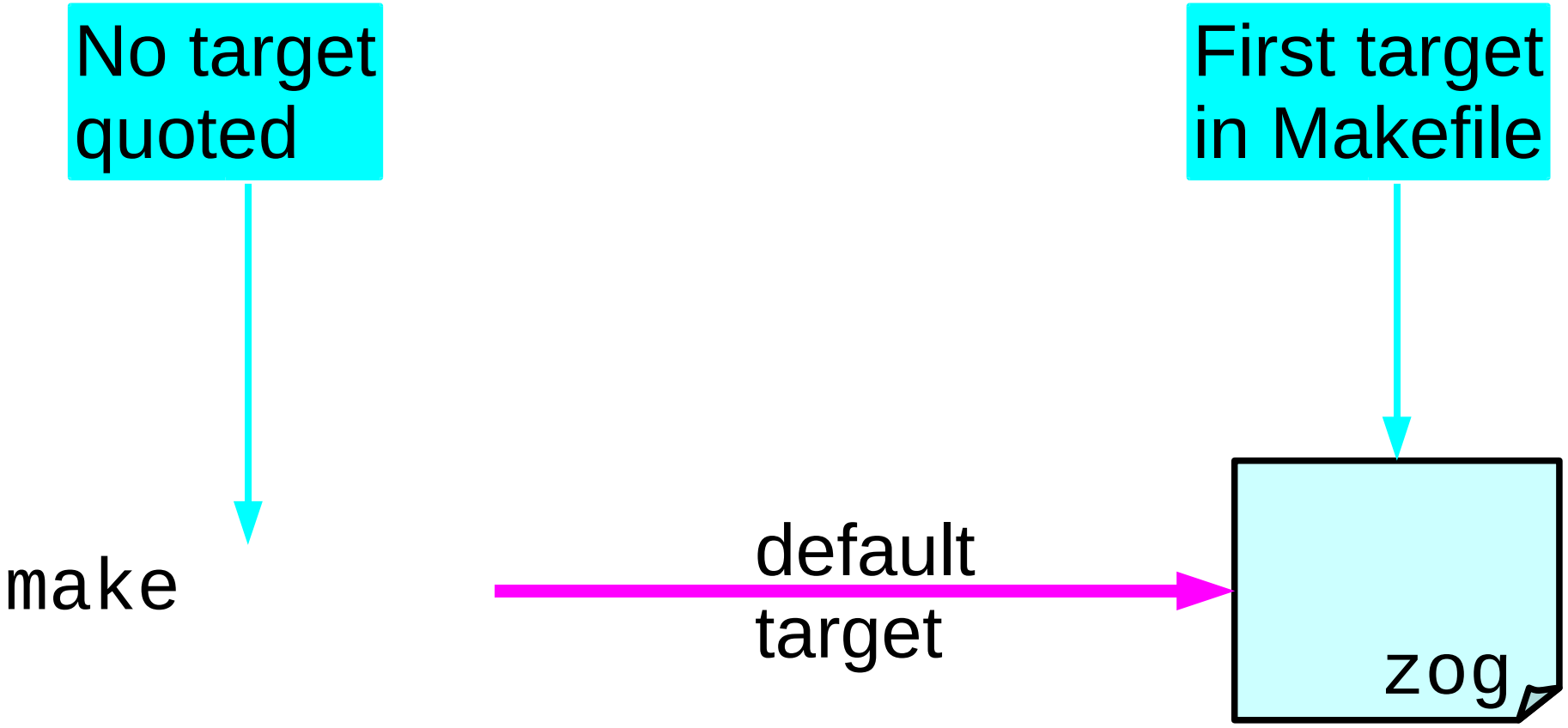
one.o two.o ...

flub

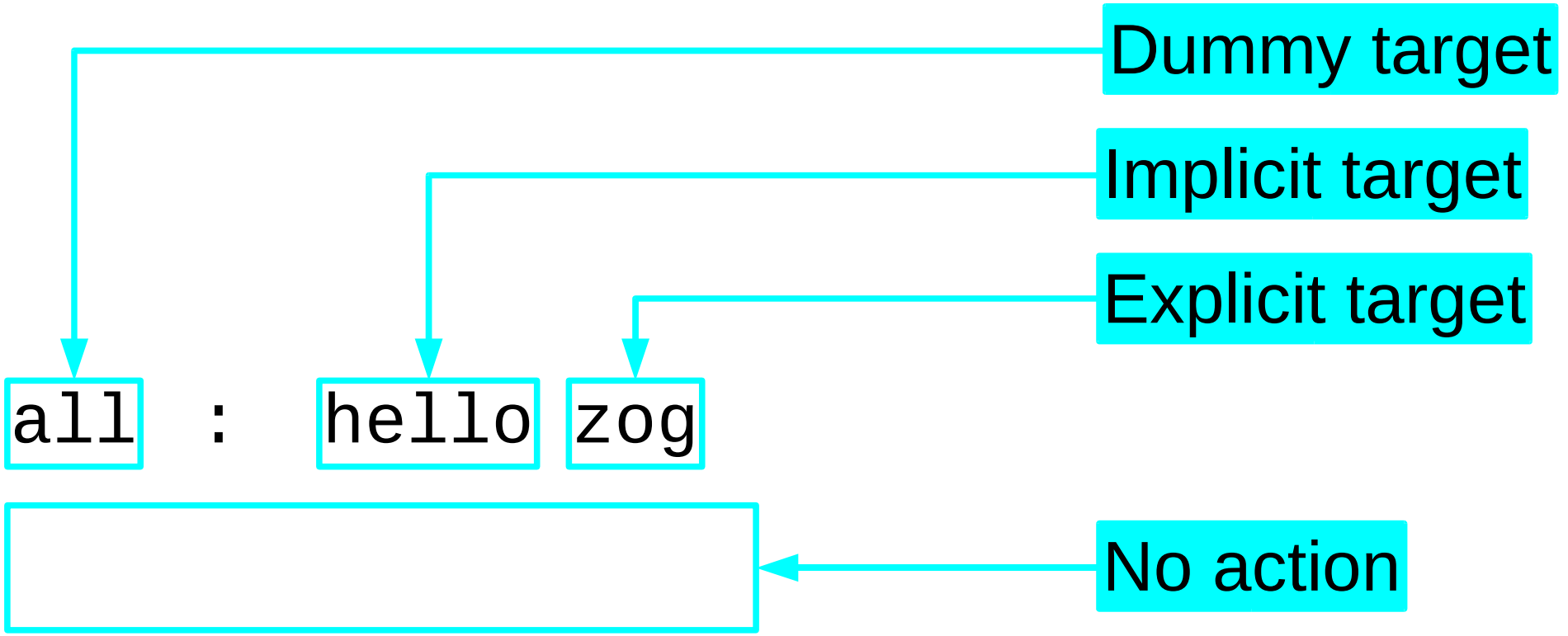
Default target?

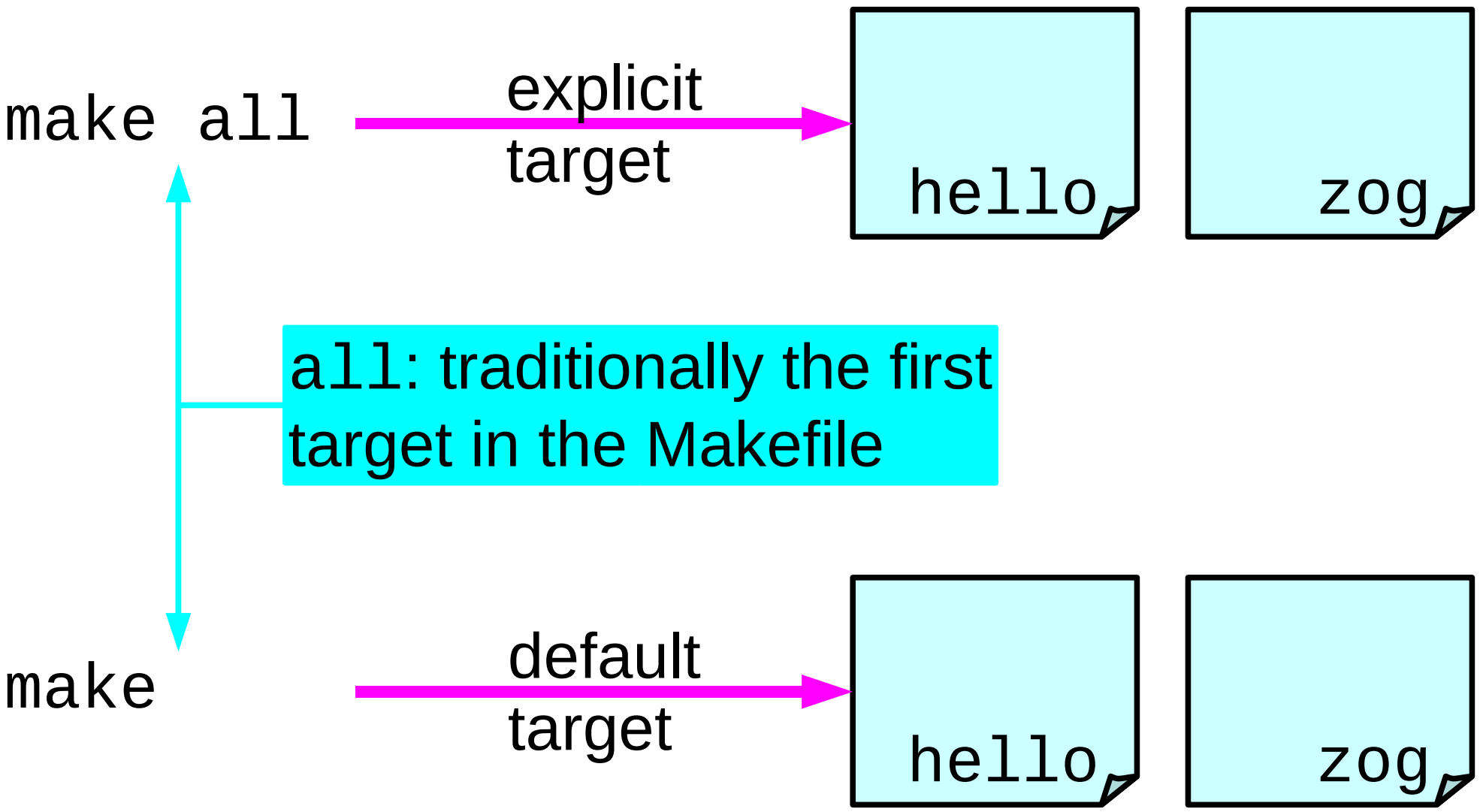


Default target = First in Makefile



1st traditional target: “all”




make all
explicit
target
hello
zog
all: traditionally the first
target in the Makefile
make
default
target
hello
zog

Worked example

```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686

all: hello zog

zog: alpha.o beta.o gamma.o
TAB  $(LINK.o) $^ $(LDLIBS) -o $@
```

Worked example

1. Start from scratch

```
$ rm -f *.o zog hello
```

2. Test the “all” target

```
$ make all
cc ... hello.c -o hello
cc ... -c -o alpha.o alpha.c
cc ... -c -o beta.o beta.c
cc ... -c -o gamma.o gamma.c
cc ... alpha.o beta.o gamma.o -o zog
```

Worked example

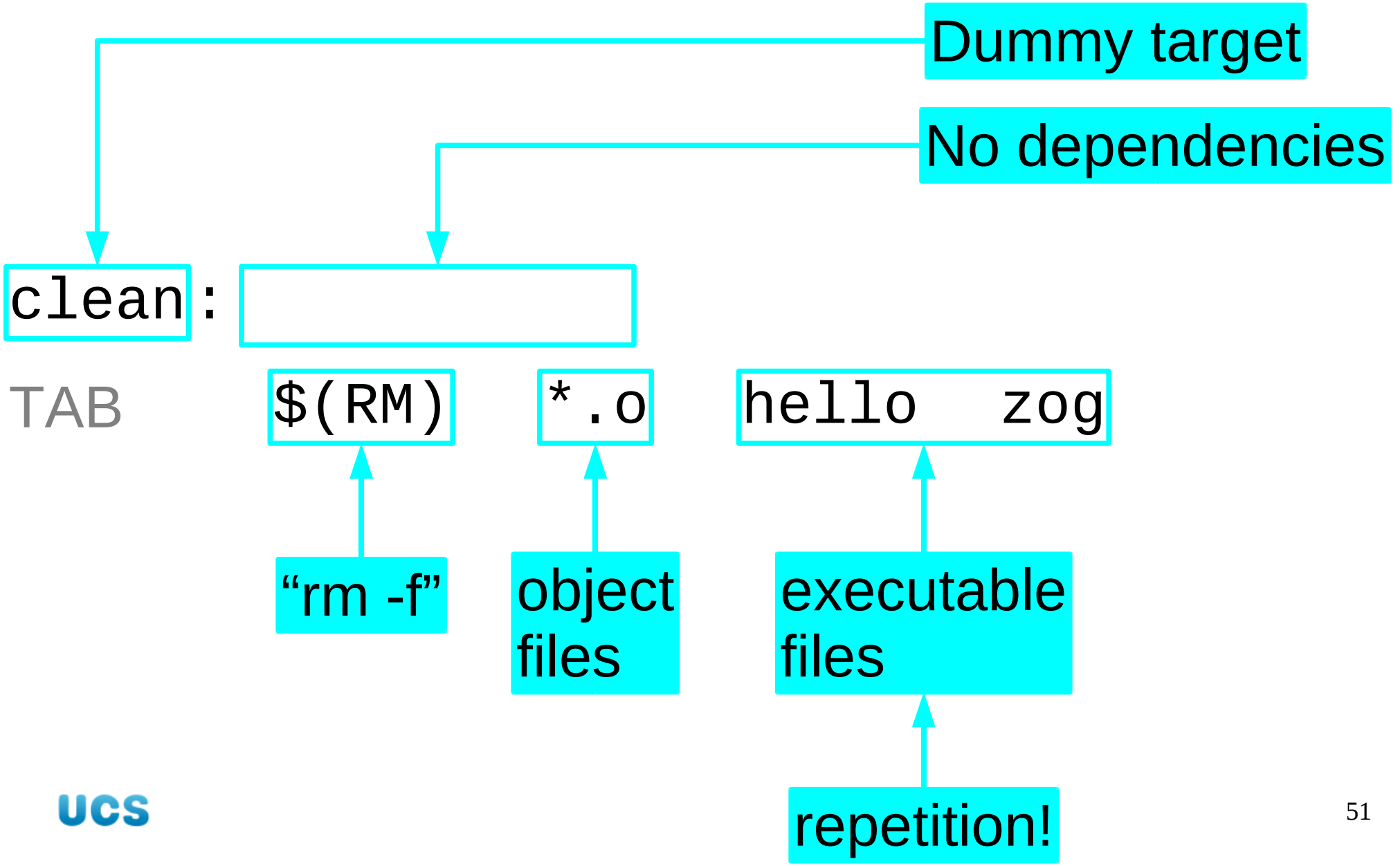
3. Start from scratch again

```
$ rm -f *.o zog hello
```

4. Test the default target

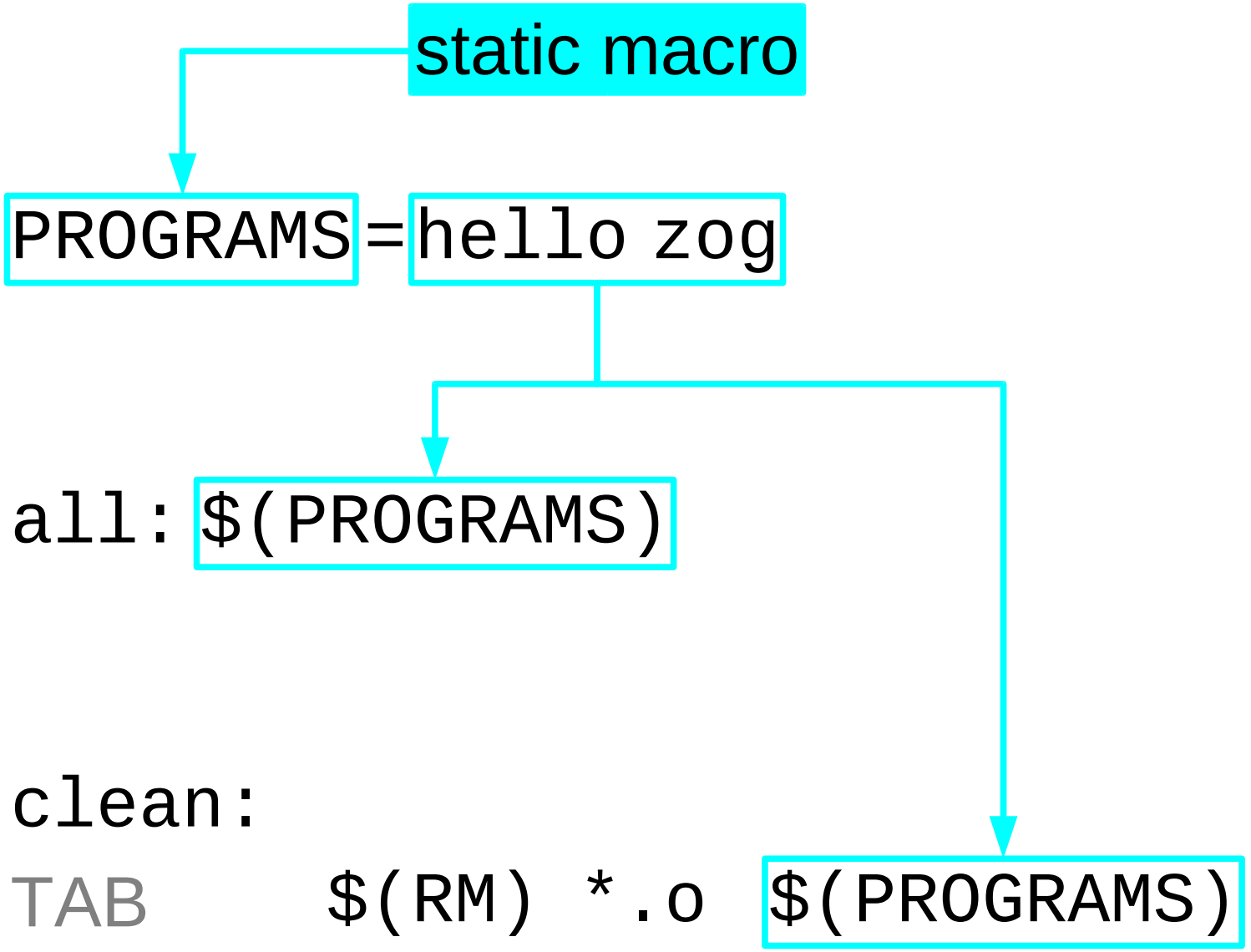
```
$ make
cc ... hello.c -o hello
cc ... -c -o alpha.o alpha.c
cc ... -c -o beta.o beta.c
cc ... -c -o gamma.o gamma.c
cc ... alpha.o beta.o gamma.o -o zog
```

2nd traditional target: “clean”



Repetition

```
all: hello zog

clean:
TAB     $(RM) *.o  hello zog
```


static macro
PROGRAMS=hello zog
all: $(PROGRAMS)
clean:
TAB $(RM) *.o $(PROGRAMS)

Worked example

```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686

PROGRAMS=hello zog

all: $(PROGRAMS)

clean:
	$(RM) *.o $(PROGRAMS)

zog: alpha.o beta.o gamma.o
…
```

Worked example

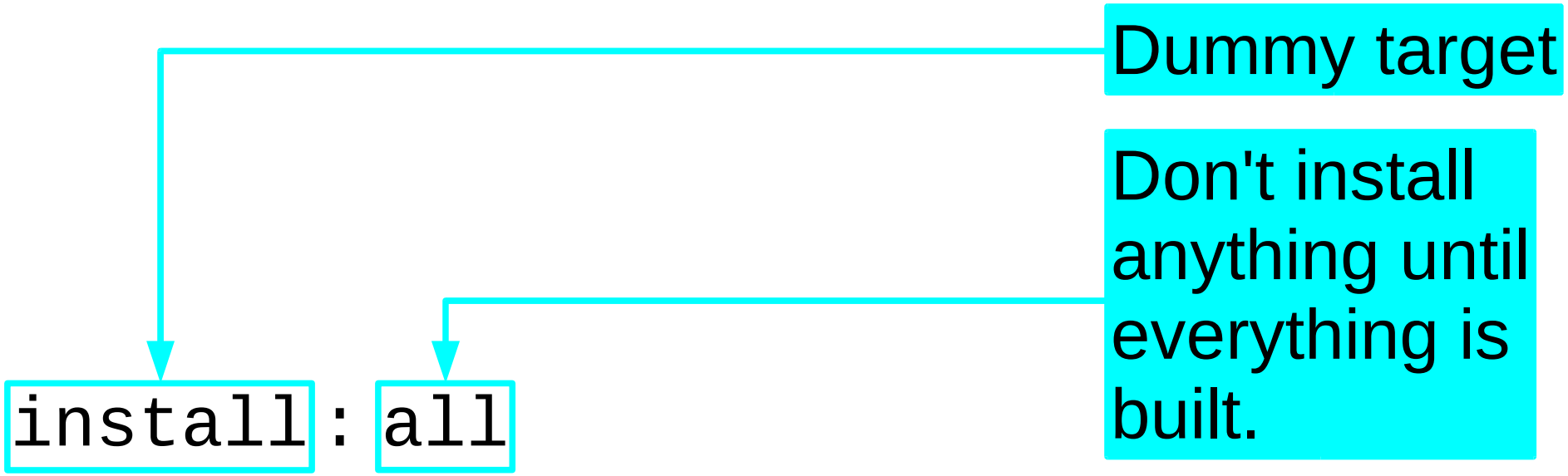
1. Start from scratch.

```
$ make clean
rm -f *.o hello zog
```

2. Create everything.

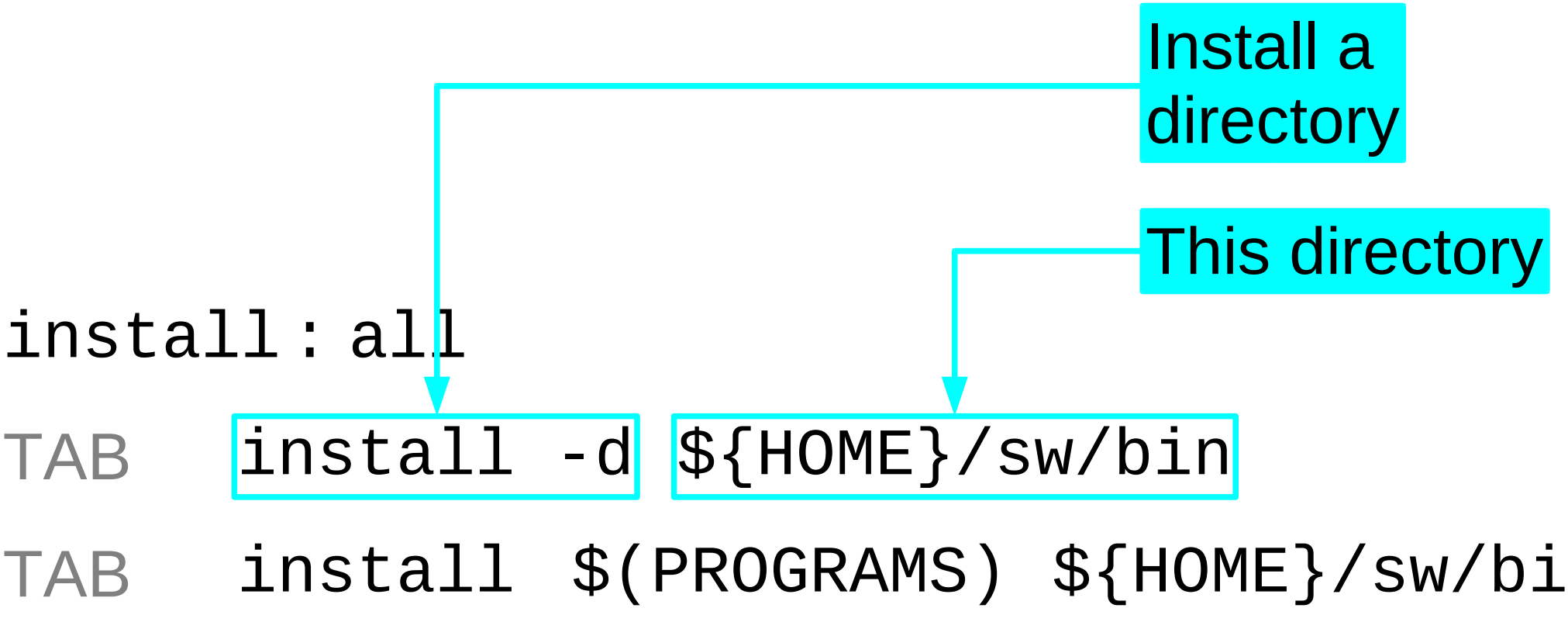
```
$ make
```

3rd traditional target: “install”

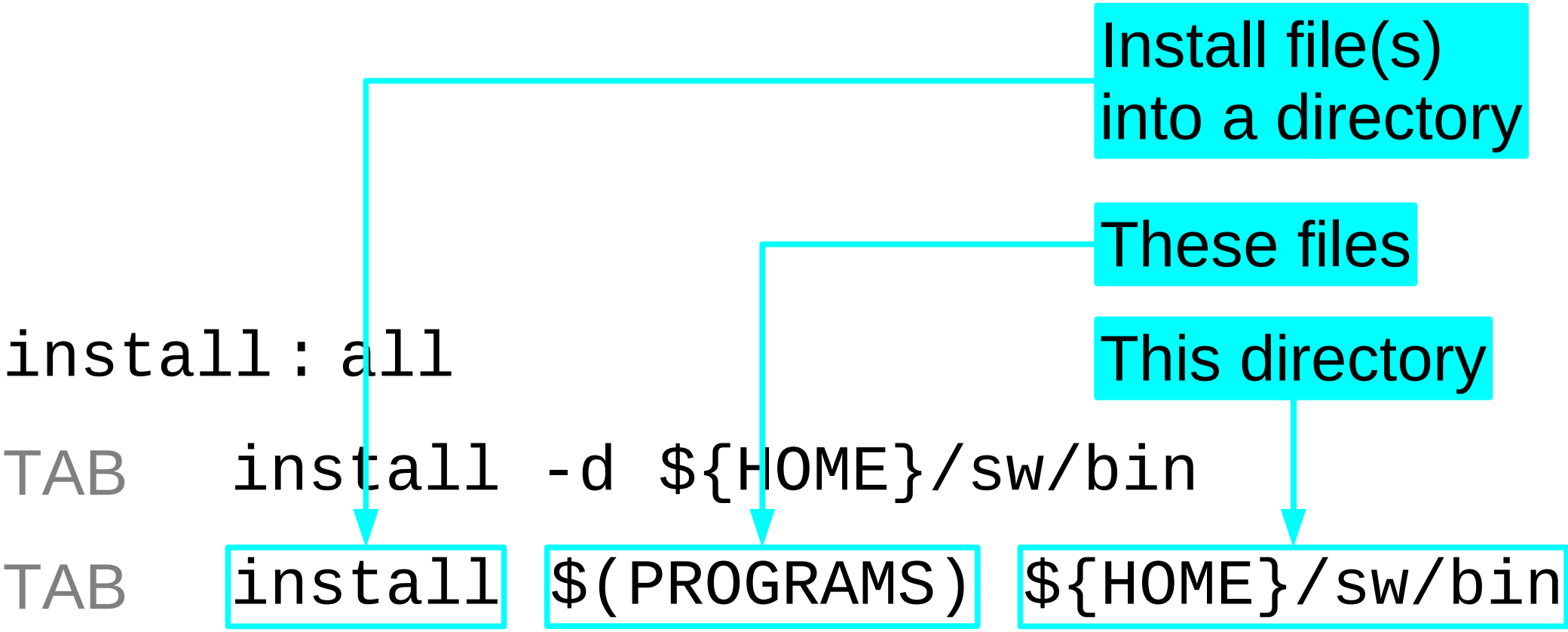



```
TAB    install -d ${HOME}/sw/bin
TAB    install  $(PROGRAMS) ${HOME}/sw/bin
```

3rd traditional target: “install”



3rd traditional target: “install”

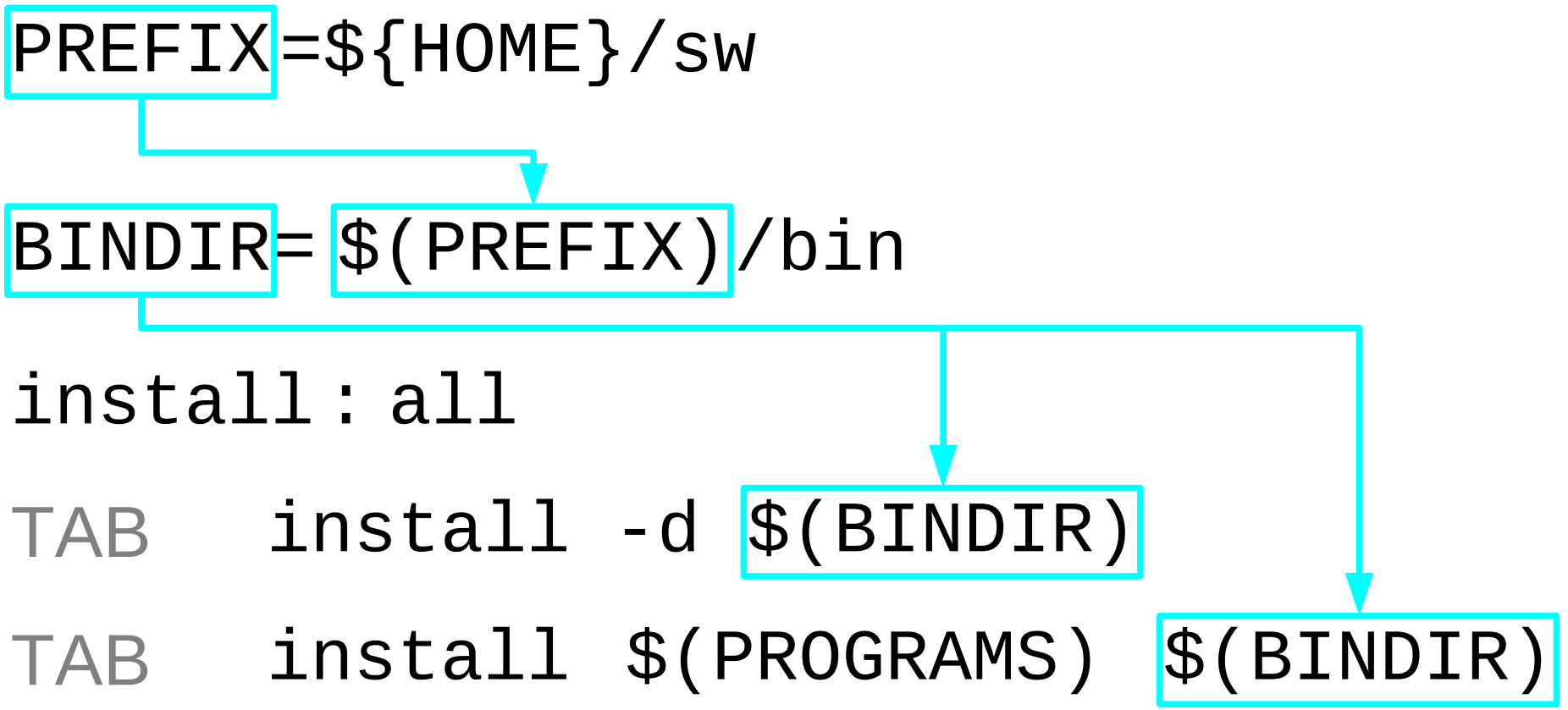


Larger example

```
install: all
    install -d ${HOME}/sw/bin
    install $(PROGRAMS) ${HOME}/sw/bin
    install -d ${HOME}/sw/lib
    install $(LIBRARIES) ${HOME}/sw/lib
    install -d ${HOME}/sw/share/man/man1
    install $(MANPAGES) \
${HOME}/sw/share/man/man1
```

Repetition

```
install: all
TAB    install -d ${HOME}/sw/bin
TAB    install $(PROGRAMS) ${HOME}/sw/bin
```


PREFIX=${HOME}/sw
BINDIR= $(PREFIX)/bin
install : all
TAB install -d $(BINDIR)
TAB install $(PROGRAMS) $(BINDIR)

Larger example

```
PREFIX=${HOME}/sw

BINDIR=$(PREFIX)/bin
LIBDIR=$(PREFIX)/lib
MANDIR=$(PREFIX)/share/man/man1

install: all
  install -d $(BINDIR)
  install $(PROGRAMS) $(BINDIR)
  install -d $(LIBDIR)
  install $(LIBRARIES)  $(LIBDIR)
...
```

Updating our Makefile

```
…
PREFIX=${HOME}/sw
BINDIR=$(PREFIX)/bin

PROGRAMS=hello zog

all: $(PROGRAMS)

install: all
TAB   install -d $(BINDIR)
TAB   install $(PROGRAMS) $(BINDIR)
…
```

Worked example

```
$ make install
cc ... hello.c   -o hello
cc ... -c -o alpha.o alpha.c
cc ... -c -o beta.o beta.c
cc ... -c -o gamma.o gamma.c
cc ... alpha.o beta.o gamma.o  -o zog
install -d /home/rjd4/sw/bin
install hello zog /home/rjd4/sw/bin
```

Principles for a good Makefile

1. Necessity
2. Consistency
3. No redundancy
4. Standard targets

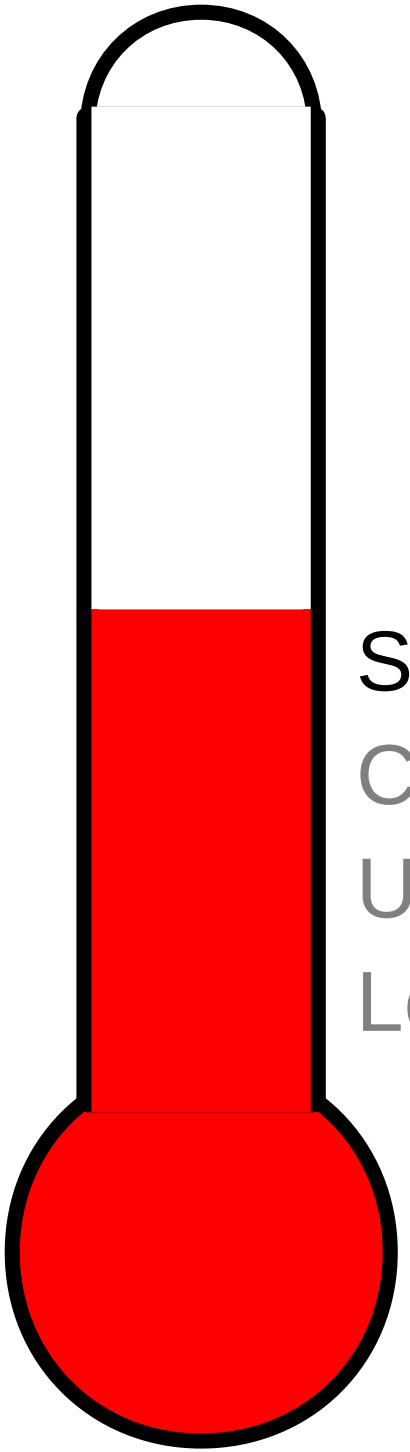
`Makefile`

`make`

built-in rules

`Makefile`

static macros
explicit rules
automatic macros
standard targets



Simple make
Configured builds
Unpacking
Location

Coffee break

“Coffee is a beverage which puts one to sleep if not drunk.”

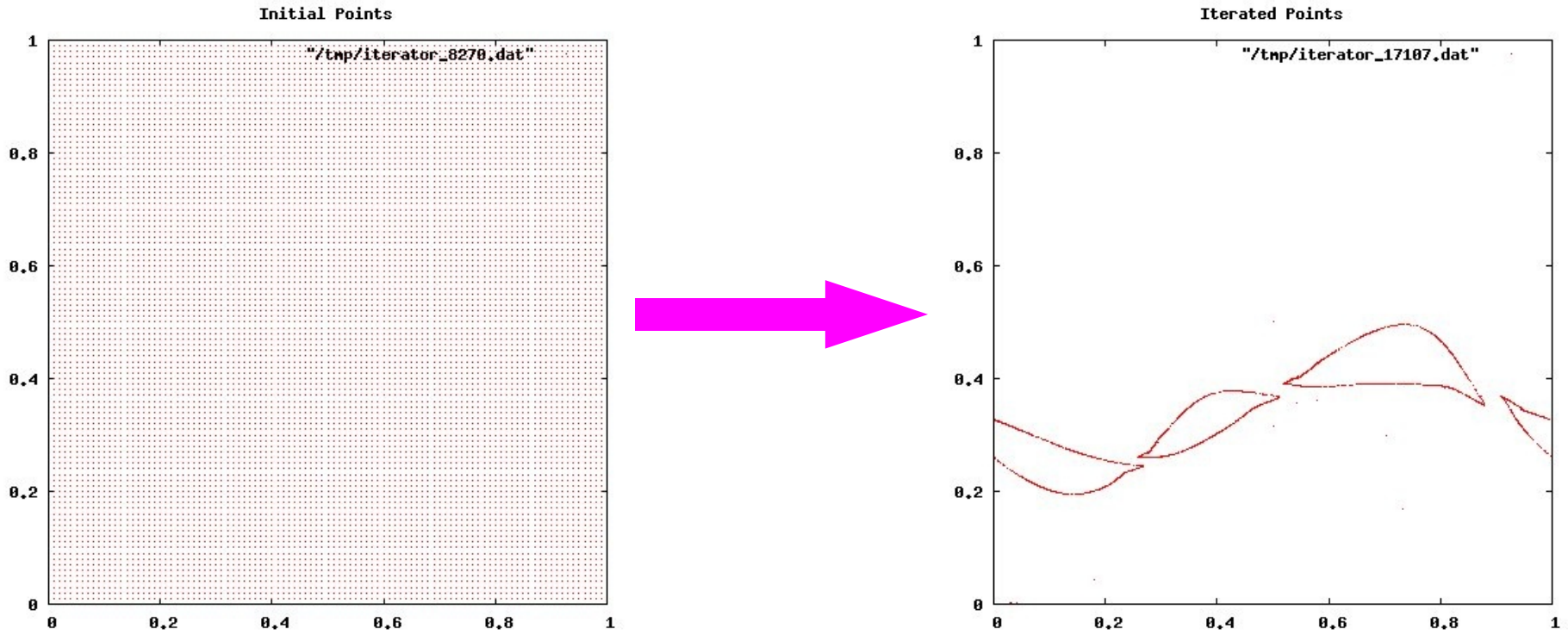
Alphonse Allais

Ten minutes

Using libraries

Iterate points

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x+y \\ y\left(1+\epsilon\, sin\left(2\,\pi\, x\right)\right) \end{pmatrix}$$



Worked example

1. `/ux/Lessons/Building` → `/tmp/building`

   `iterator.c`

   `iterator.sh`

Worked example

2. `$ make iterator`

```
cc … iterator.c   -o iterator
/tmp/cco7S3Aa.o: In function `main':
iterator.c:(.text+0x15b):
undefined reference to `sin'
collect2: ld returned 1 exit status
make : ***[iterator] Error 1
```

Doesn't understand the sin() function!

p.s. Our first error from cc

```
$ make iterator
```

```
cc … iterator.c   -o iterator
/tmp/cco7S3Aa.o: In function `main':
iterator.c:(.text+0x15b):
undefined reference to `sin'
collect2: ld returned 1 exit status
make : ***[iterator] Error 1
```

Build program straight from the source file

Temporary object file

p.s. Our first error from cc

```
$ make iterator

cc … iterator.c   -o iterator
/tmp/cco7S3Aa.o: In function `main':
iterator.c:(.text+0x15b):
undefined reference to `sin'
collect2: ld returned 1 exit status
make : ***[iterator] Error 1
```

Error in this function

Defined in this file

p.s. Our first error from cc

```
$ make iterator

cc … iterator.c   -o iterator
/tmp/cco7S3Aa.o: In function `main':
iterator.c:(.text+0x15b):
undefined reference to `sin'
collect2: ld returned 1 exit status
make : ***[iterator] Error 1
```

The error message

p.s. Our first error from cc

```
$ make iterator

cc … iterator.c   -o iterator
/tmp/cco7S3Aa.o: In function `main':
iterator.c:(.text+0x15b):
undefined reference to `sin'
collect2: ld returned 1 exit status
make : ***[iterator] Error 1
```

“ld” is the linker (a.k.a. “loader”)

p.s. Our first error from cc

```
$ make iterator

cc … iterator.c   -o iterator
/tmp/cco7S3Aa.o: In function `main':
iterator.c:(.text+0x15b):
undefined reference to `sin'
collect2: ld returned 1 exit status
make : ***[iterator] Error 1
```

An error from make

Building this target

make's error message

Worked example

```
$ make iterator

cc … iterator.c   -o iterator
/tmp/cco7S3Aa.o: In function `main':
iterator.c:(.text+0x15b):
undefined reference to `sin'
collect2: ld returned 1 exit status
make : ***[iterator] Error 1
```

Doesn't understand the sin() function!

System libraries

`libc` The “ C” library.

1,392 functions.

Linked automatically by the C compiler.

We've never had to worry about it.

```
read()
write()
snprintf()
execle()
fork()
inet_addr()
...
```

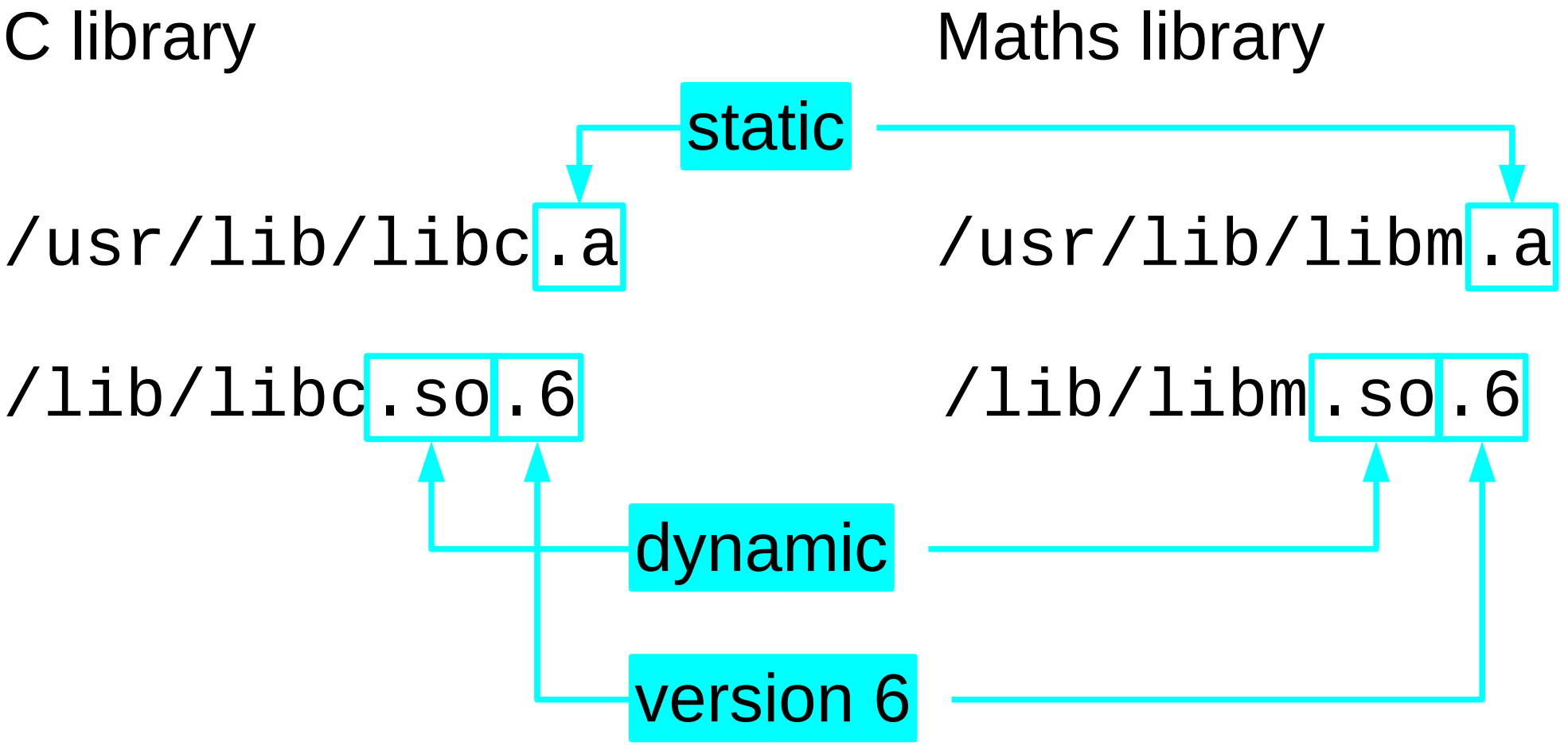
System libraries

401 functions.

Not linked automatically by the C compiler.

We need to link this library explicitly.

System libraries: the files



System libraries: cc options

C library

`/usr/lib/libc.a`

`/lib/libc.so.6`

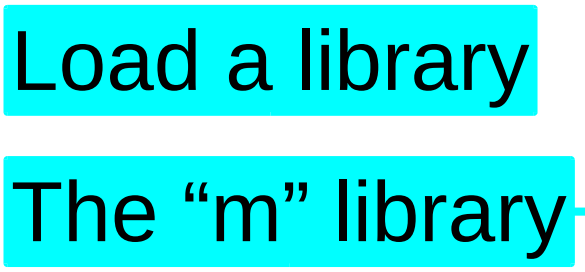
No options required

Maths library

`/usr/lib/libm.a`

`/lib/libm.so.6`

`-lm`



System libraries: the Makefile

C library

```
LDLIBS=
```

Empty value: the default

Maths library

```
LDLIBS=-lm
```

Link in lib**m**

Worked example

3. Edit `Makefile`

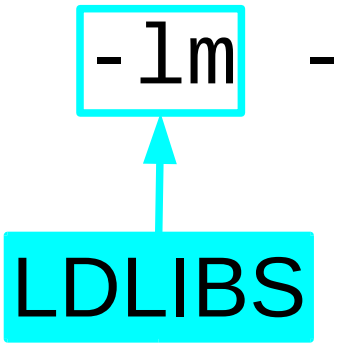
```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686
LDLIBS=-lm

PREFIX=${HOME}/sw
BINDIR=$(PREFIX)/bin

PROGRAMS=hello zog iterator

all: $(PROGRAMS)
…
```

Adds the maths library to *every* program built.

Worked example

4. `$ make iterator`

```
cc … iterator.c -lm -o iterator
```

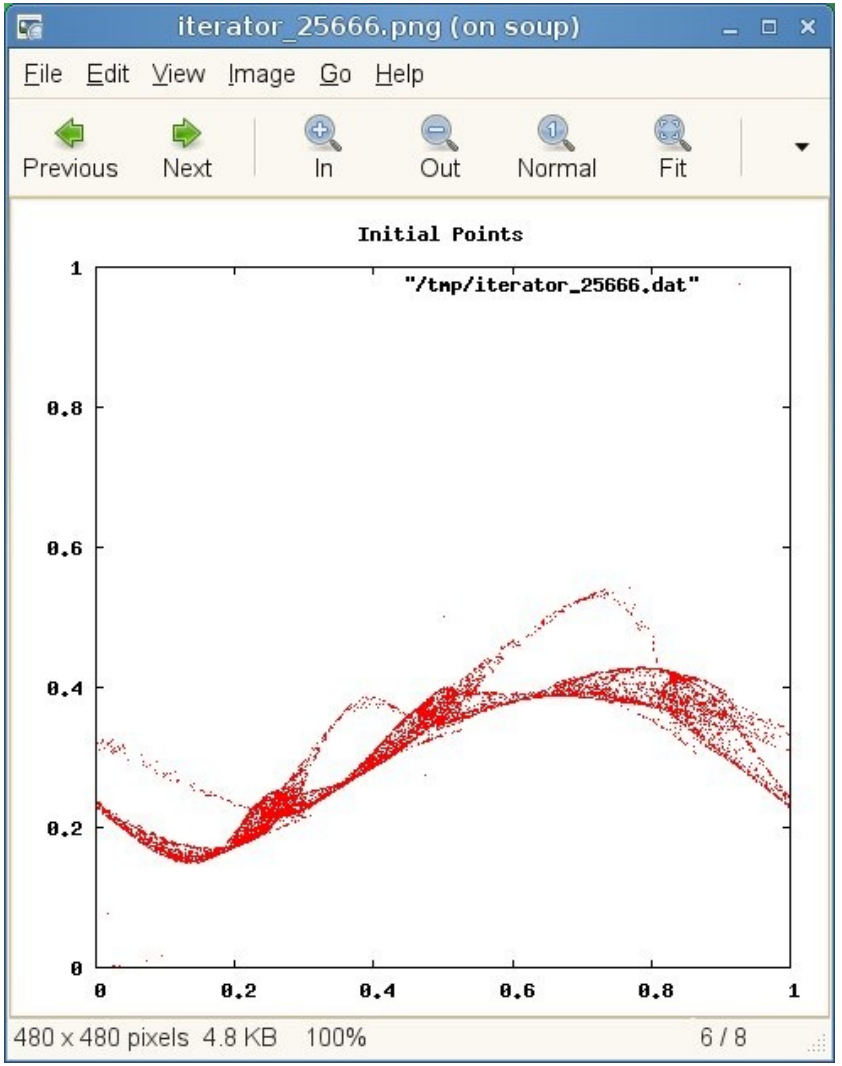


5. `$ ls -l iterator`

```
-rwxr-xr-x 1 … iterator
```

Worked example

6. `$ ./iterator.sh 0.60`



Other libraries

Maths library

```
/usr/lib/libm.a
/lib/libm.so.6

-lm
```

Other library

```
/usr/lib/libthing.a
/lib/libthing.so.n

-lthing
```

Static linking

Rarely worth it

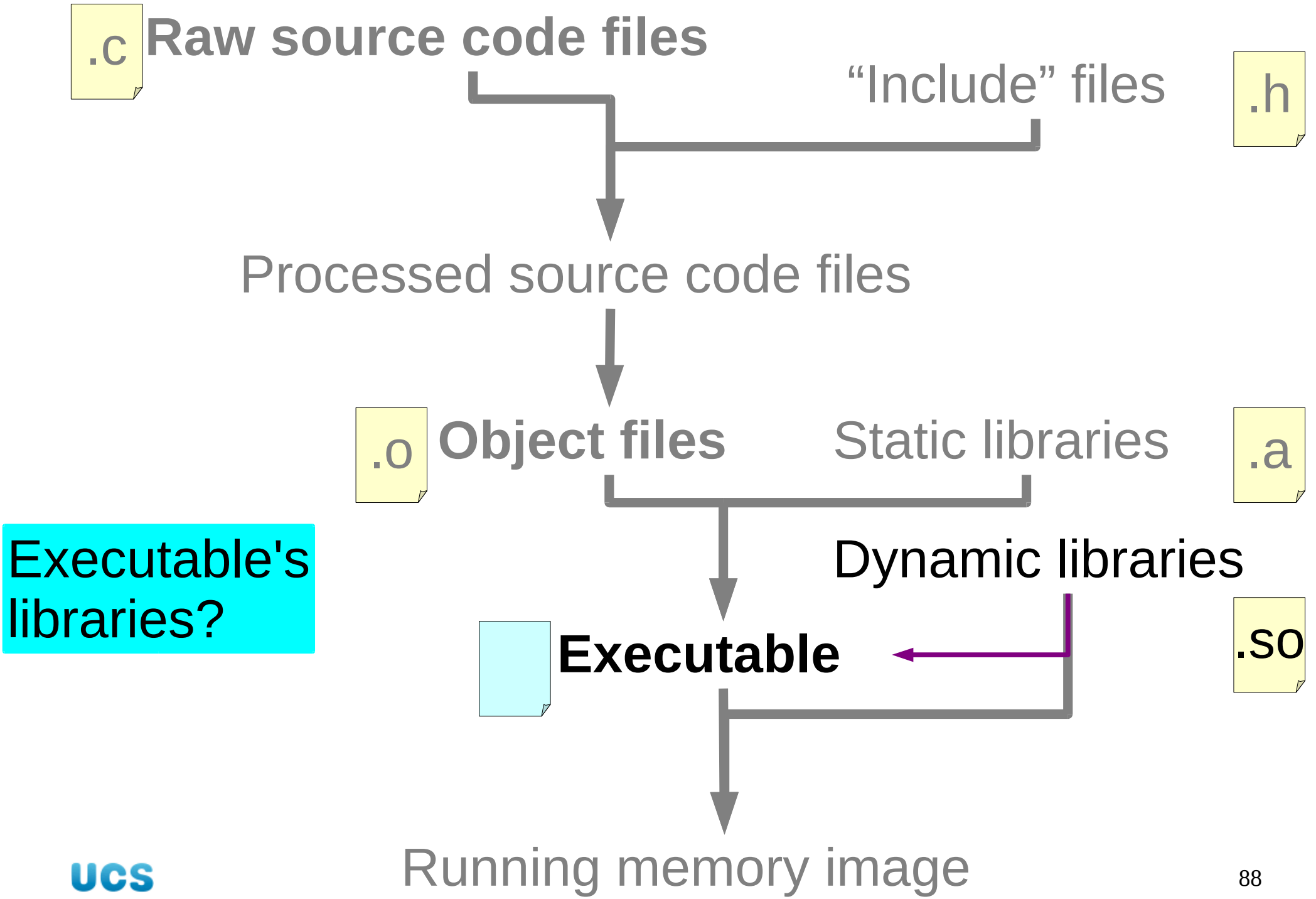
```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686
LDLIBS=-lm
LDFLAGS=-static

PREFIX=${HOME}/sw
BINDIR=$(PREFIX)/bin

PROGRAMS=hello zog iterator


…
```

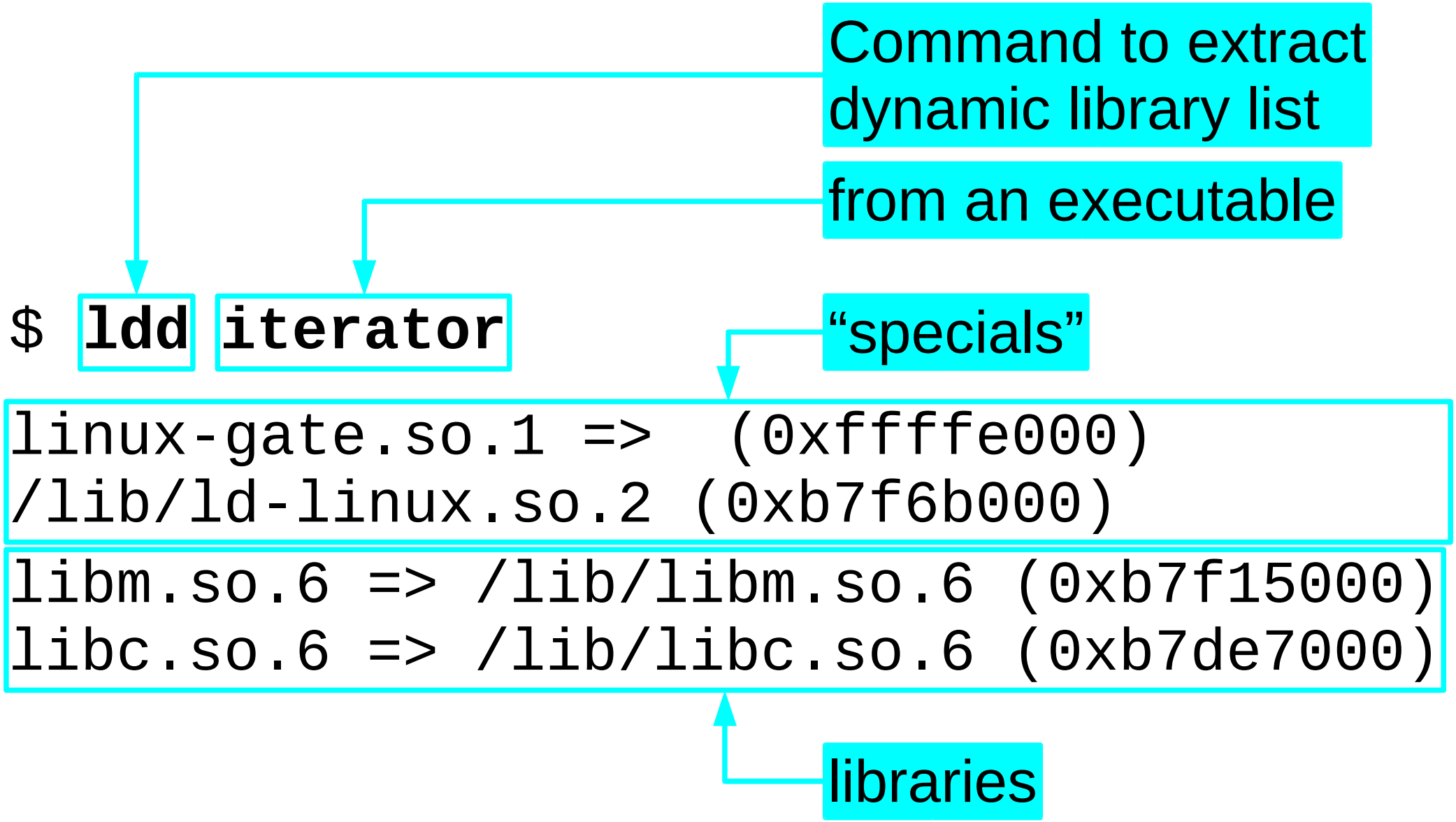
Size difference

```
$  ls -l iterator
```

Dynamic: 11,065 bytes

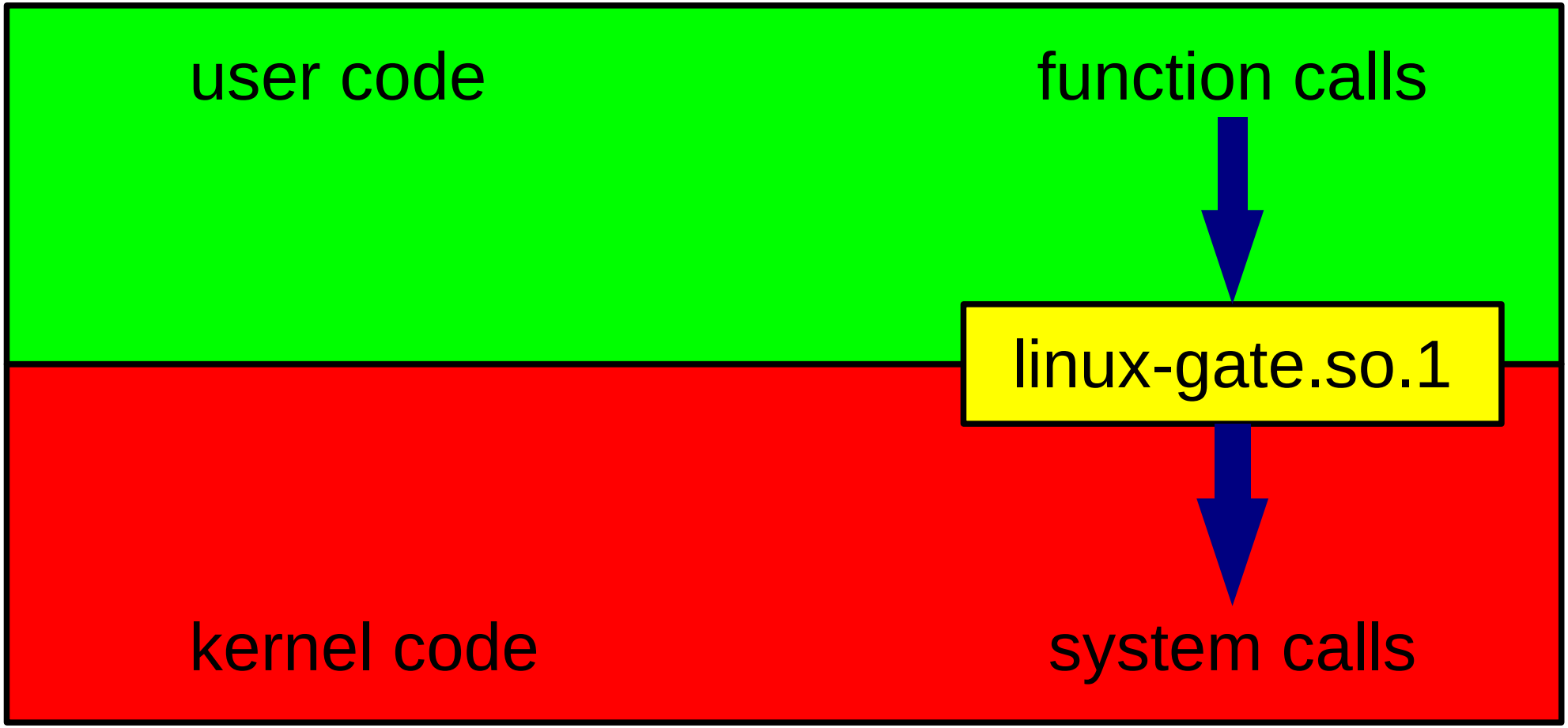
Static: 2,338,249 bytes!


.c
Raw source code files
“Include” files
.h
Processed source code files
.o
Object files
Static libraries
.a
Executable's libraries?
Dynamic libraries
.so
Executable
Running memory image


Command to extract dynamic library list
from an executable
$ ldd iterator
“specials”
linux-gate.so.1 => (0xffffe000)
/lib/ld-linux.so.2 (0xb7f6b000)
libm.so.6 => /lib/libm.so.6 (0xb7f15000)
libc.so.6 => /lib/libc.so.6 (0xb7de7000)
libraries

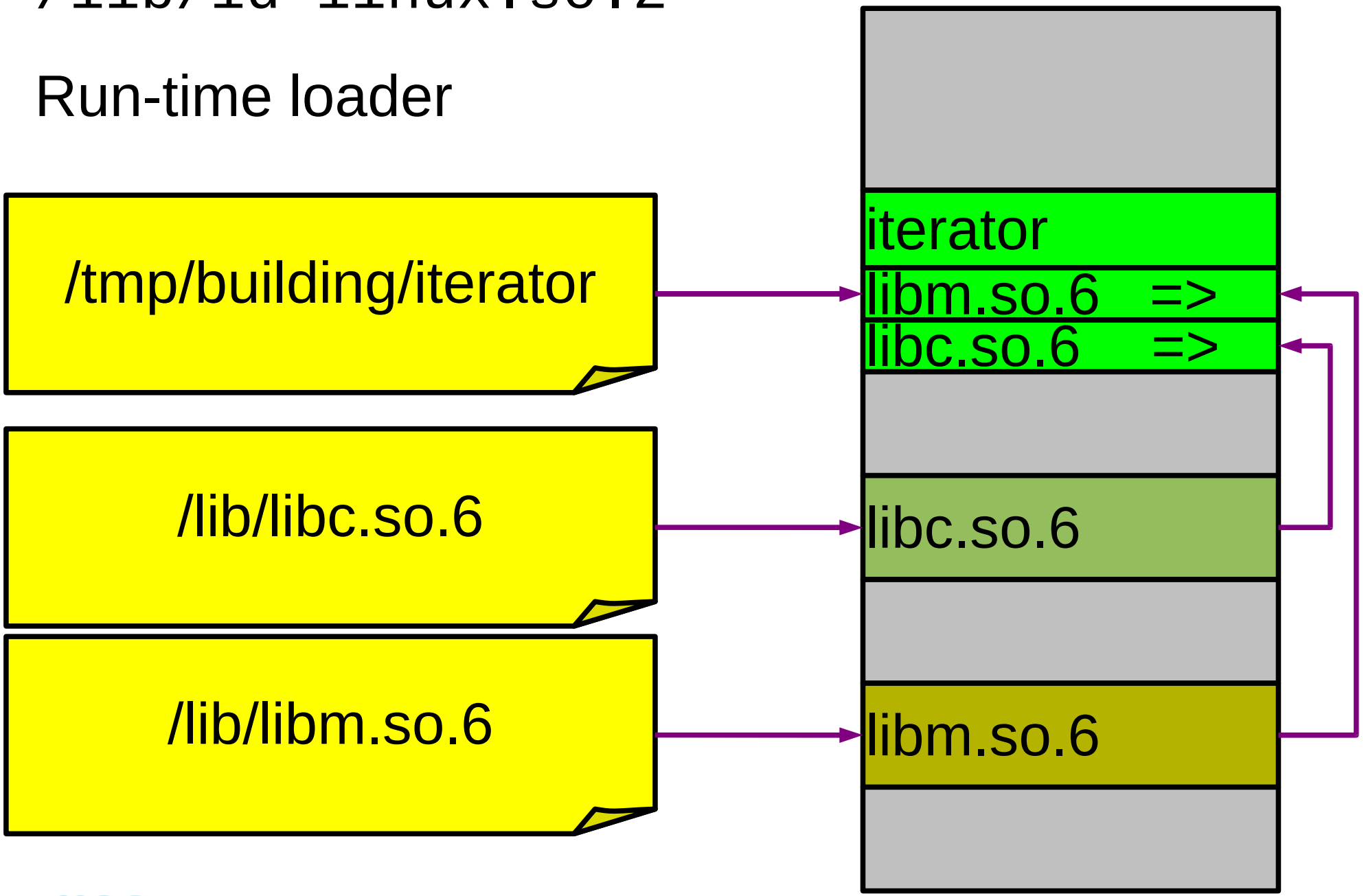
`linux-gate.so.1`

“Virtual dynamic library”



/lib/ld-linux.so.2

Run-time loader



Exercise: Building a library

"**G**NU **L**inear **P**rogramming **K**it"

`/ux/Lessons/Building/glpk-4.40.tar.gz`

1. Unpack
2. Configure
3. Build
4. Install

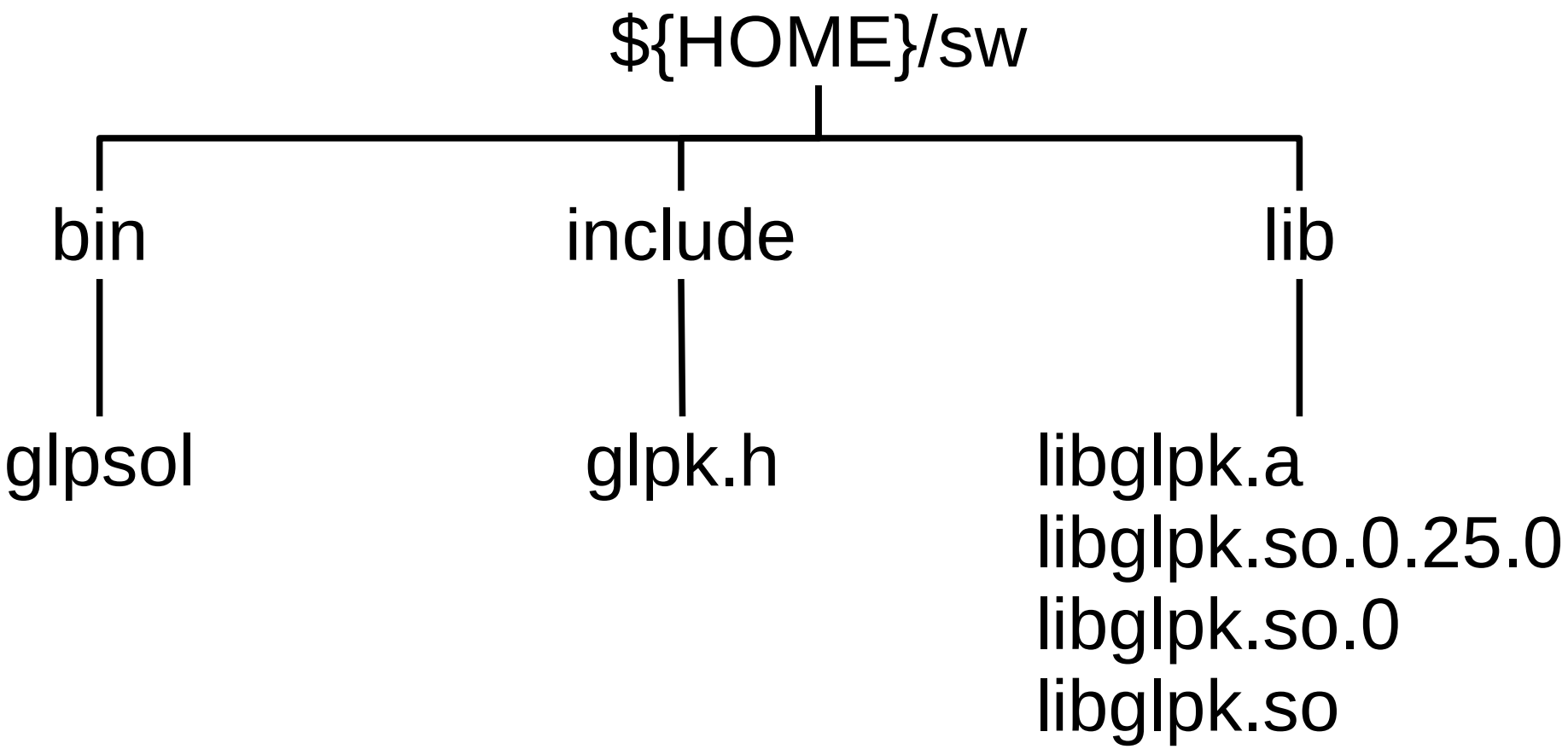
Coffee break

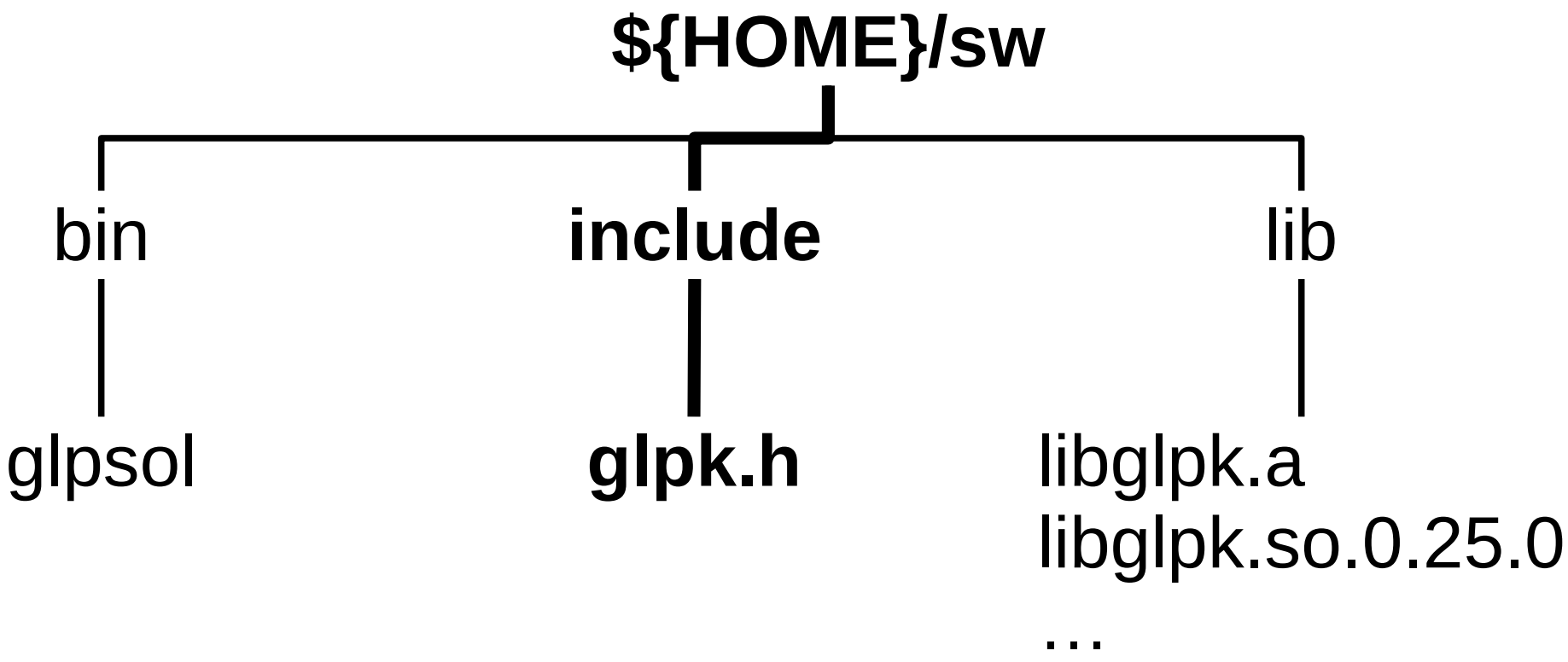
“In Seattle you haven't had enough coffee until you can thread a sewing machine while it's still running.”

Jeff Bezos

Ten minutes

UCS


${HOME}/sw
bin
include
lib
glpsol
glpk.h
libglpk.a
libglpk.so.0.25.0
libglpk.so.0
libglpk.so

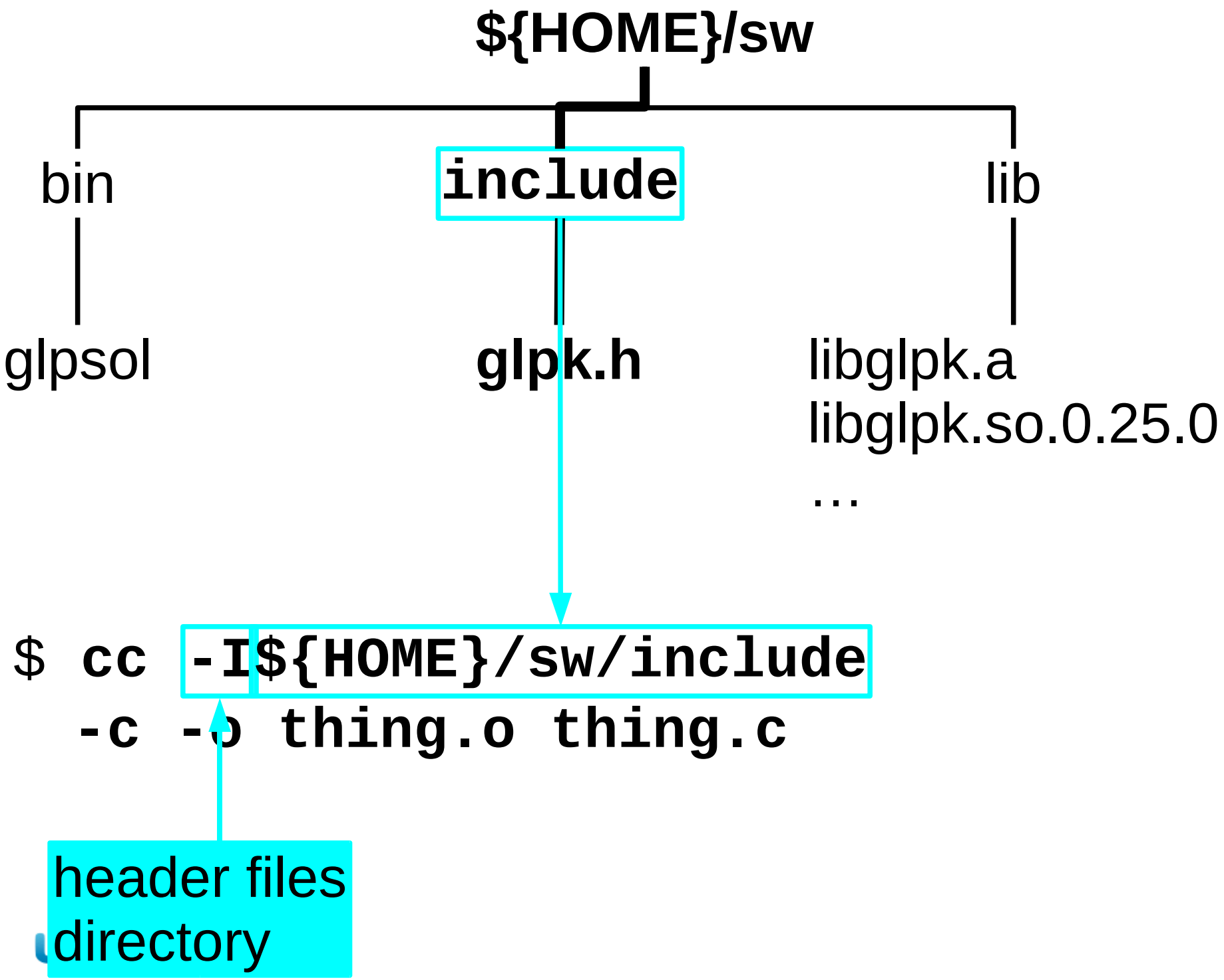


.c **Raw source code files**

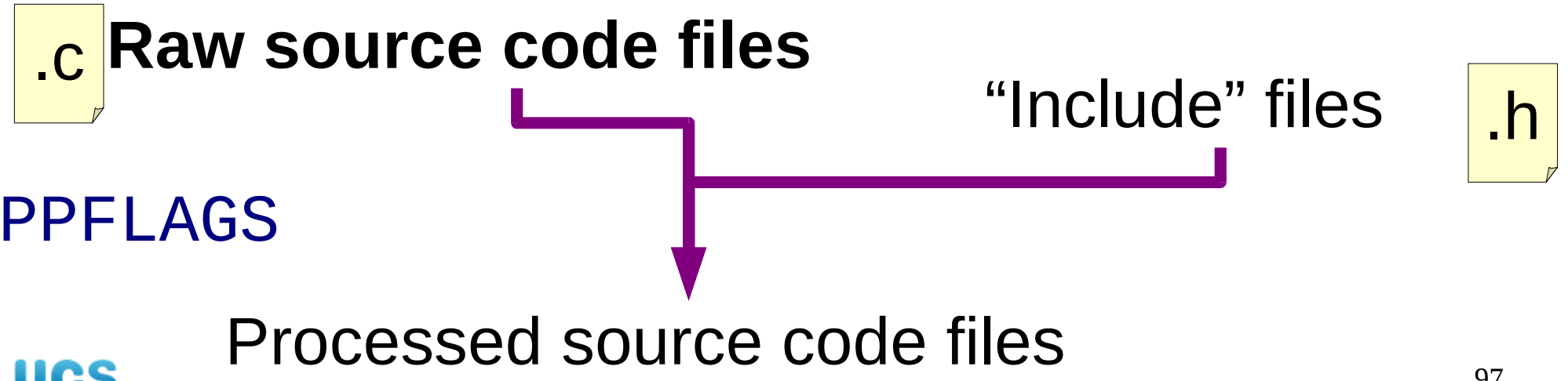
"Include" files .h

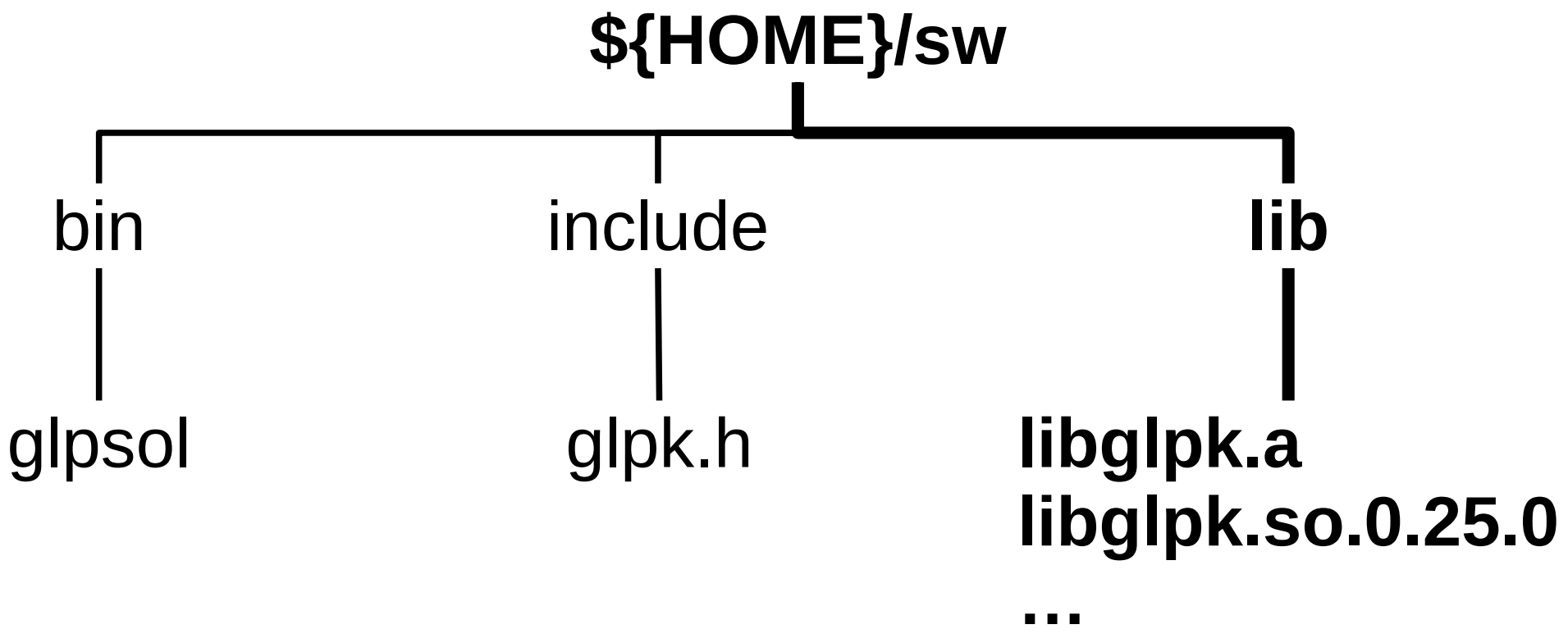
CPPFLAGS

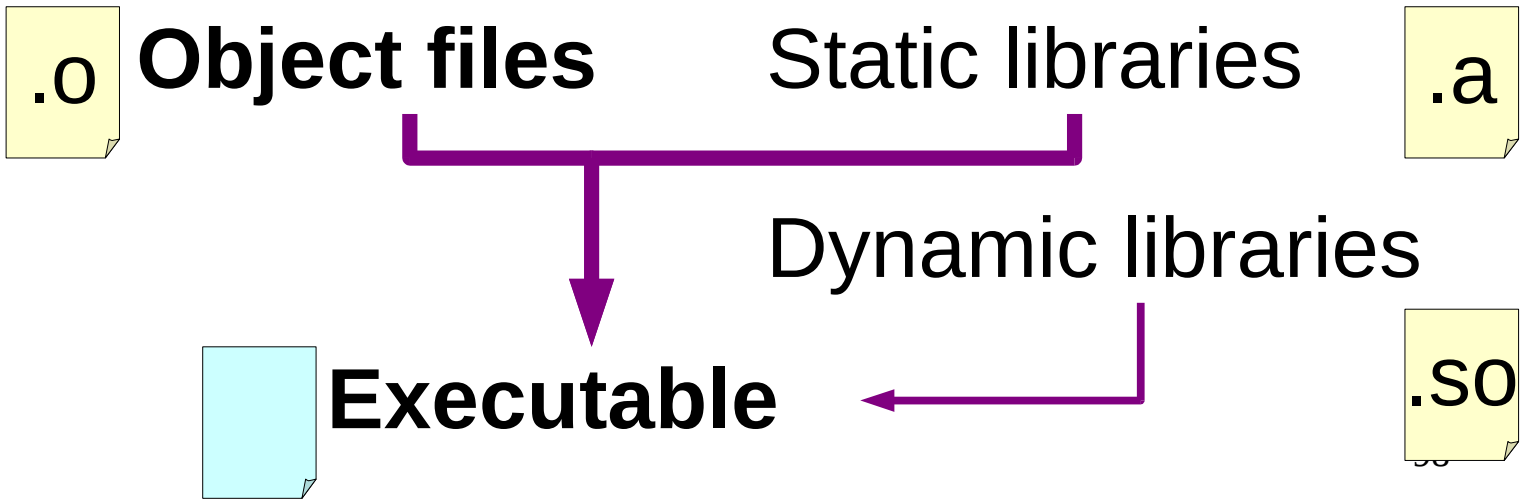
Processed source code files


${HOME}/sw
bin
include
lib
glpsol
glpk.h
libglpk.a
libglpk.so.0.25.0
...
$ cc -I${HOME}/sw/include
-c -o thing.o thing.c
header files
directory

```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686
LDLIBS=-lm

PREFIX=${HOME}/sw
BINDIR=$(PREFIX)/bin
CPPFLAGS=-I$(PREFIX)/include
…
```



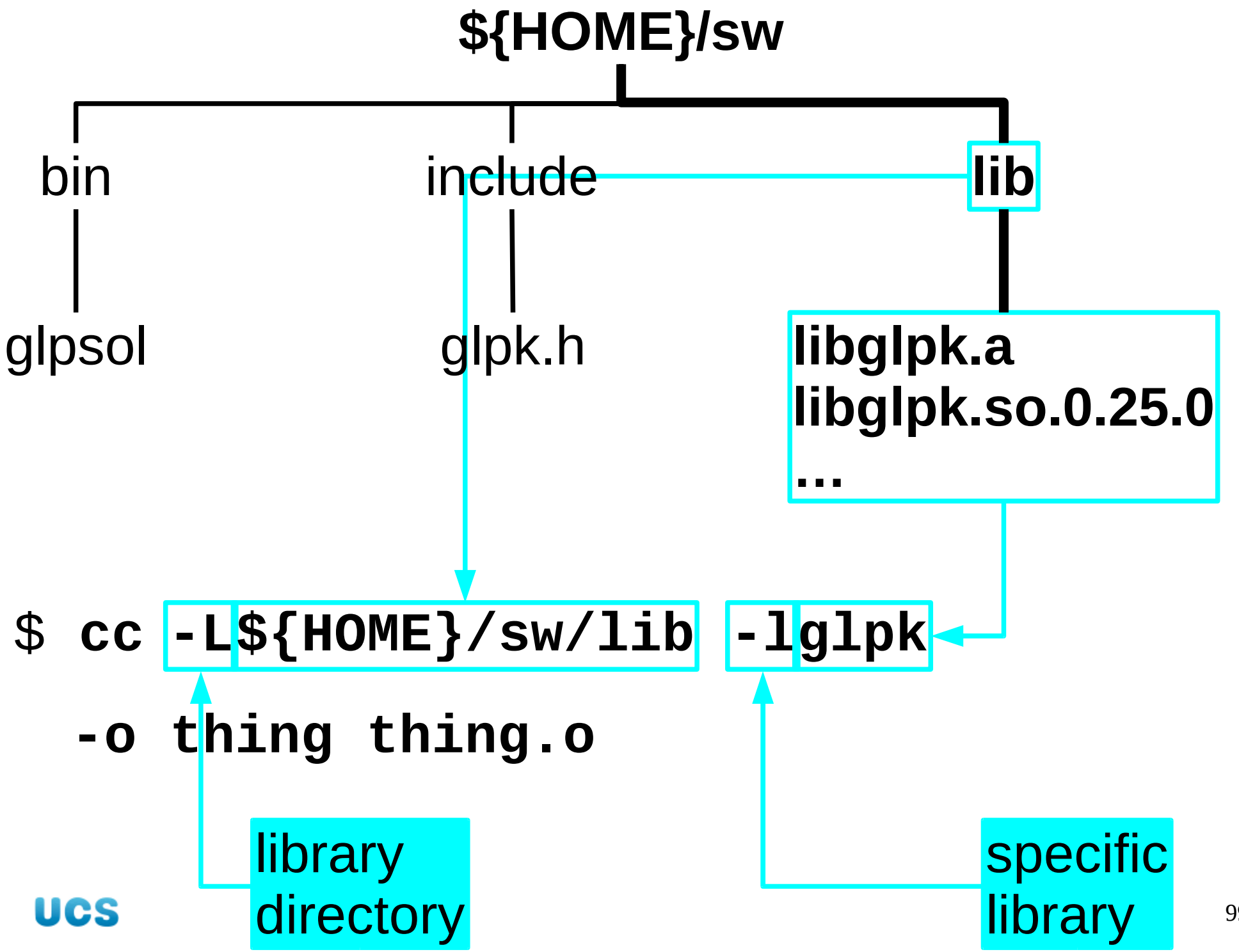




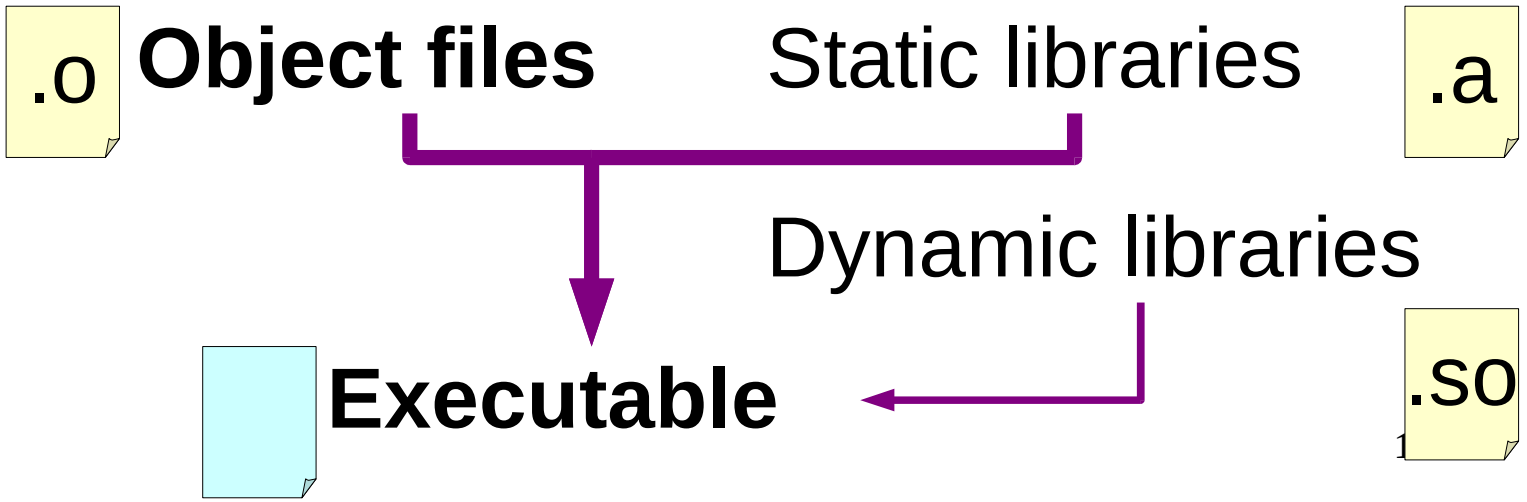
LDFLAGS

LDLIBS

UCS


${HOME}/sw
bin
include
lib
glpsol
glpk.h
libglpk.a
libglpk.so.0.25.0
...
$ cc -L${HOME}/sw/lib -lglpk
-o thing thing.o
library directory
specific library

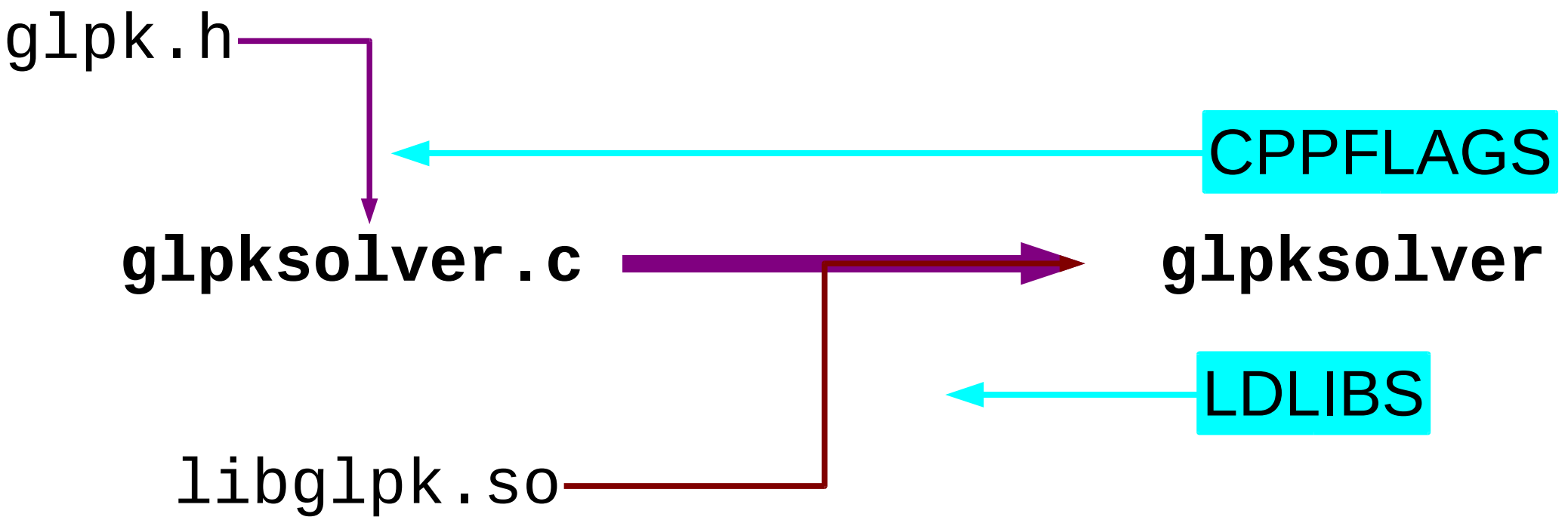
```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686

PREFIX=${HOME}/sw
BINDIR=$(PREFIX)/bin
CPPFLAGS=-I$(PREFIX)/include
LDLIBS=-L$(PREFIX)/lib -lglpk
…
```



LDFLAGS

LDLIBS

Exercise: Using the GLPK

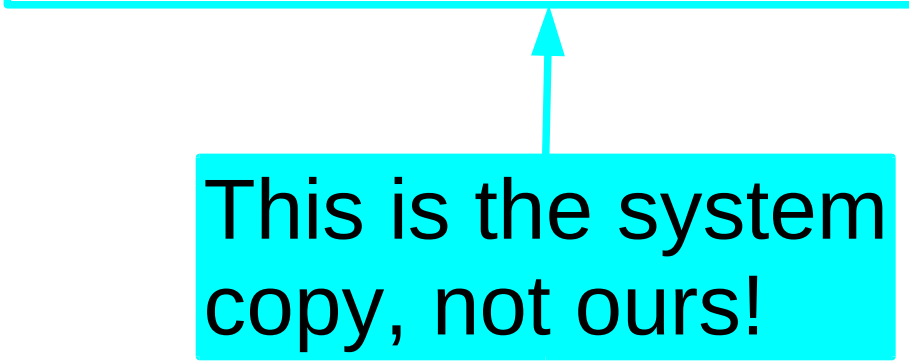


Program seems to run OK

```
$ ./glpksolver
…
OPTIMAL SOLUTION FOUND

*** Output in file graph.eps
```

But...

```
$ ldd glpksolver
linux-gate.so.1  =>  (0xffffe000)
/lib/ld-linux.so.2
libc.so.6 => /lib/libc.so.6
libm.so.6 => /lib/libm.so.6
libglpk.so.0 => /usr/lib/libglpk.so.0
```



Two copies of the library

```
/usr/lib/libglpk.so.0
/usr/lib/libglpk.so.0.16.0
```

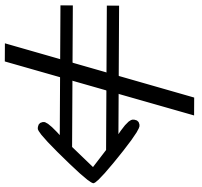
System library

```
/home/rjd4/sw/lib/libglpk.so.0
/home/rjd4/sw/lib/libglpk.so.0.25.0
```

Our library

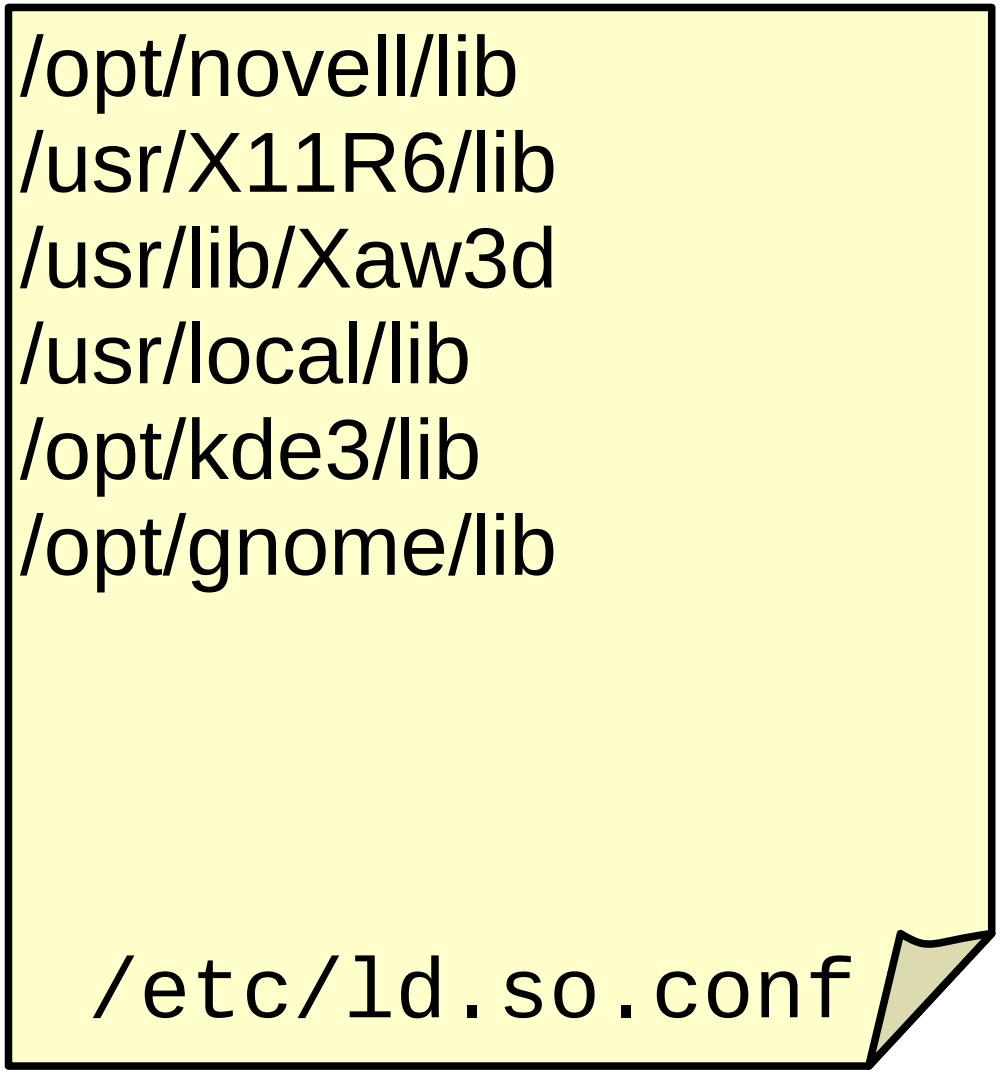
We have a newer version that we want to use.

Finding libraries at build time



Finding libraries at run time

Finding libraries at run time — 1



List of directories

No way to use ${HOME}

System libraries

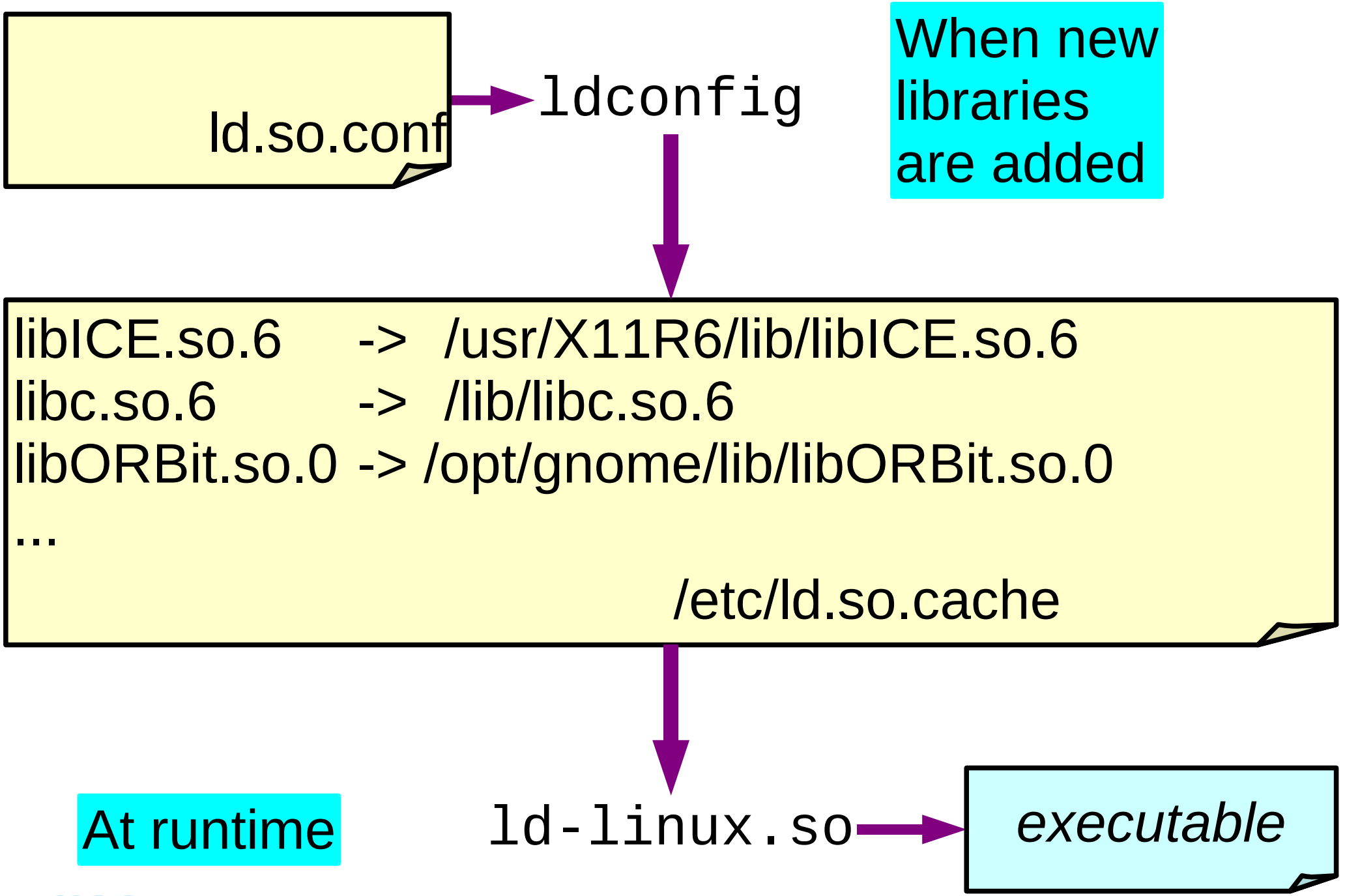
Finding libraries at run time — 2

`/lib/ld-linux.so.2`

Built-in list of directories

/lib/tls
/lib
/usr/lib/tls
/usr/lib

Still no way to use ${HOME}


ld.so.conf
ldconfig
When new libraries are added
libICE.so.6 -> /usr/X11R6/lib/libICE.so.6
libc.so.6 -> /lib/libc.so.6
libORBit.so.0 -> /opt/gnome/lib/libORBit.so.0
...
/etc/ld.so.cache
At runtime
ld-linux.so
executable

Personal extras

1. `LD_LIBRARY_PATH`
2. `/etc/ld.so.conf`
3. Built-ins

Exercise: Update .bashrc

1. `bashrc3` `${HOME}/.bashrc`

export LD_LIBRARY_PATH="${HOME}/sw/lib"

Exercise: Update .bashrc

2. Existing terminal window:

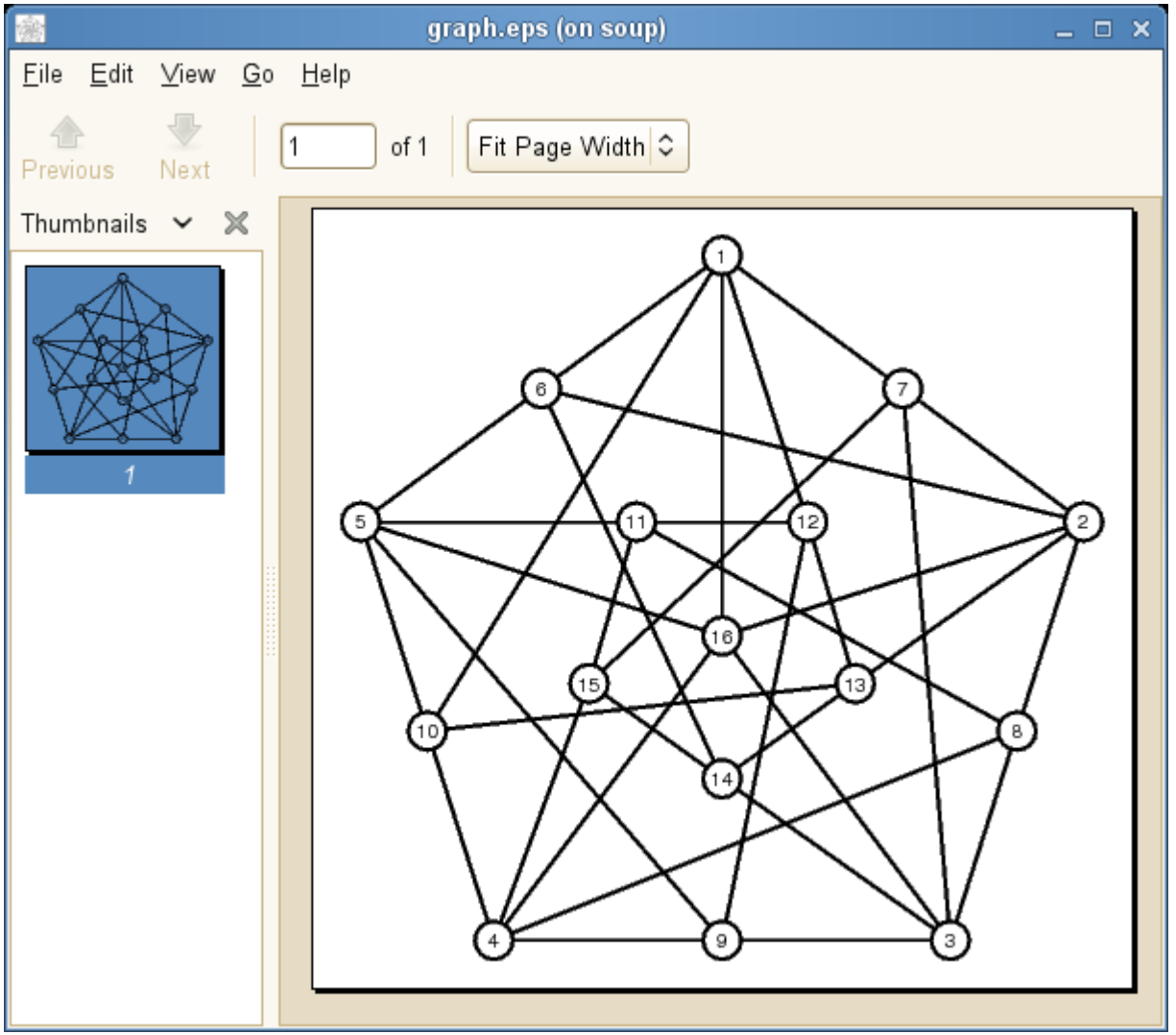
$ **ldd glpksolver**
libglpk.so.0 => /usr/lib/libglpk.so.0

3. New terminal window:

$ **ldd glpksolver**
libglpk.so.0 => /home/rjd4/sw/lib/libglpk.so.0

Viewing the graph

```
$ evince graph.eps
```



UCS

Build time

headers `CPPFLAGS`

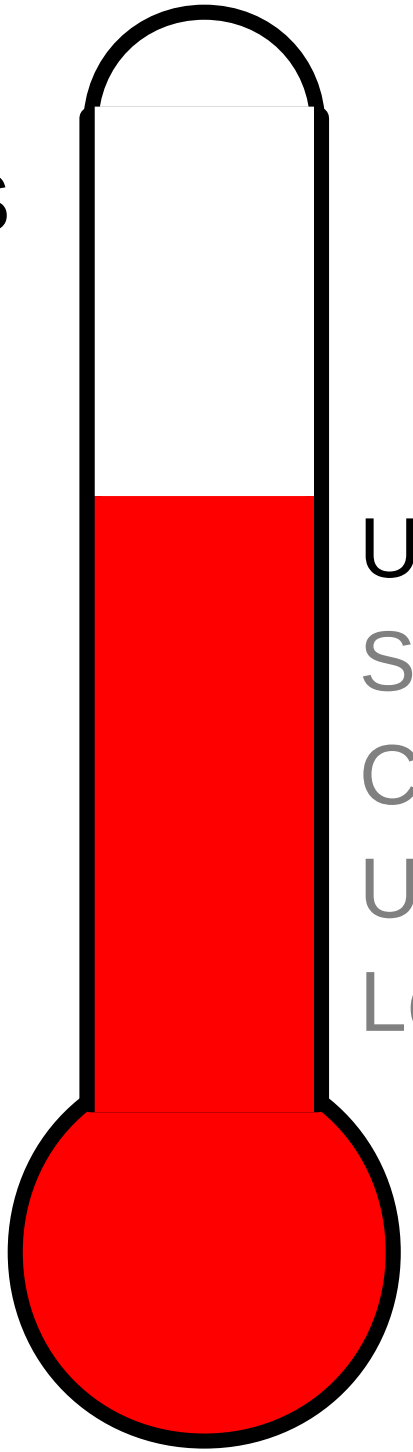
libraries `LDLIBS`

Run time

libraries

`LD_LIBRARY_PATH`

Tools `ldd`



Using libraries

Simple make

Configured builds

Unpacking

Location