Building, installing and running software

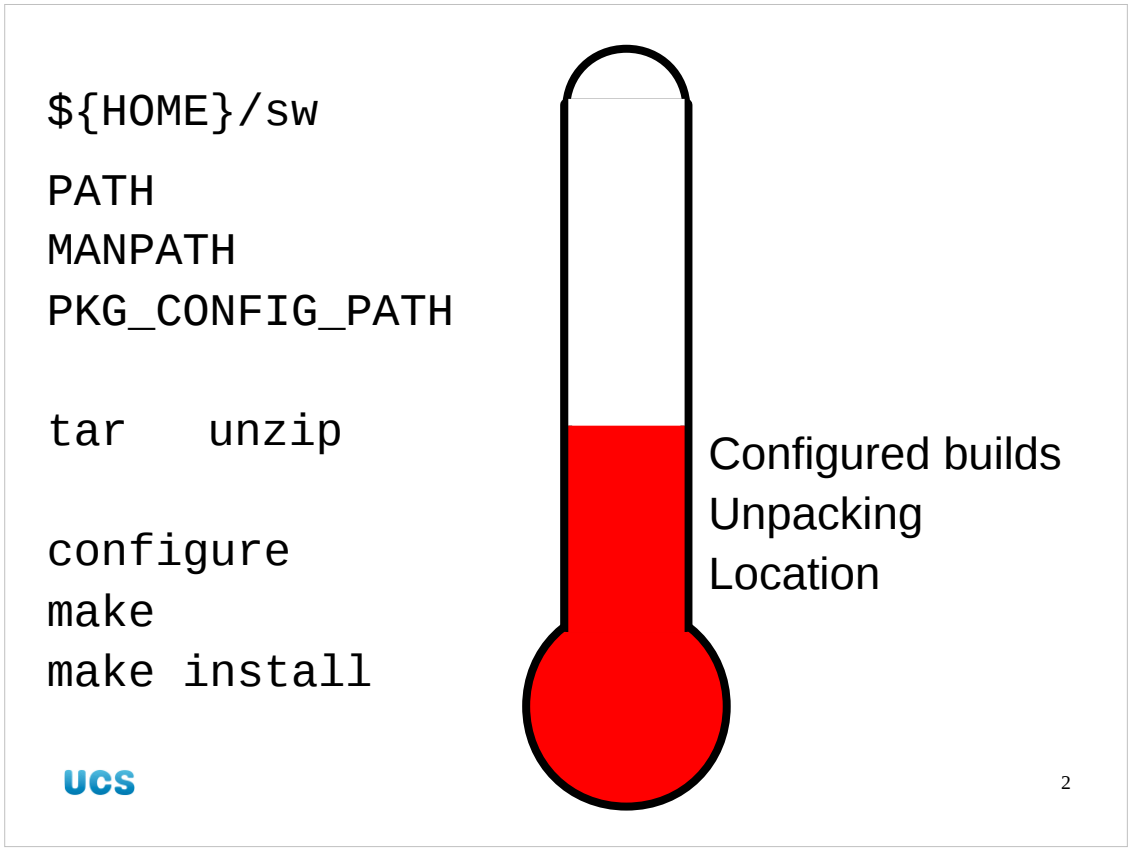
Day two

Bob Dowling
University Computing Service

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

UCS

Welcome back to the second session of the course.



We will start by reviewing where we have got to. In the last session we identified a location for our personal software repository: `${HOME}/sw`. We set a collection of environment variables defining search paths for various components of the system: `PATH` for command location, `MANPATH` for manual pages, `PKG_CONFIG_PATH` for `pkg-config` package data.

For the actual building of software to go in this location we saw how to unpack archive files with either tar or unzip, how to configure the software with the configure script and finally how to build and install the software with `make` and `make install` respectively.

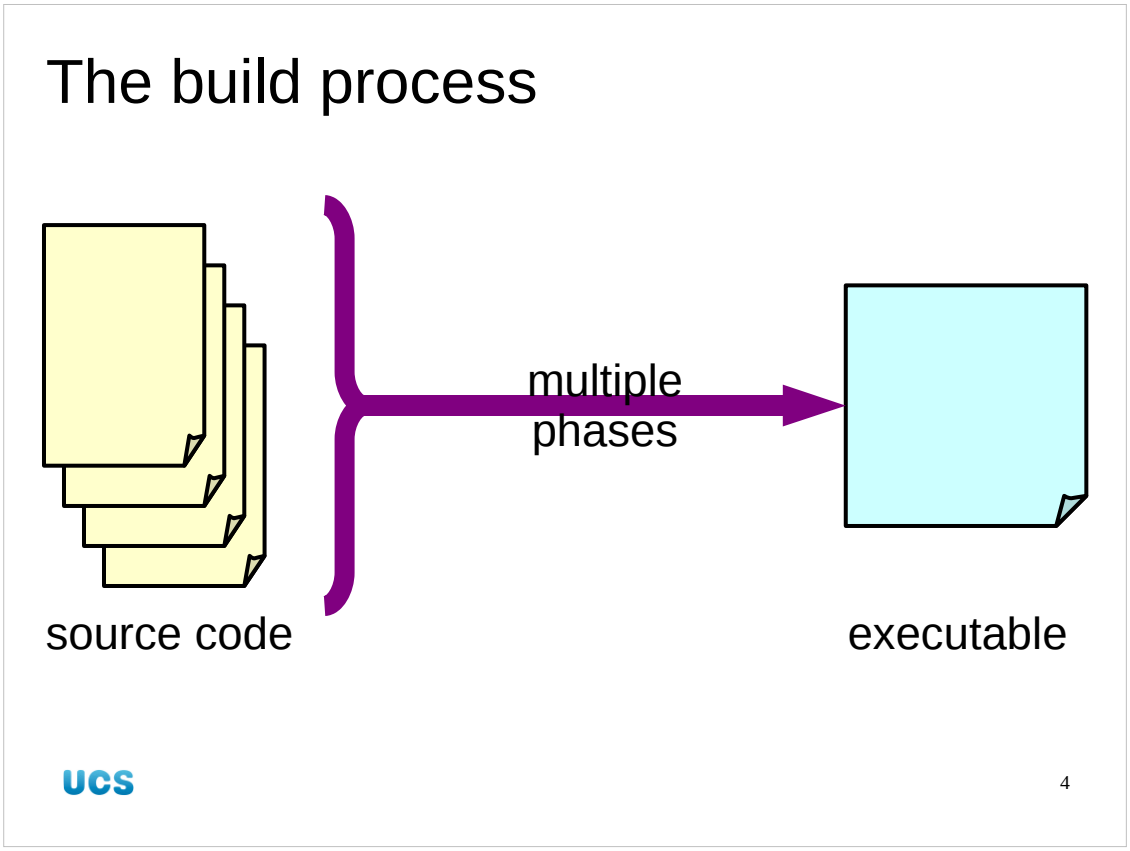
Are there any questions from the first session?

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	

UCS

This afternoon we will look at `make` itself, and how to use it in the absence of any `configure` script to do the work for us. Then we will look at the use of libraries and how to access them from our personal, non-system repositories.



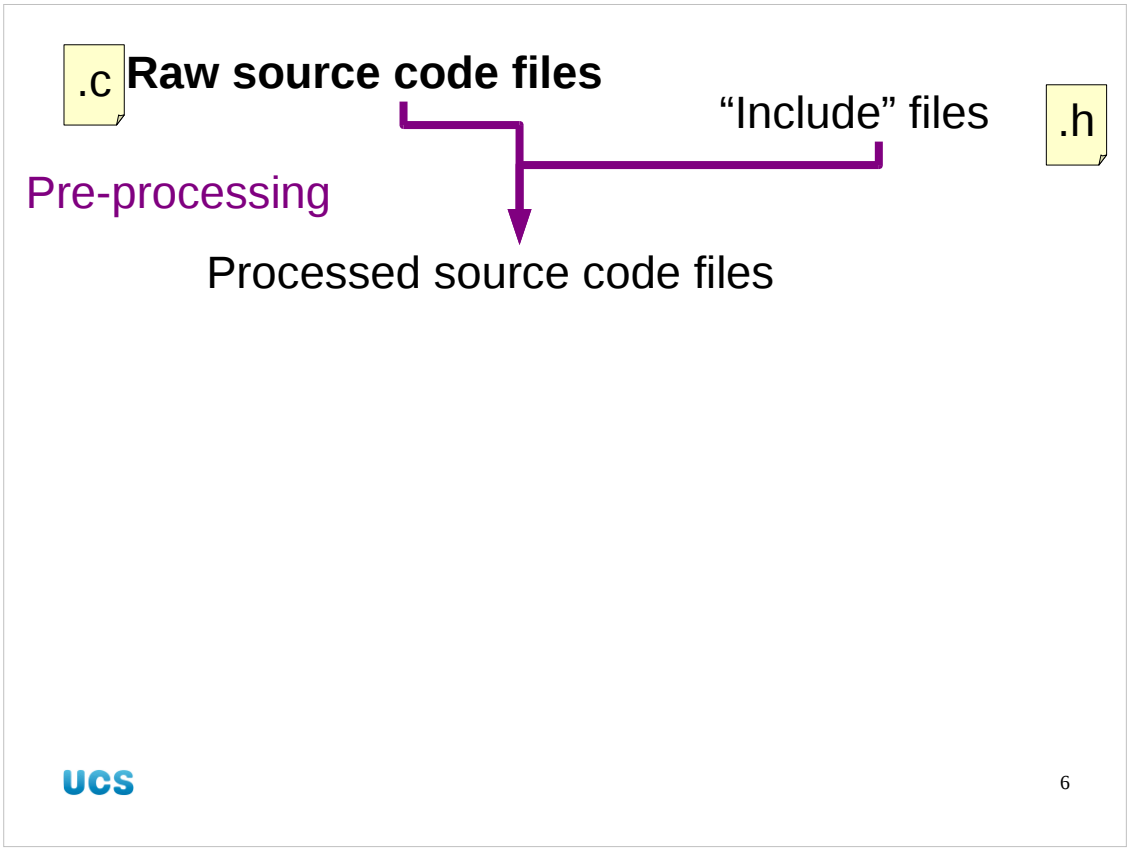
The `make` program automates the build process. To understand `make` we need to understand what the build process is. It consists of multiple phases, each of which has its own command with its own options. These will map on to options in `make`.

In the beginning was the source code.

Actually, that's false. In the beginning were scribbled down notes on a piece of scrap paper, but as far as the computer is concerned we start with source code.

In the slides I'm using C files as examples. The principles apply across all compiled languages, though. Don't get hung up on the detail of which language I'm using.

Also note that I'm talking about multiple source files.



The first thing that happens to each of those source files is that they are textually pre-processed. C, for example, has a set of statements that start with a hash, “#”. These are handled at this stage.

Statements of the form “`#include <stdio.h>`” get replaced with the complete text of the file `/usr/include/stdio.h` (with 840 lines in it). These substituted files have substitutions of their own and so on. Furthermore there is basic variable (“macro”) substitution going on as well and simple switching on and off of blocks of code depending on the values of these variables.

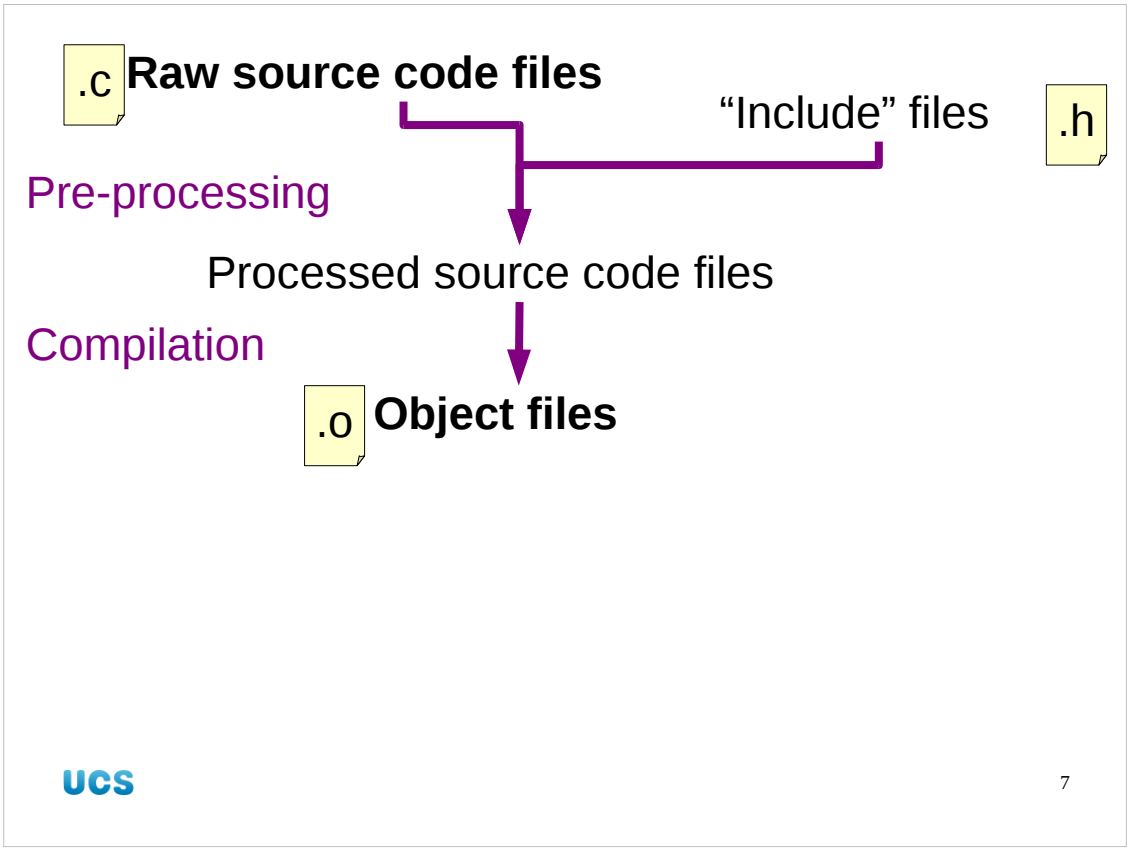
Ultimately, the single line

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

becomes 343 lines of compilable C unless modified by other settings.

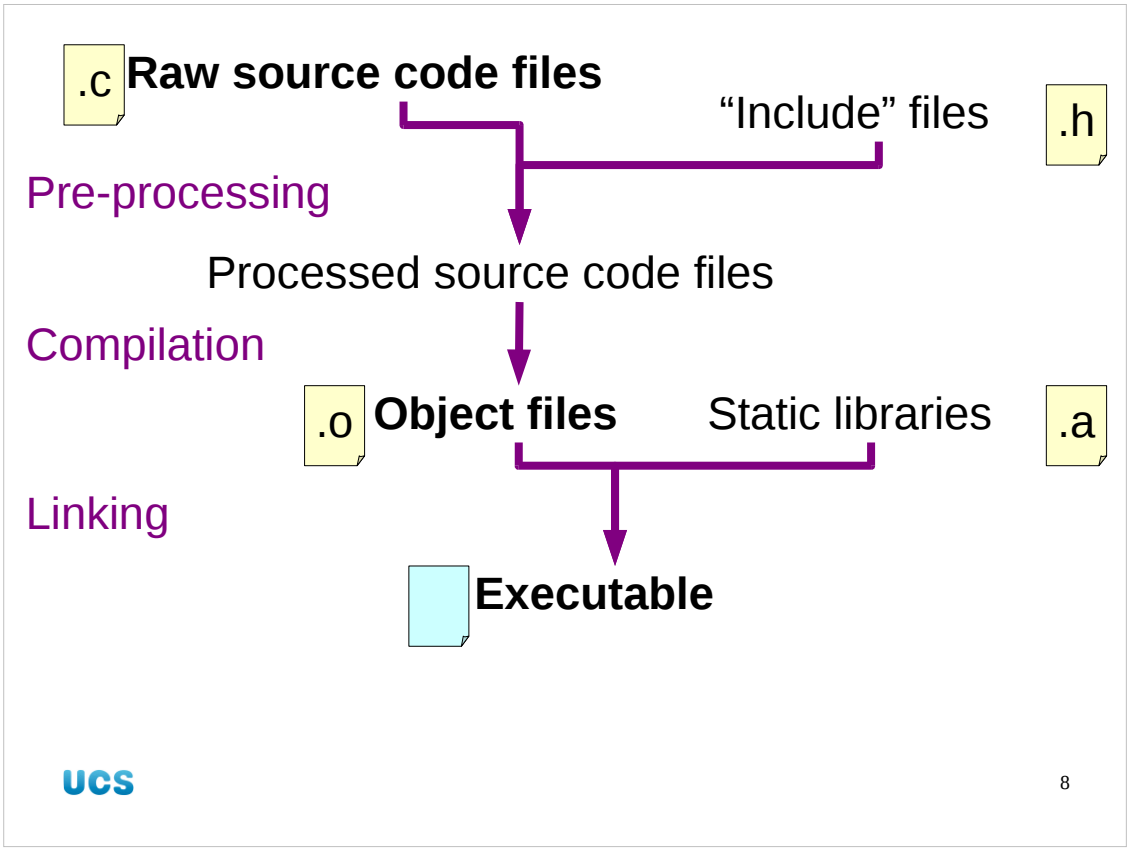
Pre-processing is not the exclusive reserve of C, though. Some Fortran systems use a pre-processor phase too. The naming convention used by `make` is that files whose names end in the uppercase “`.F`” are Fortran that need pre-processing and that files whose names end in the lowercase “`.f`” are pure Fortran that don't need pre-processing.

Pre-processing is normally done entirely internally. The processed source code files don't actually manifest themselves in the file system unless you take special efforts to get them. The `gcc` compiler has a “`-E`” option to only do the pre-processing and to output the processed source code to the standard output.



Next, the source code (possibly re-written by the pre-processor) is compiled. This is the processing that the pre-processor is pre- to.

Compilation consists of taking the individual plain text source files and turning them into machine code for the computer. Each source file, foo.c say, is individually converted into a machine code (or “object code”) file called an object file, foo.o, which implements exactly the same functionality as the source code file. Any function calls in the source code are translated to function calls in the machine code. If the function isn't defined in the source code then it's not defined in the machine code. And so it goes on. This is a pure “translation” process; source code is translated, file for file, into machine code.

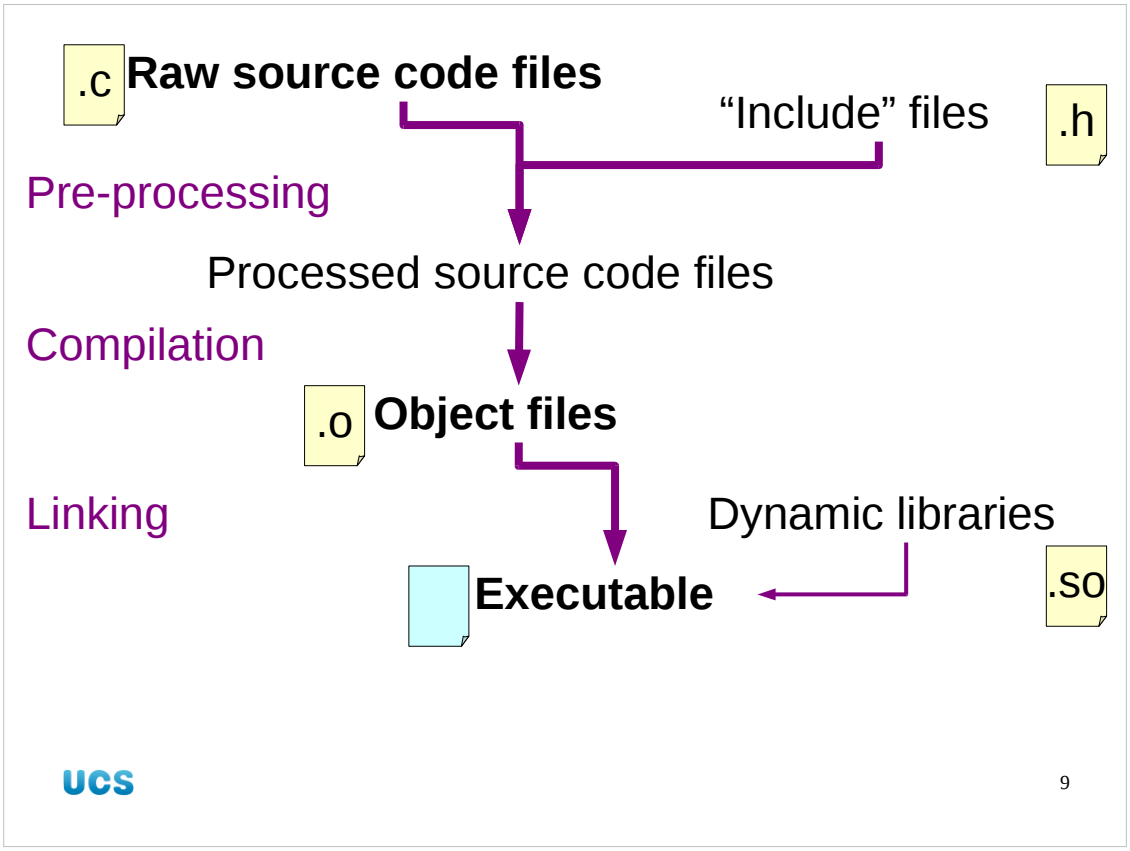


The next stage is called “linking”. This is the combination of the various machine code files into a single executable file. This comes in two flavours (“static” and “dynamic”) which have one critical point in common and the other critical point as a differentiator.

In the linking stage, both forms of linking combine the various object files to create a single executable file. In this stage the machine code that calls a function gets modified so that if another file being linked provides the function then the code now refers to the explicit routine that provides that function. While they were in separate files they couldn't be hooked together like that. Now they are being combined into a single file (literally “linked”) they can be.

The difference between static and dynamic linking comes with the functions that are provided by external libraries.

In static linking, the library is treated simply as a collection of object files for each of the functions it offers. the relevant object files are read from this collection and copied into the executable as if they were provided by local **.**o files that had been built from source code. These library files typically end in “**.**a” which stands for “**a**rchive” because they are essentially archives of object files.



The alternative to static linking is dynamic linking. In terms of how the `.o` files are combined this is identical to static linking. The difference comes from how external references are handled.

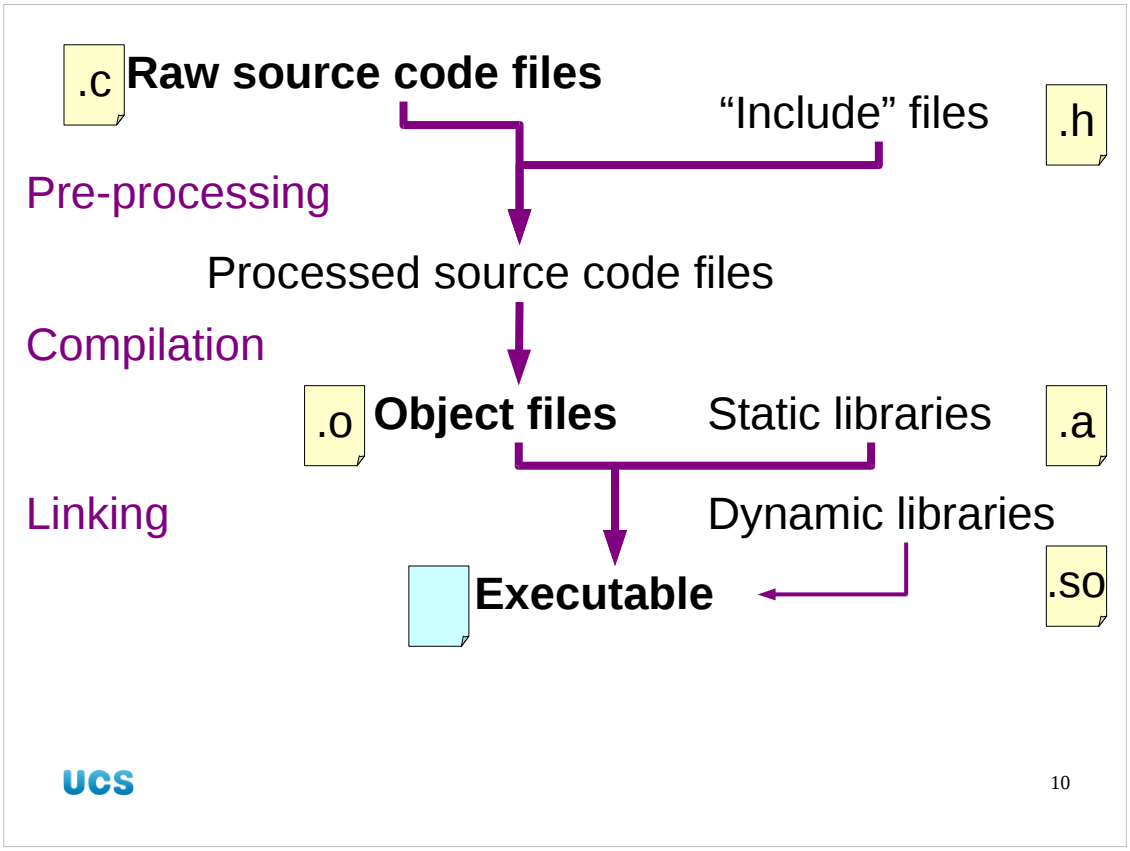
In dynamic linking, rather than copy over the object code from the library file, a note is made of which library it is in (and where within that file), and this note is slipped into the executable. This executable is incomplete and will need help at run time. It is also smaller than a statically linked executable because rather than contain copies of functions it just contains references to them.

So why have dynamic linking? There are three reasons.

The first, rather unimportant, reason is that the executables are smaller.

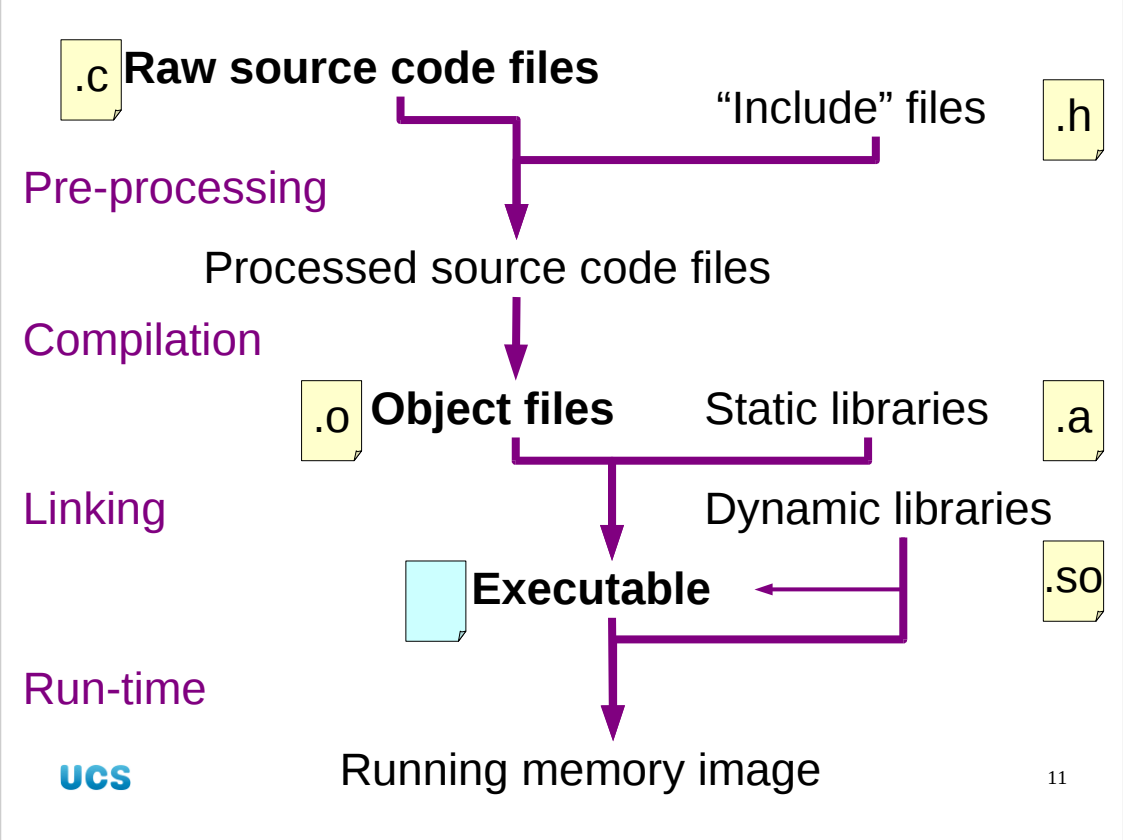
The second is that it makes upgrades easier. If an executable has a reference to an external library then that library can be upgraded and, so long as it keeps the same function interfaces, the programs that use it can take advantage of the new, improved functionality.

We will cover the third reason when we consider what happens at run time. Dynamic libraries have names typically ending in “`.so`”. We will see why then too.



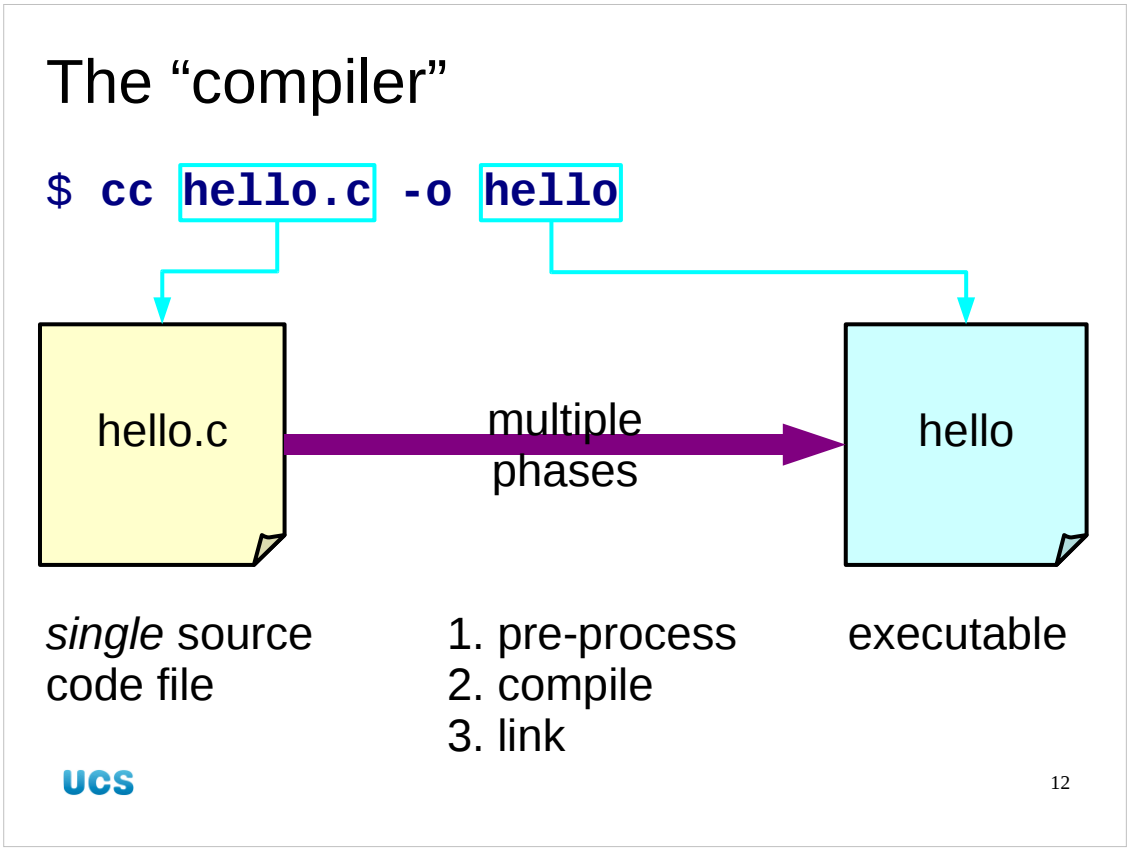
So, by one means or another we have an executable. Now what?

By the way, while we won't use the fact in this course, it is possible to mix static and dynamic linking. Some libraries can be statically linked in and others dynamically.



When a statically linked program is run, the operating system loads the file into memory and simply throws the CPU at it. (It's machine code, after all.)

For a dynamically linked program there's another stage that needs to take place. The references to external libraries need to be resolved. But this is where dynamic libraries come into their own. Only one copy of the library code needs to be loaded in to memory and then all the running programs that use it can use the same copy. For something like the system library (`libc.so`) that every executable needs this leads to a huge saving in memory used and is the third advantage of dynamic linking. This is why the library file names end with a “`.so`” suffix. It stands for “**s**hared **o**bject”.



Typically, we don't get to see all these various phases because they're all hidden behind a single command that implements all parts of the process. This is typically called “the compiler”, but it does much more than just compile in the strict sense of the word.

For example the `cc` C “compiler” can perform all three phases:

Pre-processing:	`cc -E`	only performs this phase
Compilation:	`cc -c`	only performs this phase
Linking:	`cc`	only links if it is given only `.o` files

If we have the source file, `hello.c` say, then the execution of all three phases can be accomplished with a single command:

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

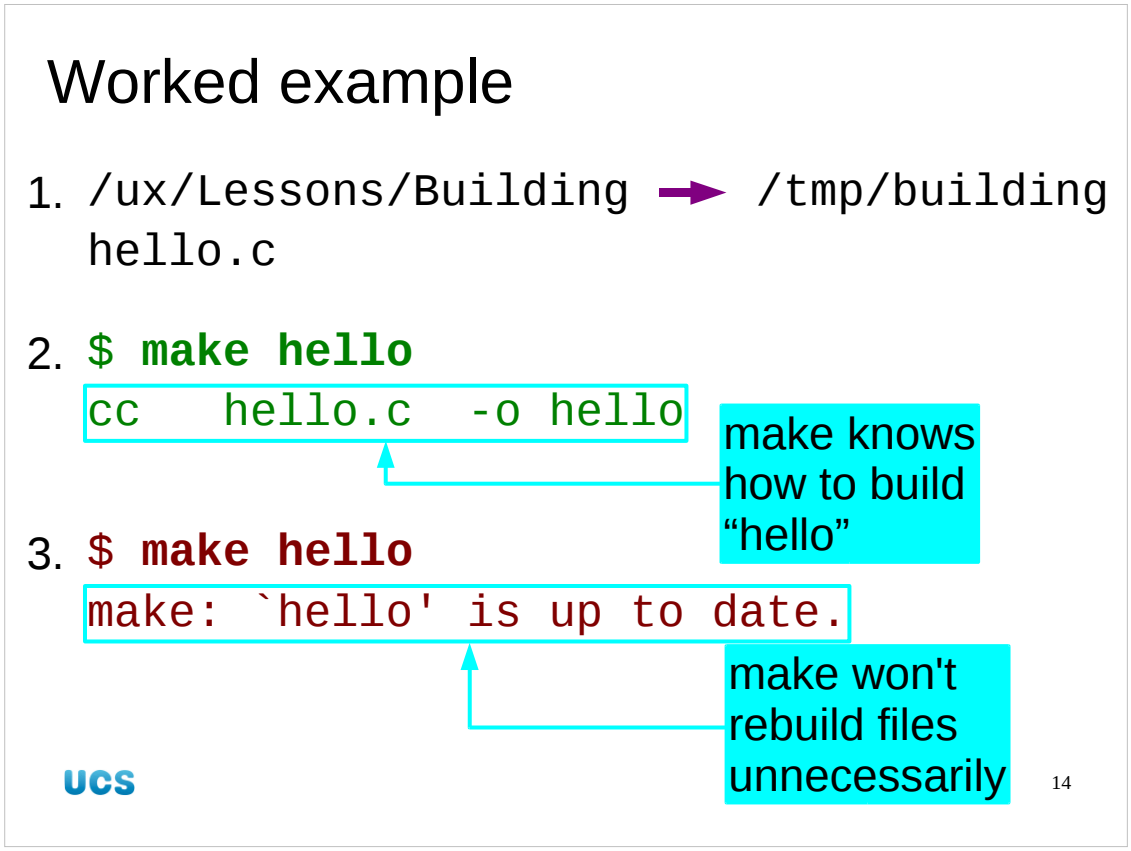
Always automate!

`make`

UCS

Perhaps surprisingly, this course is not about driving compilers on the command line. You will *never* drive `cc` directly in this course!

Instead, we will get `make` to do the work for us. As the number of the options we want to use increases, `make` will help us keep a lid on the total complexity.



So let's see this automation in practice. We will build the "`hello`" executable exactly as we talked about, but we will get `make` to do the work for us.

We will need a single source code file that builds a complete executable. As ever, we have one I prepared earlier in the usual directory. Copy it into your `/tmp/building` directory:

```
$ cp /ux/Lessons/Building/hello.c /tmp/building
```

Once it's there, we will join it and ask `make` to build our executable which we will call "`hello`":

```
$ cd /tmp/building
$ make hello
cc   hello.c  -o hello
```

Note that `make` prints out the commands it is executing on our behalf. Interestingly, and critically for our understanding of `make`, if we run that last command again we get different behaviour:

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

Worked example

4. Update `hello.c`

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

5. `$ make hello`

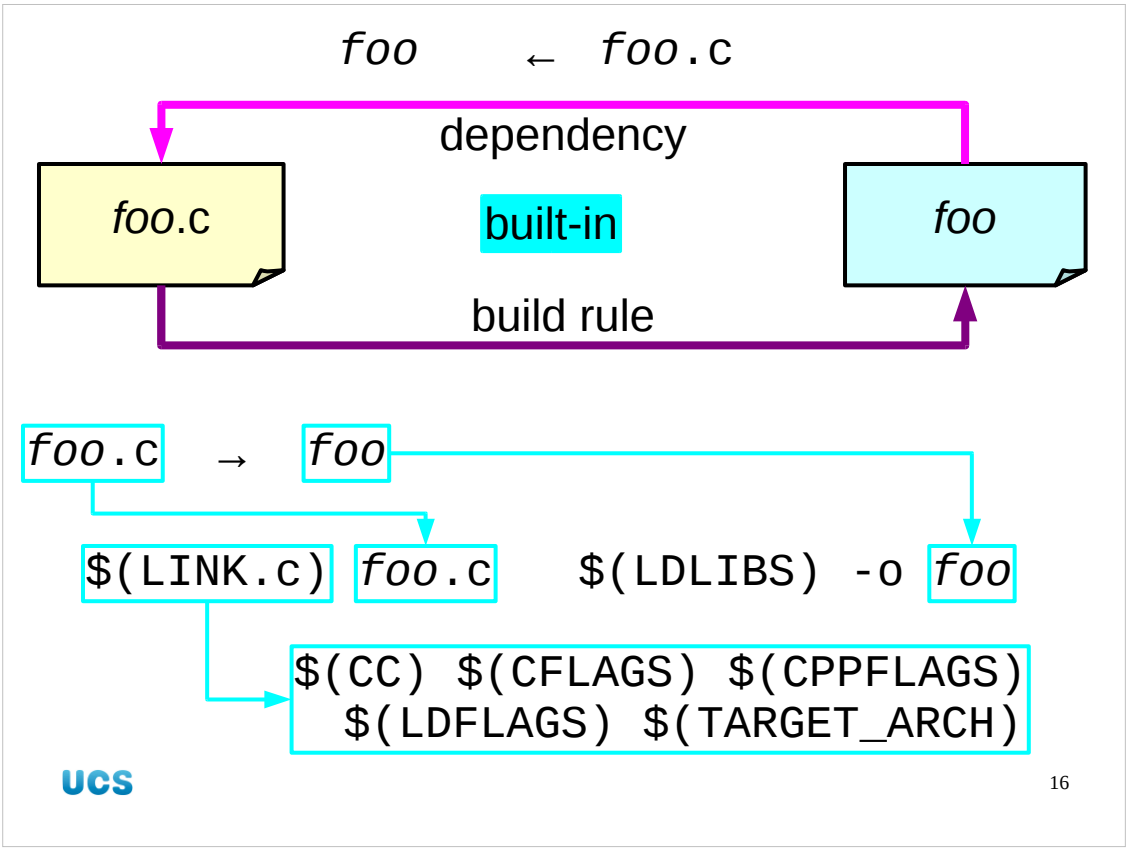
```
cc   hello.c  -o hello
```

make rebuilds the executable this time

UCS

So, if `make` won't rebuild a file that's up to date, let's make that untrue by updating the source file. In the text editor of your choice edit the `hello.c` file to change the string of characters (typically called just a "string" in computing) from "`Hello, world!`" to "`Goodbye, world!`" on line 17 of the file. (PWF Linux comes equipped with (at least) `emacs`, `gedit`, `pico` and `vi`. If you have never used a Linux text editor before we recommend `gedit`.)

Then type "`make hello`" a third time. This time, like the first, it will compile the executable. The first time it did something because the target file did not exist. The third time it did something because the target file existed but was out of date compared to its source file.



There are two halves to what `make` must have built-in. It must have some dependencies that say that if we ask for "`hello`" to be built it should look to see if it has a `hello.c` file. It must also have the instructions for what to do with it.

There is a built-in rule in make that says that a file called "`hello`", or "`foo`" or anything else without a suffix can be built from a file called "`hello.c`" or "foo`.c`" etc. There are also rules for building it from potential Fortran source files "`hello.f`" or "foo`.f`" but make has a priority system and "`.c`" trumps "`.f`".

This rule contains an "action", the command(s) to be obeyed to convert foo`.c` to `thing`. This is written in terms of `make`'s internal variables called "macros". The command for building `foo` out of foo`.c` is written in terms of two macros called `LINK.c` and `LDLIBS`. We will revisit `LDLIBS` later (it defaults to being empty) and look at `LINK.c`.

The action also has slots for the target file and the dependent file to be dropped in, obviously.

The `LINK.c` macro is written in terms of macros itself. Each of these macros has a simple value (often blank). We will visit the meaning of these macros very shortly.

I said that we would not be driving the compiler directly in this course and we will not. What we will be doing is setting these macros so that `make` can drive the compiler, and adding rules of our own to the built-in set.

Common macros — CC

C compiler

Default value — cc

Possible other values:

GNU C compiler — gcc

POSIX C compiler — c99

UCS

We're going to quickly run through the six macros that matter most to us.

CC defines the C compiler and defaults to “cc”. If you want to change this to “gcc” you can but on PWF Linux cc is just a link to gcc. If your system has different C compilers then setting this macro (and we will see how to in a moment) selects the compiler to use.

If you have bought an expensive commercial compiler for your system, this is how you select to use it.

Common macros **CFLAGS**

C compiler options	
Default value	*none*
Possible other values:	
Optimisation level	`-O3`
Debugging level	`-g3`

There are two sets of flags typically passed to the C compiler corresponding to the two phases of the compilation of a C program. You do not need to understand the complexities of C compilation in detail but a rough idea is useful for understanding `make`'s macros.

`CFLAGS` sets the options for the actual compilation of the (post-processed) C code to machine code. This can be used to set optimization and debugging options.

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
```

UCS

CPPFLAGS sets options on the “pre-processor” phase. A C program contains a number of pre-processor directives. These allow for the textual inclusion of other files (so-called “include files”), the setting of constant values (“C pre-processor macros”), switching on or off blocks of code depending on the values of these macros.

The C pre-processor is very powerful and is actually used by other languages too. The CPPFLAGS make macro contains these instructions.

Especially in C-only contexts the distinction between the pre-processor flags and the compilation flags is often lost and CFLAGS is used for both. The `pkg-config` command has an option `--cflags-only-I` to give just the pre-processor options (for CPPFLAGS) and a `--cflags-only-other` option to give the compilation options (for CFLAGS). However, if the two variables are being combined then typically CFLAGS is used for both and the `pkg-config` option `--cflags` gives both combined.

Common macros **LDFLAGS**

Linking options	
Default value	*none*
Possible other values:	
Require static linking	`-static`

`LDFLAGS` is the macro that sets all the options on how the linker is to do its job. It can help strip out unnecessary elements in the machine code or, as shown in the slide's example, do its linking statically.

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
```

UCS

When linking in external libraries, the linker needs to know two things: what libraries to load and where the libraries are.

LDLIBS is the macro that defines the extra libraries and where they might be found if not in the standard locations. It is this macro that `pkg-config` can give values for with its `--libs` option.

We will return to the issue of tracking down libraries in a few slides' time.

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
```

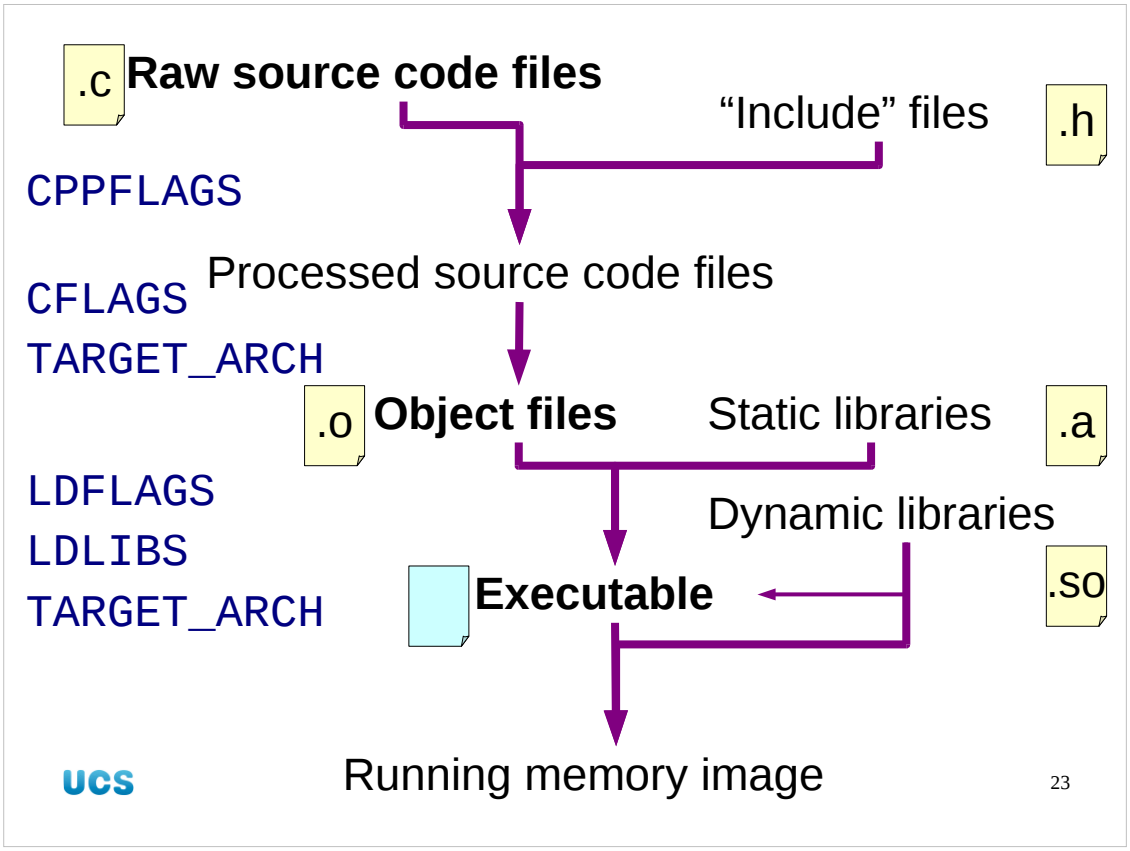
UCS

The last macro to look at directs the build system as to what architecture to build for. The reason we mention it is that the default behaviour (if `TARGET_ARCH` is unset) is not what you might expect.

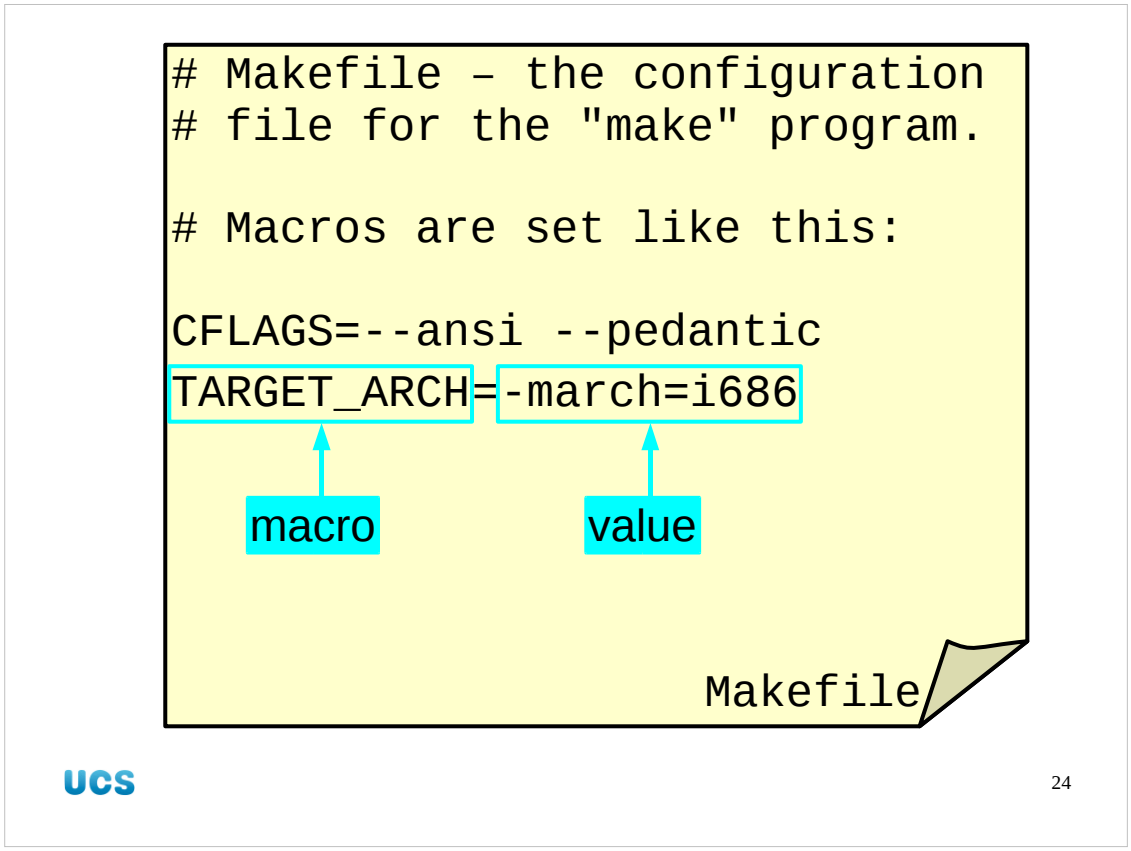
On an Intel or AMD GNU/Linux system the compiler defaults to producing generic machine code that will run on any “x86” architecture (from the really old 386 chips up to the modern AMD64 chips). The compilers have options to say “build for this specific architecture, exploiting its extra features” and the `TARGET_ARCH` macro is where they go. The exact option depends on the compiler and the platform. For `gcc` on an x86 system the machine architecture option is “`-march`” (“**m**achine-specific option: **arch**itecture”) and it takes the value of the architecture to be built specifically for. You can get your system's architecture with the `arch` command.

Because the architecture can influence both the compilation and linking phases this macro gets used in two different places. This is why we don't just roll its functionality into `CFLAGS`.

Warning: Code compiled with this option is not portable (in its compiled state) between machines of different architectures. Check with the `arch` command if you are using unknown kit.



Finally, here they all are in place in our diagram of the build process. Note that TARGET_ARCH appears twice.



So we have these macros. How do we get at them? How do we set them?

The standard way is to create a file called “`Makefile`” and to put the definitions in them as a series of lines

MACRO=VALUE

Comments can be added to the file starting with the “#” (“hash”) sign.

For example, if we wanted the C compiler to have some options (`CFLAGS` macro) to demand ANSI standard C (`--ansi` option) in its source code and to treat any errors as fatal (`--pedantic` option) then we add the line

```
CFLAGS=--ansi --pedantic
```

to the `Makefile`.

Worked example

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

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

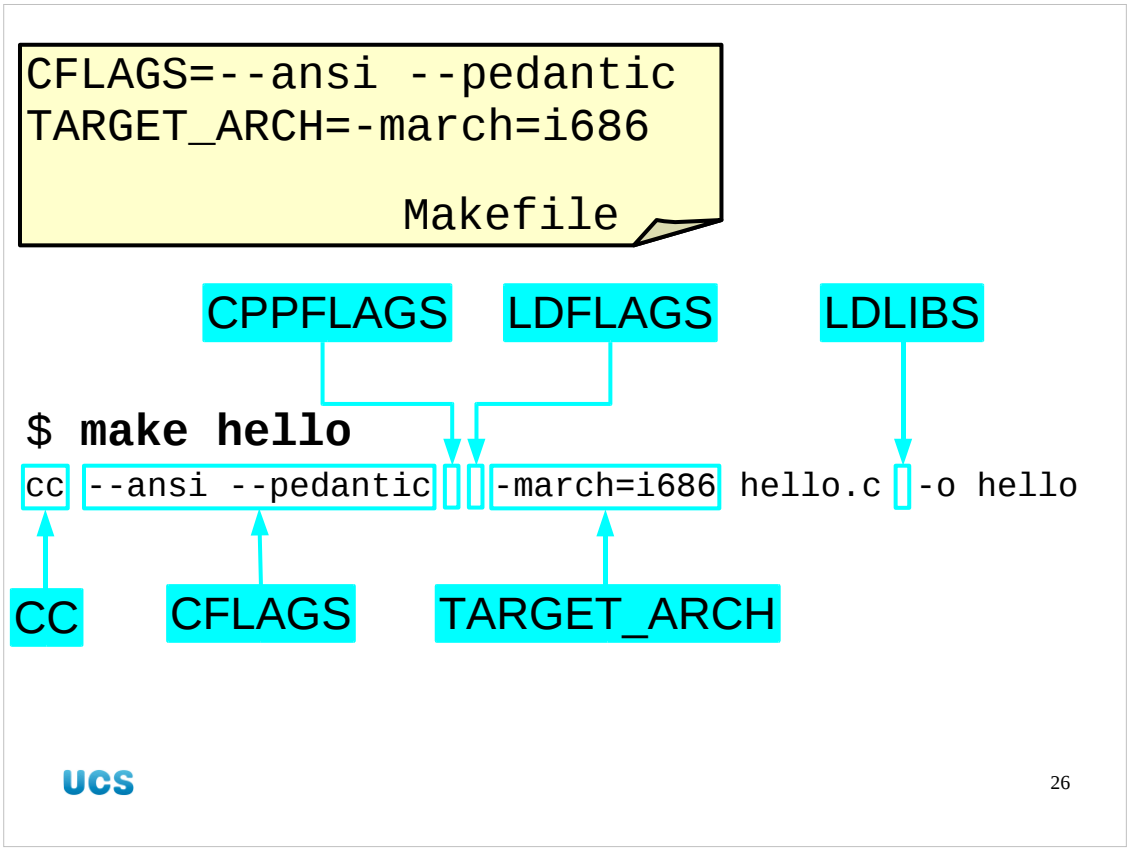
UCS

So let's see this in practice.

Make sure you are in the `/tmp/building` directory and, using your favourite text editor (`emacs`, `gedit`, `pico`, `vi`, etc.) create a file called `Makefile` (with a capital "M") containing the two lines

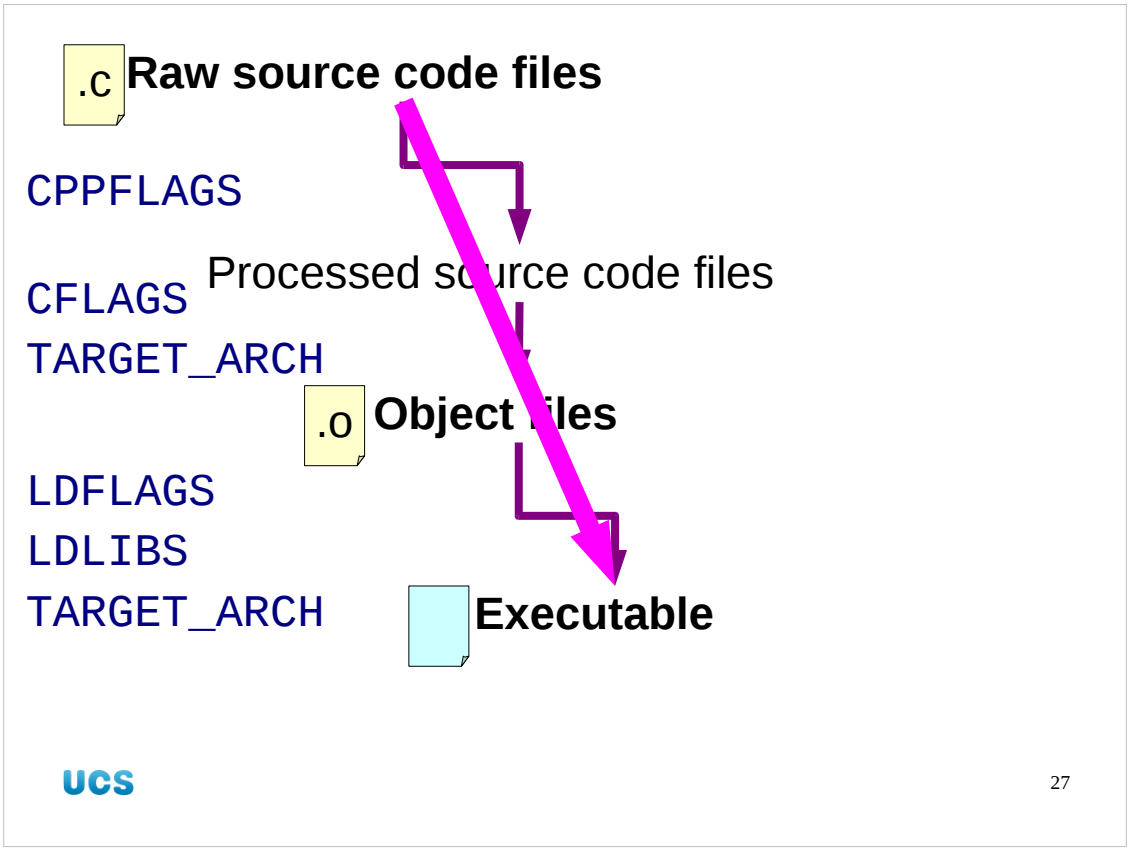
```
CFLAGS=--ansi --pedantic
TARGET_ARCH=-march=i686
```

Next, remove the old "`hello`" program. The make program looks to see if programs are out of date with respect to the files they are built from, but not with respect to the `Makefile` itself.



In our worked example, we see the values of the macros defined in the `Makefile` picked up by `make` and applied to the command it runs. Note that by using `make` we don't have to remember what order the compiler options come in. As the lists of options grow longer and more complex this becomes increasingly useful.

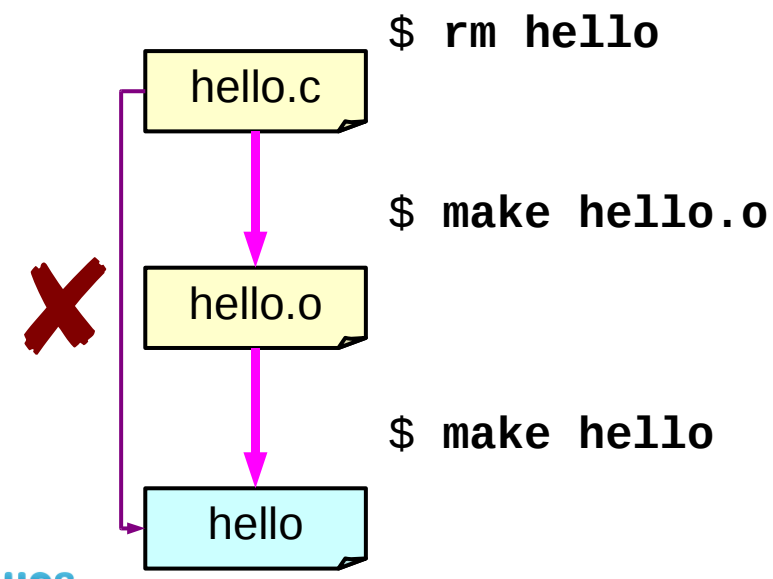
Note that `CC` has taken its “`cc`” default value, and that `CPPFLAGS`, `LDFLAGS` and `LDLIBS` have all taken their empty default values.



It's also worth noting that `make` has used `cc`'s “all in one” nature to leapfrog over an intermediate stage. The object file never appears as part of the build process. In fact it is hidden away in `/tmp`.

To understand `make` better, and to give us a place to move forwards from, we will split this into its two constituent phases.

One step at a time



UCS

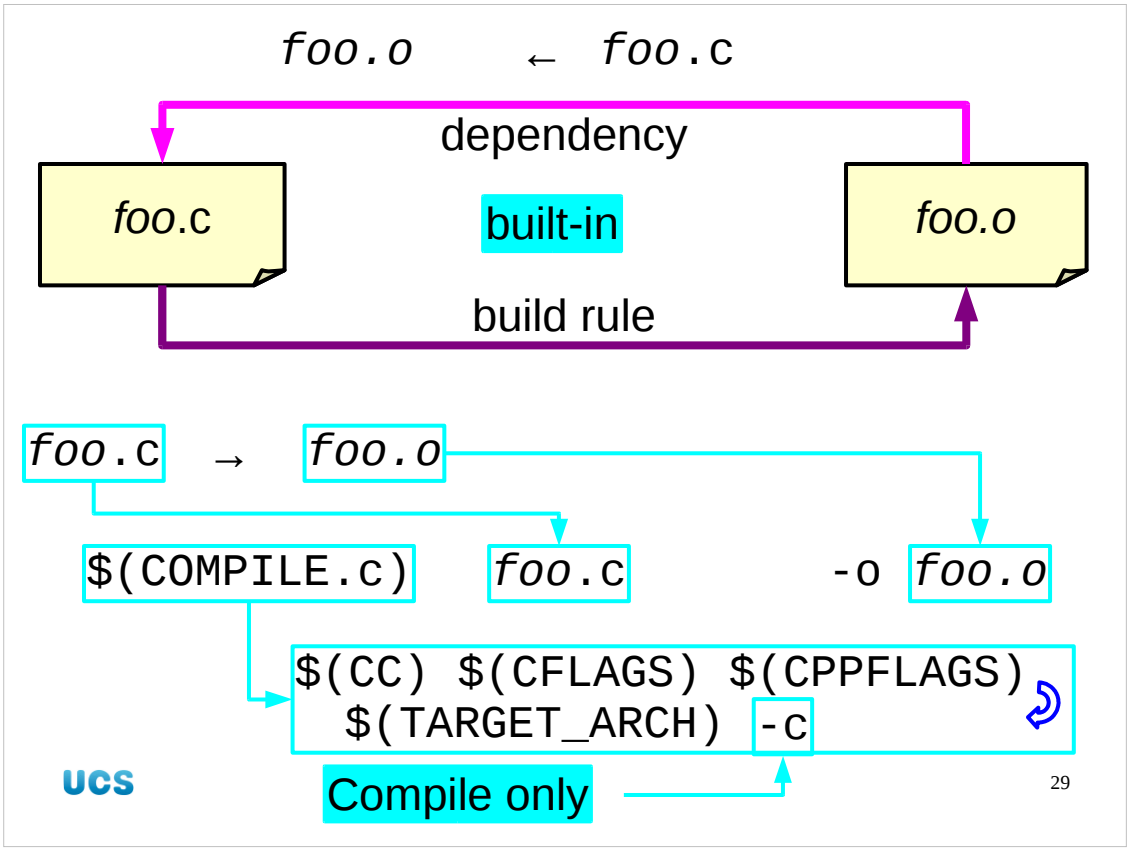
To this end we will remove our `hello` executable and rebuild it in two stages.

First we make `hello.o` explicitly. This is the object file we skipped previously.

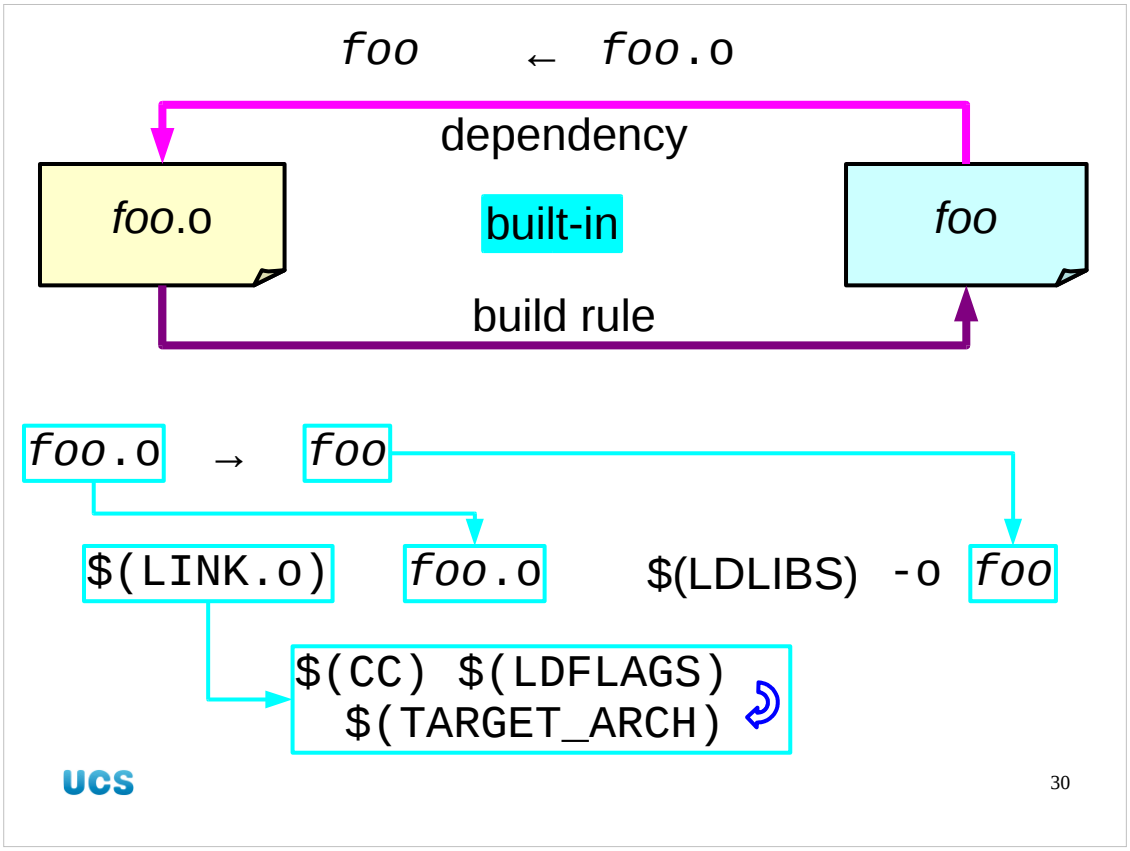
Second we make `hello` itself. Note that at this point `make` has a choice. It can either link `hello.o` or build `hello` from scratch from the source file `hello.c`.

`make` builds the executable from the object file because that is “nearer to complete” than the source file.

This reveals to us how `make` links a single object file into an executable. We will use this information when we want to link several object files rather than just one.



Just as there was a rule to build executables from "`.c`" files, there is a rule to build "`.o`" files too. Instead of the `LINK.c` macro it is defined in terms of a similar macro called `COMPILE.c`. This macro is, in turn, built up from some of the same base macros as we have already seen: `CC`, `CFLAGS`, `CPPFLAGS` and `TARGET_ARCH`. It also uses the "`-c`" option on the compiler to direct it to only compile and not to try to link.



Then we built the executable from the object file. Again there is a rule. Note that this rule is written in terms of a macro called `LINK.o`, which takes "`.o`" files to executables, just as `LINK.c` did with "`.c`" files. Again, it is written in terms of the base macros we have already met.

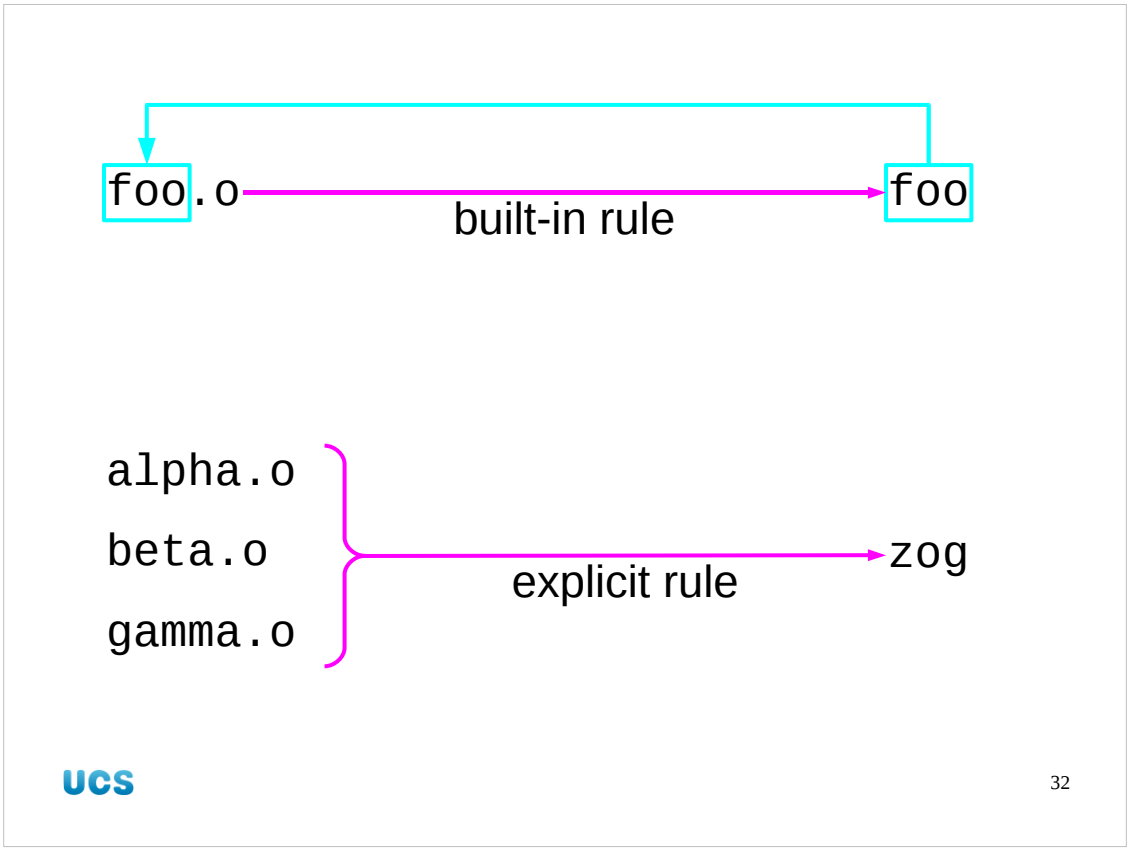
make coffee

Dependencies:
1. 45p
2. coffee machine

Ten minutes break

UCS

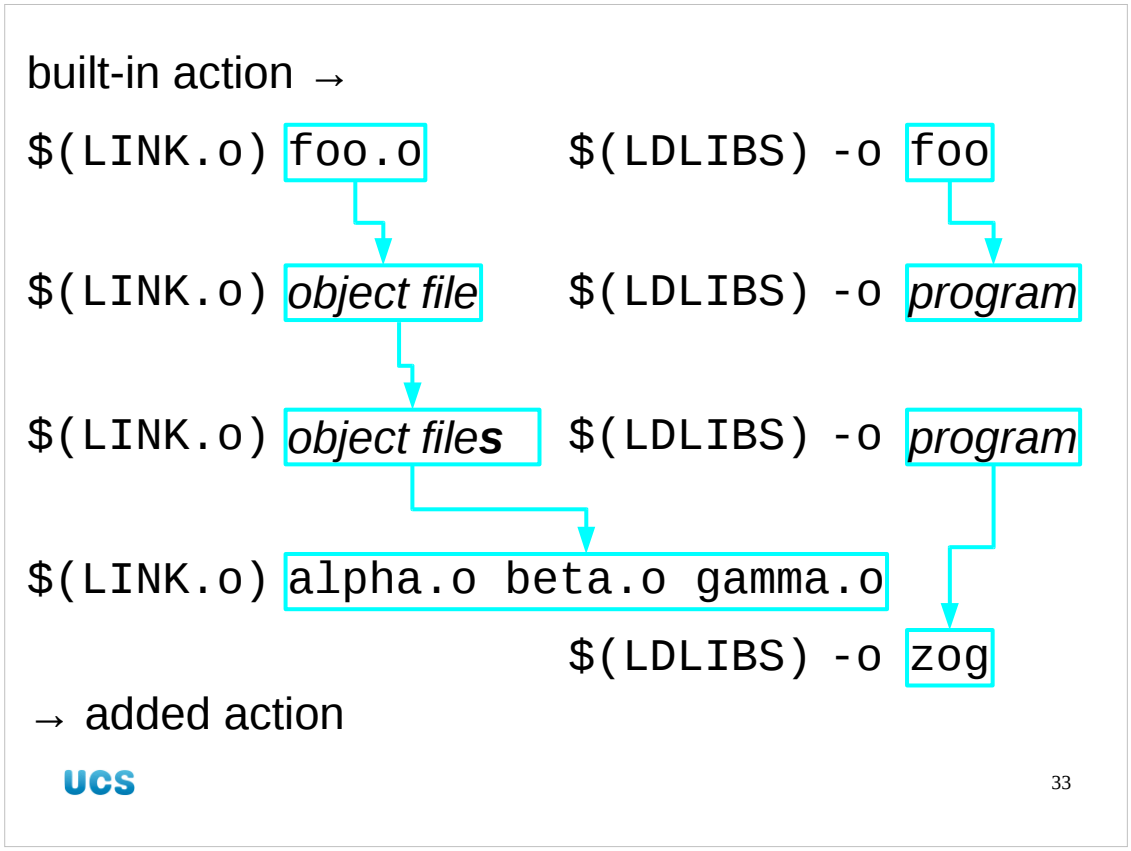
And that finishes us with the simple built-in rules. Let's have a coffee break.



Built-in rules are only possible where `make` can deduce the name of the dependent file (`foo.o`, `foo.c`, `foo.f`) from the name of the target executable (`foo`). This is easy where single files are concerned and the file names track the obvious conventions.

Life is rarely that easy. The typical executable is built of multiple object files. There is no way that `make` can determine on its own that the `zog` executable is built from three object files, `alpha.o`, `beta.o` and `gamma.o`.

We will need to add an explicit rule to `make`. As with the macros, we do this with the `Makefile`.



So what command do we need to run?

We will take this in stages. We start by looking at the built-in action for linking a single object file (`foo.o`) to create an executable program (`foo`).

```
$(LINK.o) foo.o $(LDLIBS) -o foo
```

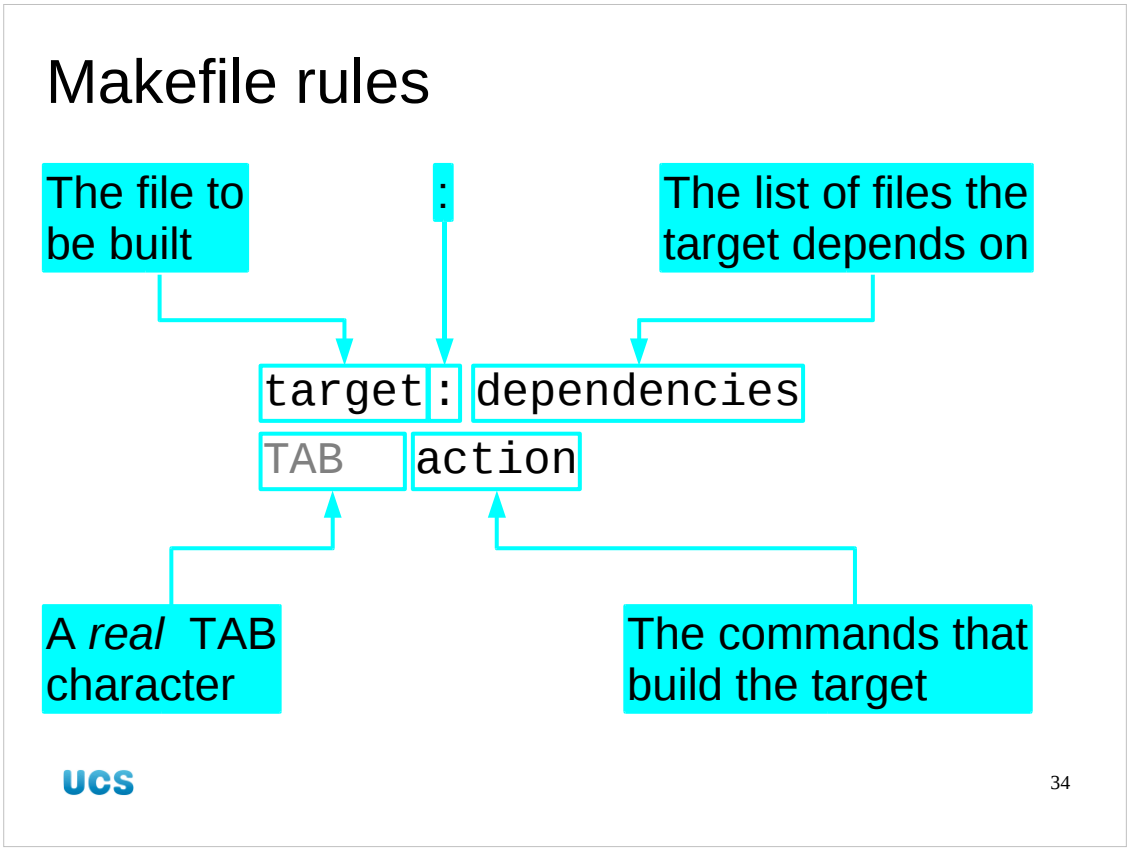
`foo.o` is the object file and `foo` is the program we are building.

There is an obvious generation which is simply to replace the single object file with the multiple object files required in the more common general case.

In our case, where the executable program `zog` is built from three object files `alpha.o`, `beta.o`, and `gamma.o` this gives us the action

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

Now all we have to do is to see how to add this action to a `Makefile`.



The structure of a rule in a Makefile is simply described but easy to get wrong.

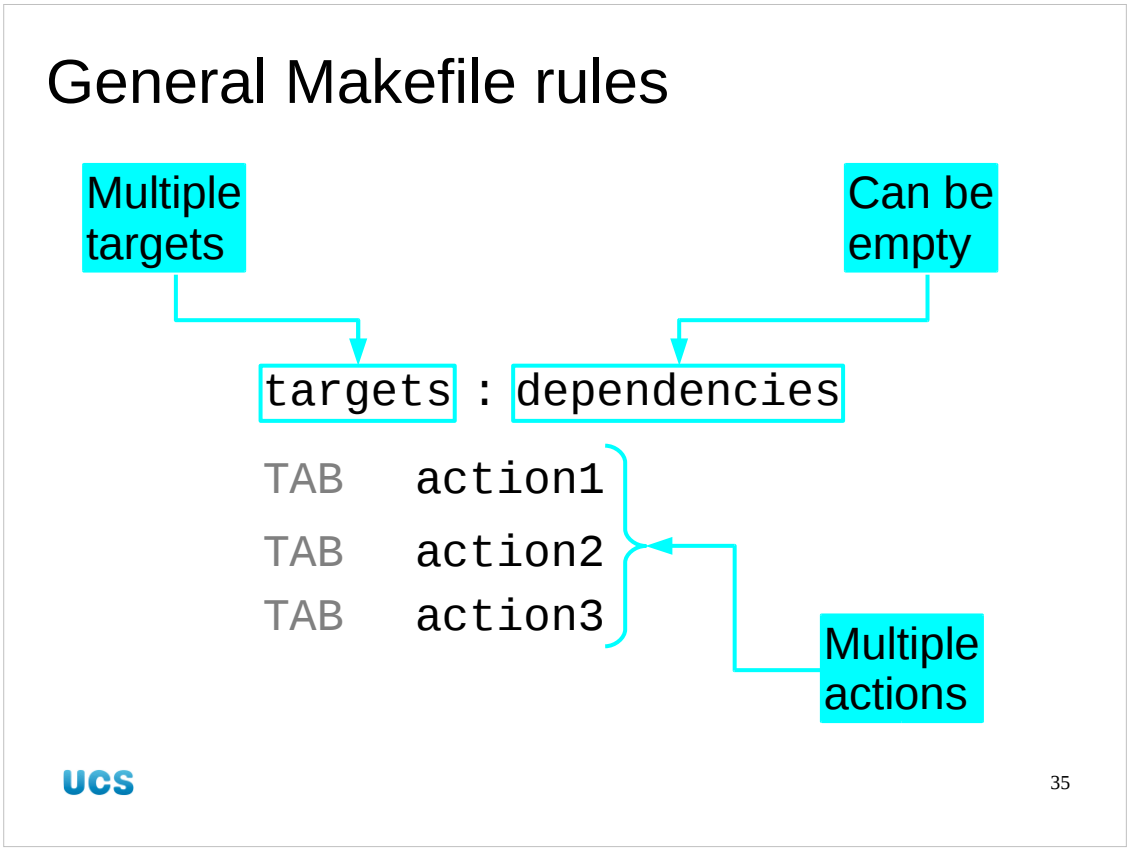
First we specify the target, the file that is going to be built. This is the thing that can follow the `make` command as an argument so adding a target “`zog`” allows us to say “`make zog`” without relying on built-in rules.

Next comes a colon. You can have white space either side of the colon, it doesn't matter and conventions vary. Pick something you like and be consistent.

After that comes the list of dependencies. If the target file, `zog`, does not exist it will always be built when you run “`make zog`”. However, if it does exist then the dependencies are the set of files it is compared against to see if it is up to date. If the target file is more recently created or updated than all of the dependencies then it is not remade. Otherwise it will be recreated.

The next line defines what the action is to do the build. It starts with a literal TAB character. That's the character you get by typing the ⇥ key on the keyboard. It may appear as a number of spaces, typically four or eight, but it is not the same thing. It must be a real TAB character. Note that copying from a terminal window and pasting can sometimes convert a TAB into the number of spaces it appears as. Be careful.

After the TAB comes the action. This is the command we established in the previous slide, typically defined in terms of standard macros, the dependency files, and the target file. The command must take the files listed in the dependencies and generate the target file. The action must *not* modify or delete the dependency files.



A rule can be slightly more general than that.

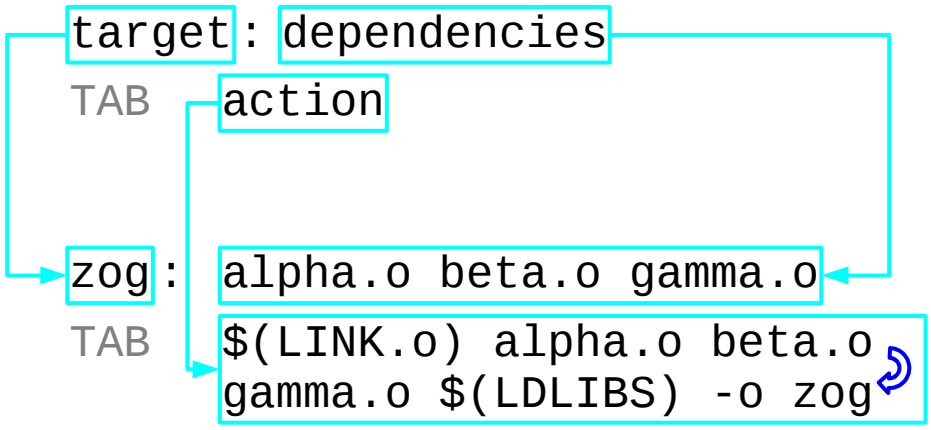
An action may generate more than one file. Multiple targets can be listed before the colon if that is the case.

The list of dependencies can be empty. If a target has no dependencies then it is treated as being always out of date so running `make` for that target will always run the rule's action.

There can be multiple actions. This is rarely a good idea and comes with a set of warnings. Each action is run independently. (They are each run in independent sub-shells for those who speak Unix.) If the first action changes directory, for example, the second action won't reflect that.

The typical reason to have multiple actions is to create one or more intermediate files and then to convert them into the ultimate target in a subsequent actions. Don't do this. Instead split the rule into two or more rules, explicitly creating the intermediate files as targets that `make` recognises as such and set a dependency on them in the second rule.

Our rule



UCS

So our rule can be built given the information we have.

The target is the single file “`zog`”.

The dependencies list is the set of three object files, `alpha.o`, `beta.o`, `gamma.o`.

The action is the linking command we determined in an earlier slide, still written in terms of the macros `LINK.o` and `LDLIBS`.

The 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
```

TAB

Line wrap

UCS

We've skipped over the issue of overly long lines in `Makefiles`. A line that ends with a backslash character (“\”) which has no spaces following it will have the next line joined to it.

Now let's address the “TAB question”. How did `make` get to be like this?

This requirement for TABs rather than just “leading white space” harks back to the creation of the first make in 1977 by Stuart Feldman of Bell Labs. The (potentially apocryphal) story is this: Feldman knocked together a quick version of `make` which used single TABs, some single character macros and a few other tricks to let him write a prototype quickly. He let some people use it and went home for the night. It proved so popular that when he returned to work the following morning it was in use in so many projects that any change of specification was vetoed by his colleagues.

Worked example

1. `$ cd /ux/Lessons/Building`
2. `$ cp alpha.c beta.c gamma.c /tmp/building`
3. `$ cd /tmp/building`

UCS

So let's put this in practice. We are going to build `zog`. First we copy over the source files into `/tmp/building` and change back to that directory.

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`

UCS

Next we will update our existing `Makefile` to add the new rule we have created.

Finally we will make the `zog` executable.

```
$ make zog
cc --ansi --pedantic  -march=i686 -c -o alpha.o alpha.c
cc --ansi --pedantic  -march=i686 -c -o beta.o beta.c
cc --ansi --pedantic  -march=i686 -c -o gamma.o gamma.c
cc  -march=i686 alpha.o beta.o gamma.o  -o zog
```

Note that we have not added any rules to create the object files from the source files. We use the built-in rules wherever possible. We depend on the object files and `make` works out what to do to get them.

Improving our rule

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

UCS

Now we have a rule that works we will improve it.

The problem with our current rule is subtle. In this very simple case it's not really a problem at all, but we will improve it nonetheless so that as cases become more complex we are already prepared for them.

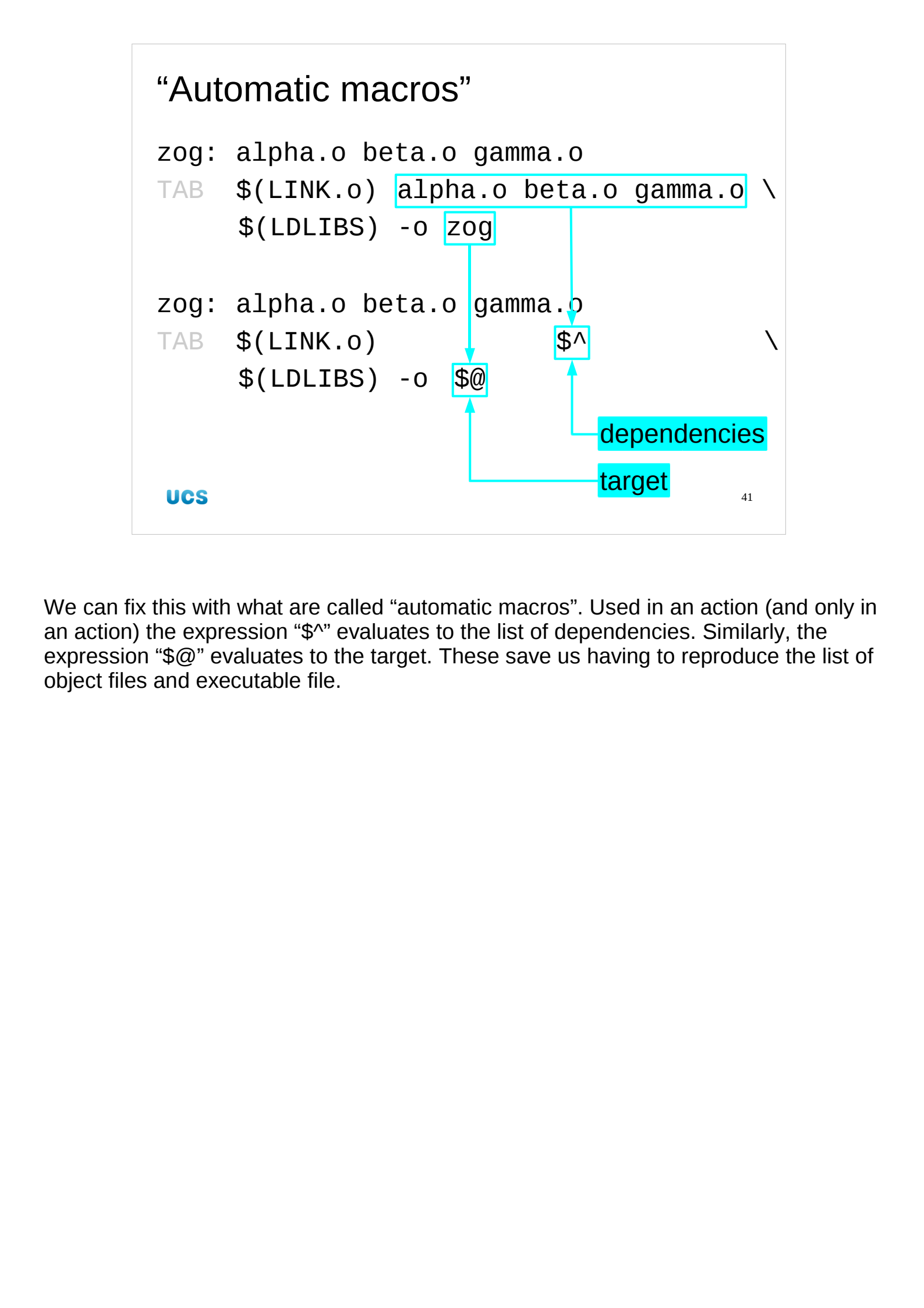
The problem is that there is repetition of data. The file name “zog” appears both in the target and the action. Similarly the list of object files appears both as the dependency list and in the action.

Why is repetition bad? A common error in software work is to modify one copy of some item (say adding delta.o to the list of object files) and forgetting to do it somewhere else (in the action, for example). With just three files this is hardly an issue as it is immediately visually obvious whether the two lists match. Now suppose you have over 1,200 files. That's not an idle example; we will be meeting it in tomorrow's session.

Generally speaking we want things defined once and only once.

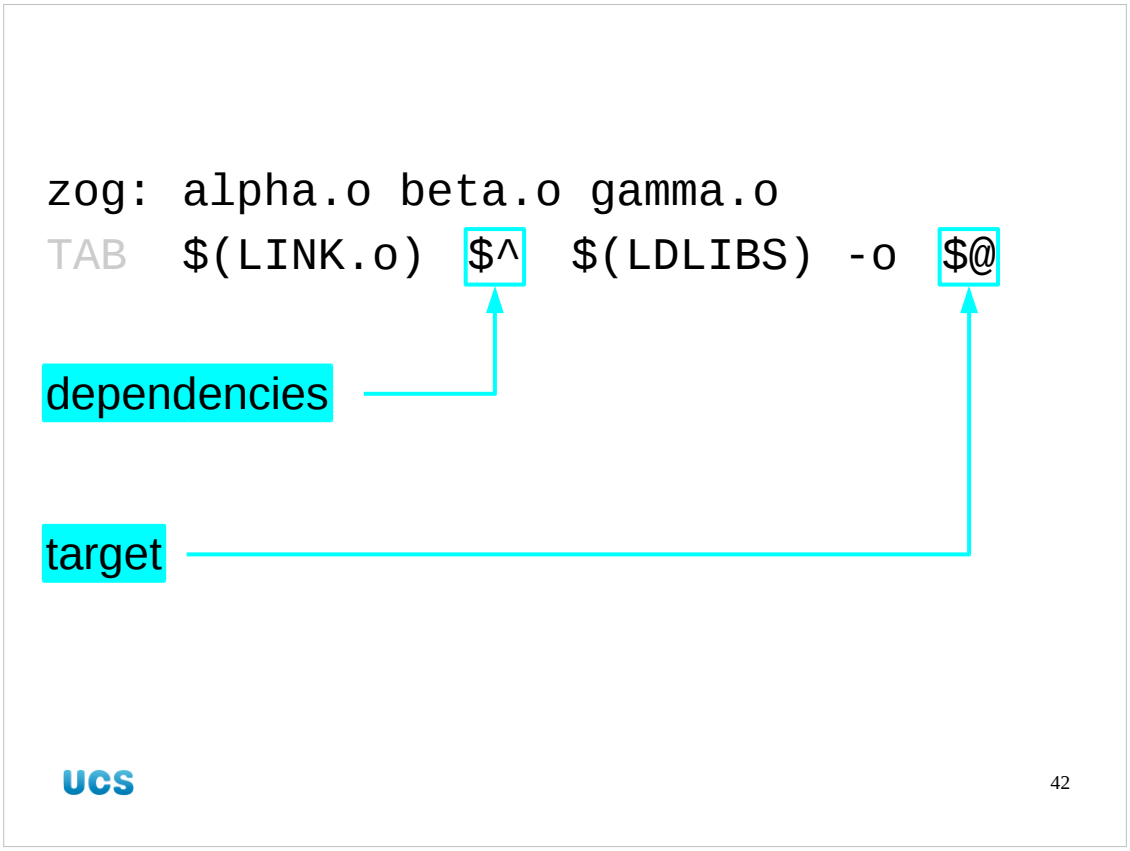
“Automatic macros”

```
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  $@
```



UCS

We can fix this with what are called “automatic macros”. Used in an action (and only in an action) the expression “$^” evaluates to the list of dependencies. Similarly, the expression “$@” evaluates to the target. These save us having to reproduce the list of object files and executable file.



The automatic macros also allow us to make our action lines shorter, which is always to be welcomed. Again, if the list of dependencies is very long this is a great relief.

Values defined rule-by-rule

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

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

alpha.o beta.o …

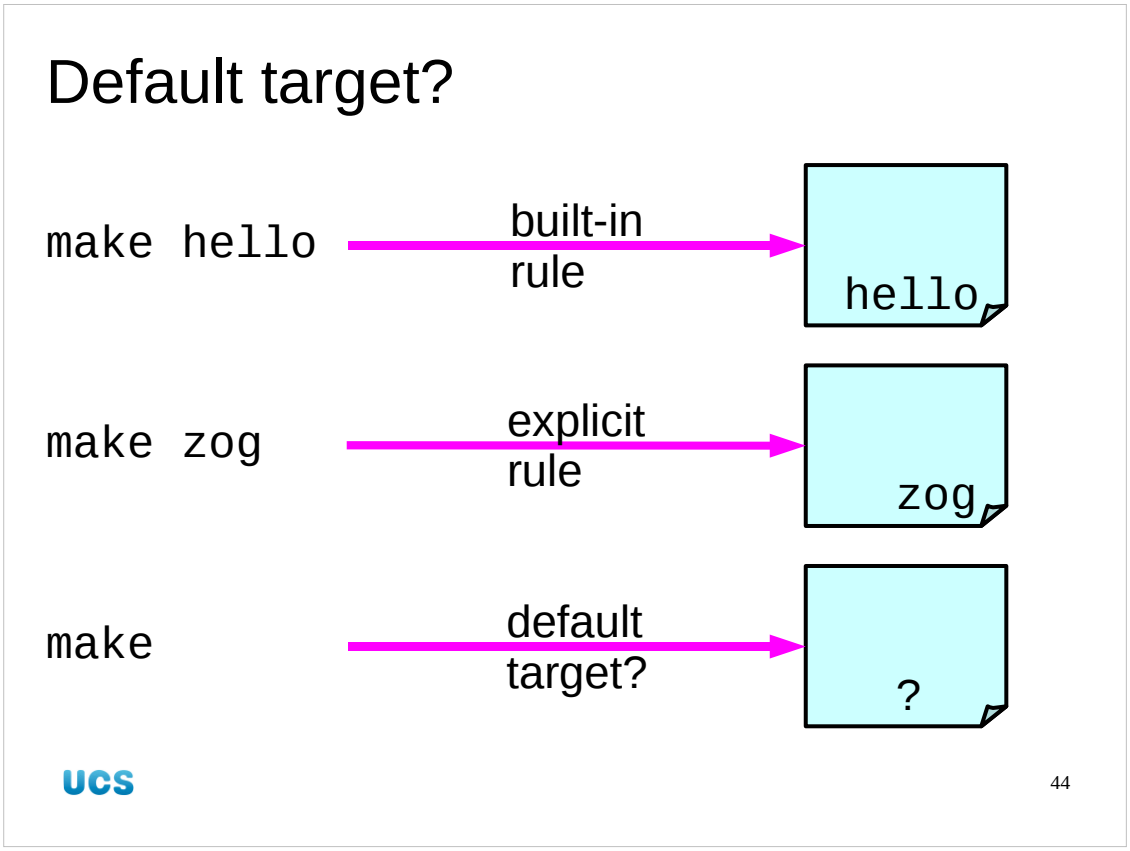
zog

one.o two.o …

flub

UCS

Let's reinforce this point about automatic macros: their values depend on the rule they are in. Here are two rules which have exactly the same line for their actions (link together the object files and generate an executable) defined in terms of the automatic macros. In the two cases, however, the automatic macros evaluate to two different pairs of values, with $^ and $@ being the dependencies and targets of the two rules respectively.



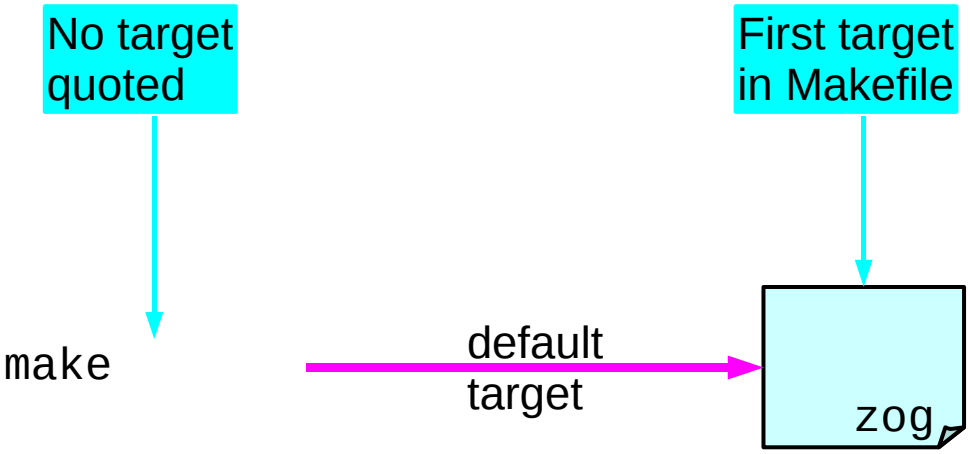
Let's take a step back and look at what we can `make` with our current `Makefile` in place.

If we say "`make hello`" then because there is a `hello.c` file in the directory the built-in rules can build the hello executable program even though there is no mention of it in the `Makefile`. The `CFLAGS` etc. macros that are defined in the `Makefile` are still used.

If we say "`make zog`" then the `zog` target is found explicitly in the `Makefile` and the corresponding actions taken after built-in rules are used to create the dependent object files.

But what happens if we just say "`make`"? What is the default target, the target that gets built if none is specified?

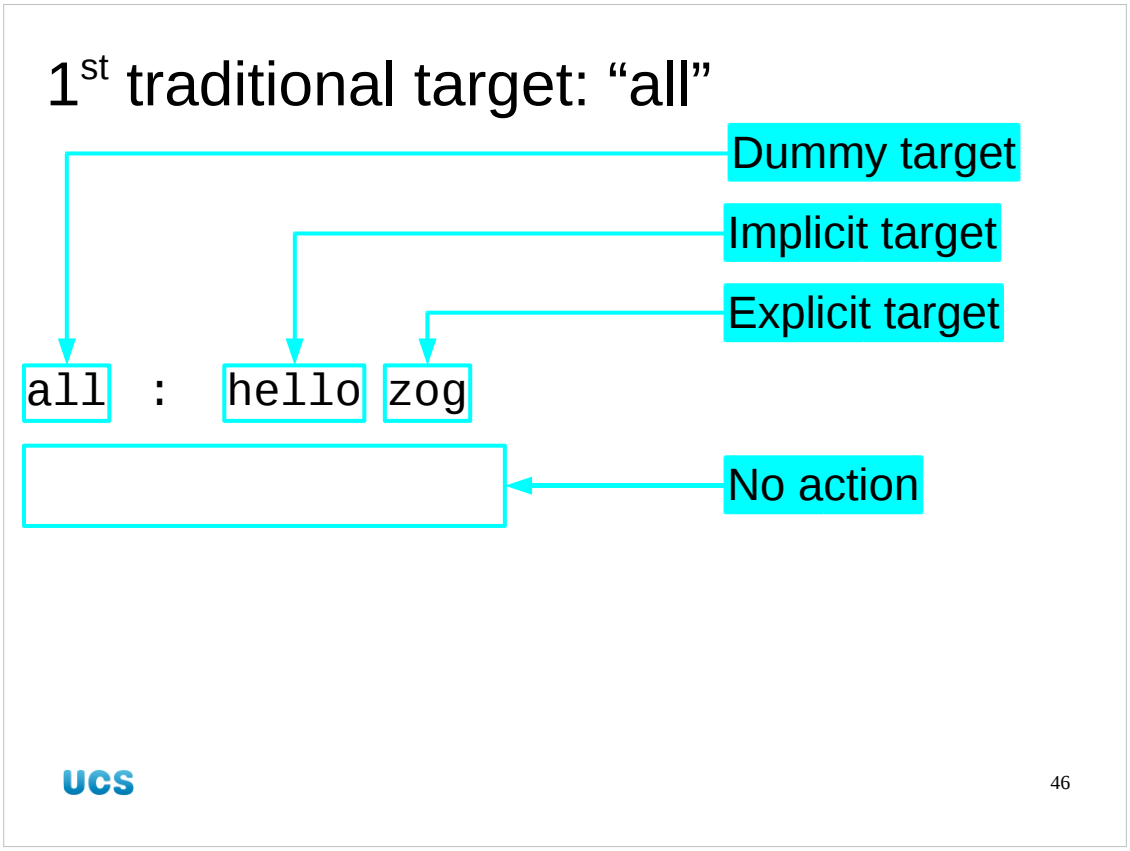
Default target = First in Makefile



UCS

With our `Makefile` the target that is built is “`zog`”.

The default target that `make` builds is “the first target in the `Makefile`”.



There are three targets that can traditionally be found in the `Makefile`, and the first one we will consider is usually placed first in the file so that it is the default target.

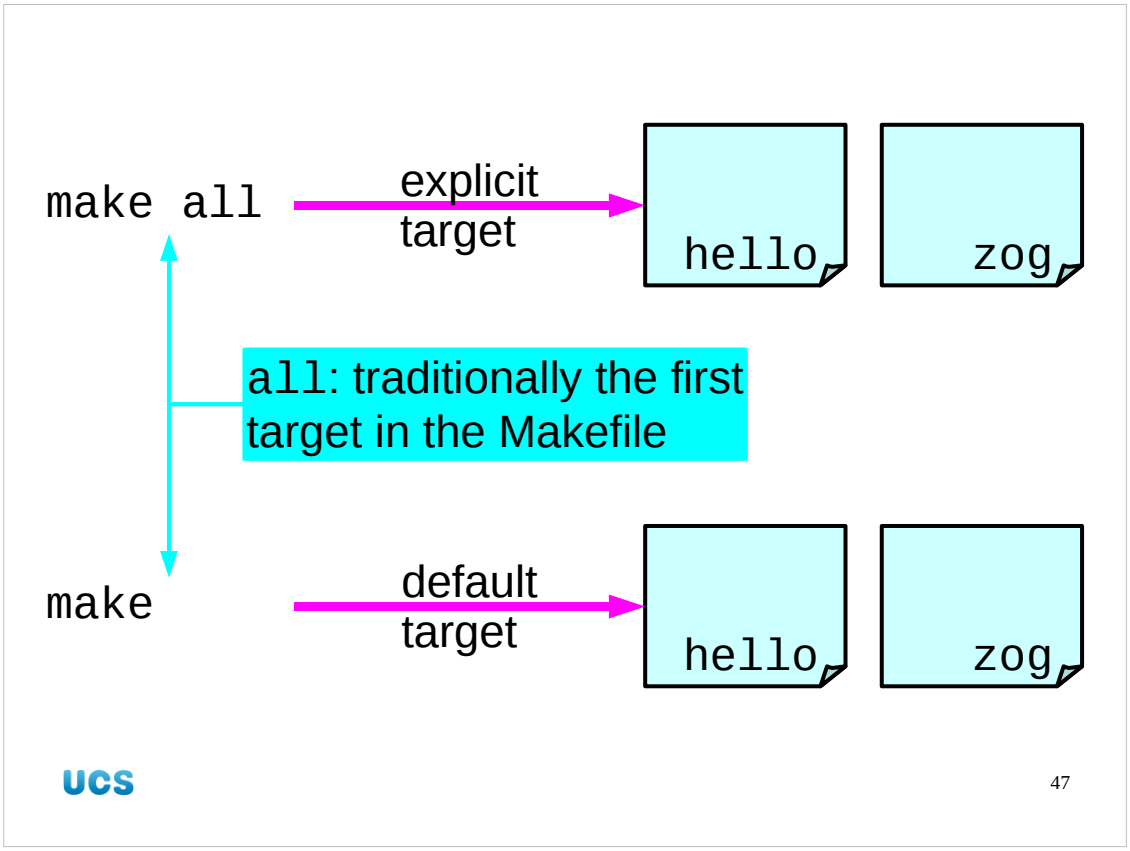
The target is called “`all`”.

This is a target that builds nothing directly; it has no action associated with it.

What it does have are dependencies. The dependencies for “`all`” are the list of all files that you want built from the set of source code in the directory. In our case this is `hello` and `zog`. (This is hello's first appearance in the `Makefile`!)

When `make` is asked to build a target (`all`, in this case) it first checks to see if all of that target's dependencies exist and are up to date themselves. So the result of requesting “`make all`” is for `hello` and `zog` to be made.

In a normal rule there would then be actions to build “`all`” from `hello` and `zog`. In our case there are no actions so `make` stops here.



We typically put the “`all`” target, listing everything to be built, first in the `Makefile` so that the command “`make`” on its own builds everything. This is what the `Makefiles` produced by `configure` do so that in the previous session we only had to type “`make`” after “`./configure …`”.

Worked example

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

all: hello zog

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

48

So now we will update our `Makefile`. Edit it with your favourite plain text editor to add an "`all`" rule which depends on `zog` and `hello`. Make sure that this rule comes before the `zog` rule.

Note that the default rule is the first rule. It does not have to come before the static macro definitions and, indeed, shouldn't. In more complex cases (coming shortly) it may use some static macros.

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
```

UCS

It's easy to test.

We will start from scratch, so remove any copies of the executables and the object files and then run "`make all`" to see if our new rule without any actions does what it should.

I've removed the `CFLAGS` and `TARGET_ARCH` compiler options from the slide so that it fits. You should see something like this:

```
$ rm -f *.o hello zog
$ make all
cc --ansi --pedantic   -march=i686 hello.c   -o hello
cc --ansi --pedantic  -march=i686 -c -o alpha.o alpha.c
cc --ansi --pedantic  -march=i686 -c -o beta.o beta.c
cc --ansi --pedantic  -march=i686 -c -o gamma.o gamma.c
cc  -march=i686 alpha.o beta.o gamma.o  -o zog
```

Worked example

3. Start from scratch again

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

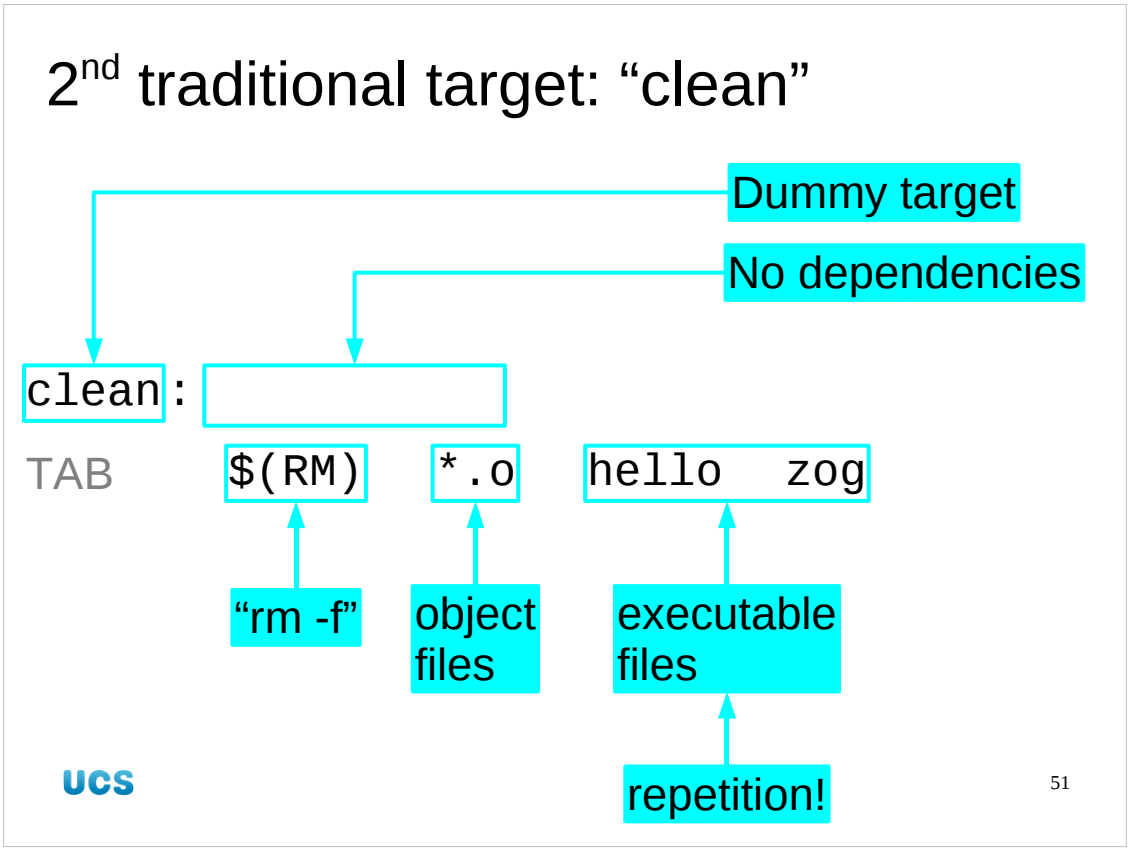
4. Test the default target

```
$ 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
```

UCS

We also need to test to see whether the new rule's “defaultness” is working. So we remove everything and rebuild it.

The output from “`make all`” and “`make`” should be *identical*.



Starting from scratch or just clearing away everything built by the `make` process is quite a common requirement, so the second traditional target we will look at is one that does just that.

The “`clean`” target has no dependencies. This means that it is always “out of date”, so if we give the command “`make clean`” its actions will always be followed.

Its actions are strange for a `Makefile` in that they don't create anything. On the contrary, they remove files.

The `RM` macro is slightly strange. It is defined to be “`rm` with whatever options are required to stop an error code being returned if some of the files being removed happen not to exist”. This is because make stops as soon as it sees an error code, so we don't want false errors. In practice this always means “`rm -f`”.

As for what we need to remove, the answer is “anything created by successful or unsuccessful `make` operations.” In our case this means the two executable programs and the object files. Note that I use “`*.o`” rather than the explicit list “`alpha.o beta.o gamma.o`”. It might have been, for example, that the build of hello failed half way through and left a `hello.o` file laying around. So long as you don't remove anything that's original source you can afford to be zealous with your `clean` actions.

Repetition

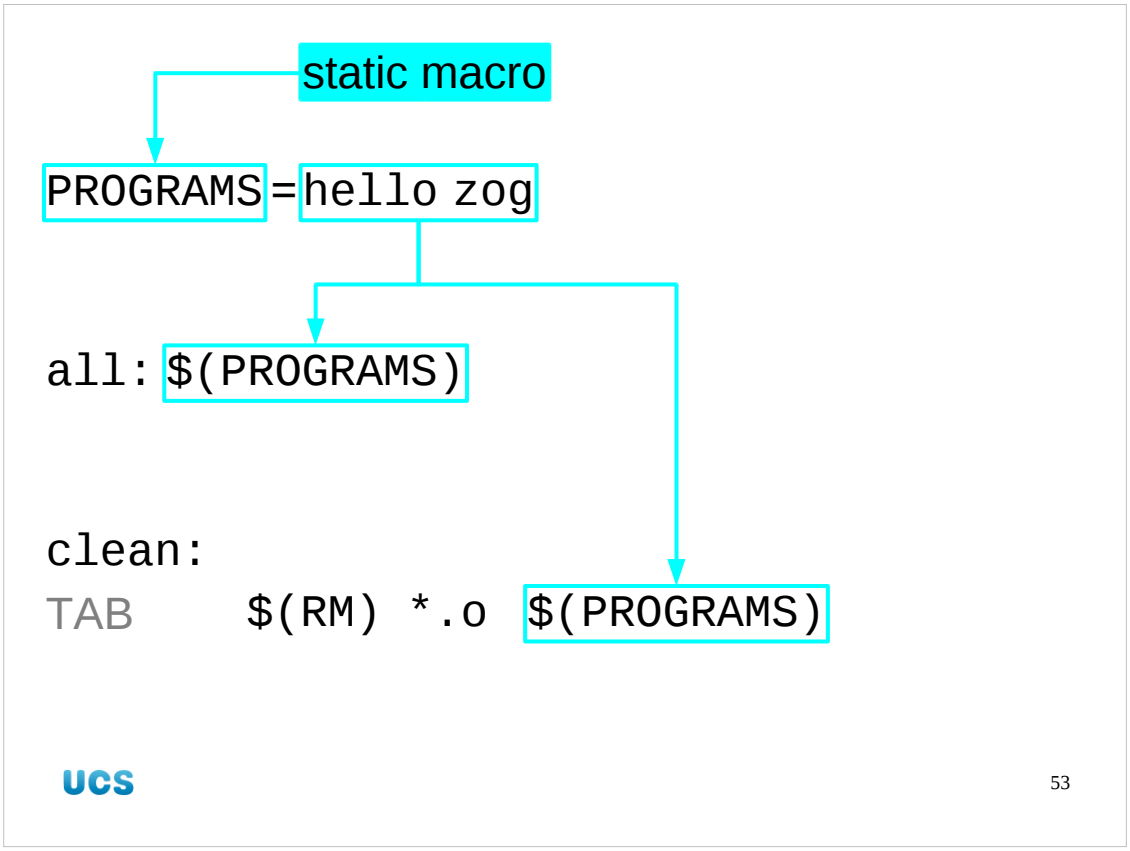

```
all: hello zog

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

UCS

Note that we have now introduced some repetition. This is bad, so we will get rid of it immediately.

Of course, in our simple case the repetition is trivial. Now imagine we were building a huge list of final product that couldn't be matched by some glob like “`*.o`”.

Alternatively, suppose we added another executable program built like `hello`. Would we remember to update “`clean`” as well as “`all`”?



The way we deal with the repetition is to add a static macro defining this list of executables we want built. Here I've called it “PROGRAMS” because that's what we are building. Another common name for it is “TARGETS” because it lists the complete set of ultimate targets we want built. We then use it rather than the explicit lists in all our subsequent standard rules, `all` and `clean`.

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
…
```


Again, we will update our `Makefile` to have this static macro and the new rule.

Worked example

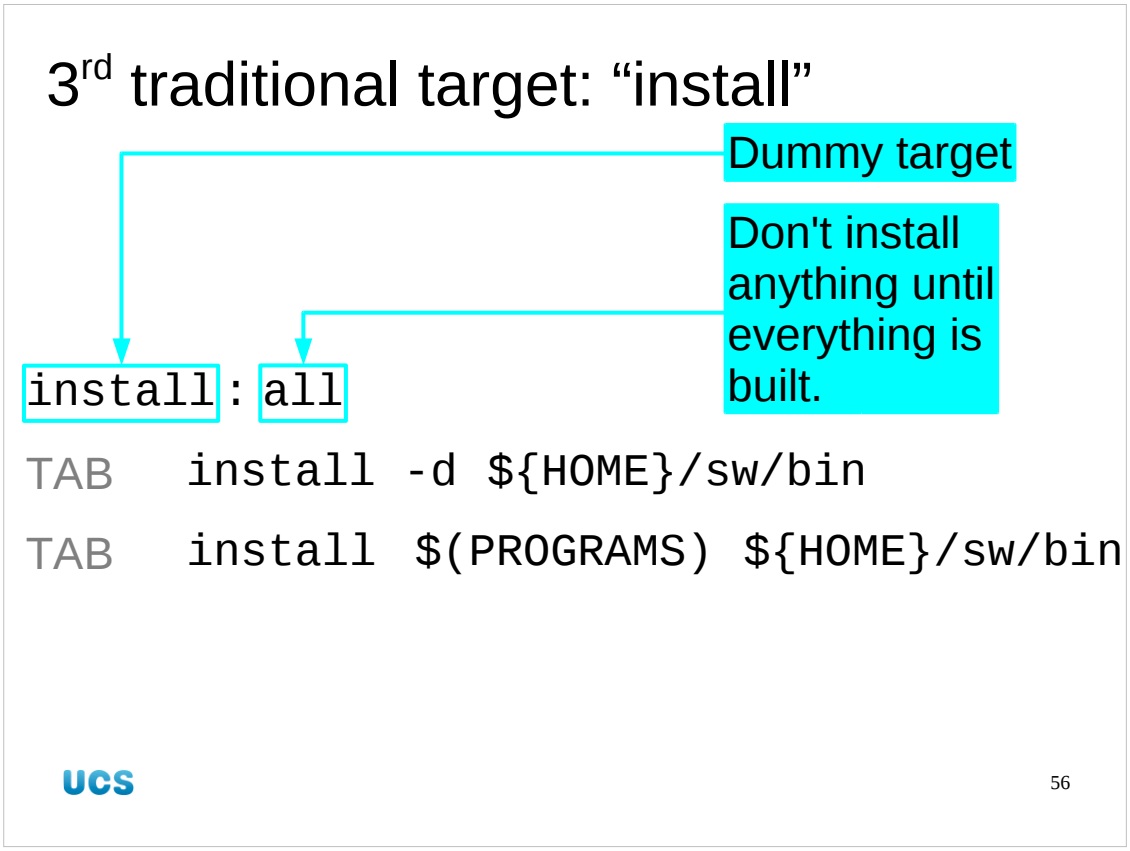
1. Start from scratch.

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

2. Create everything.

```
$ make
```

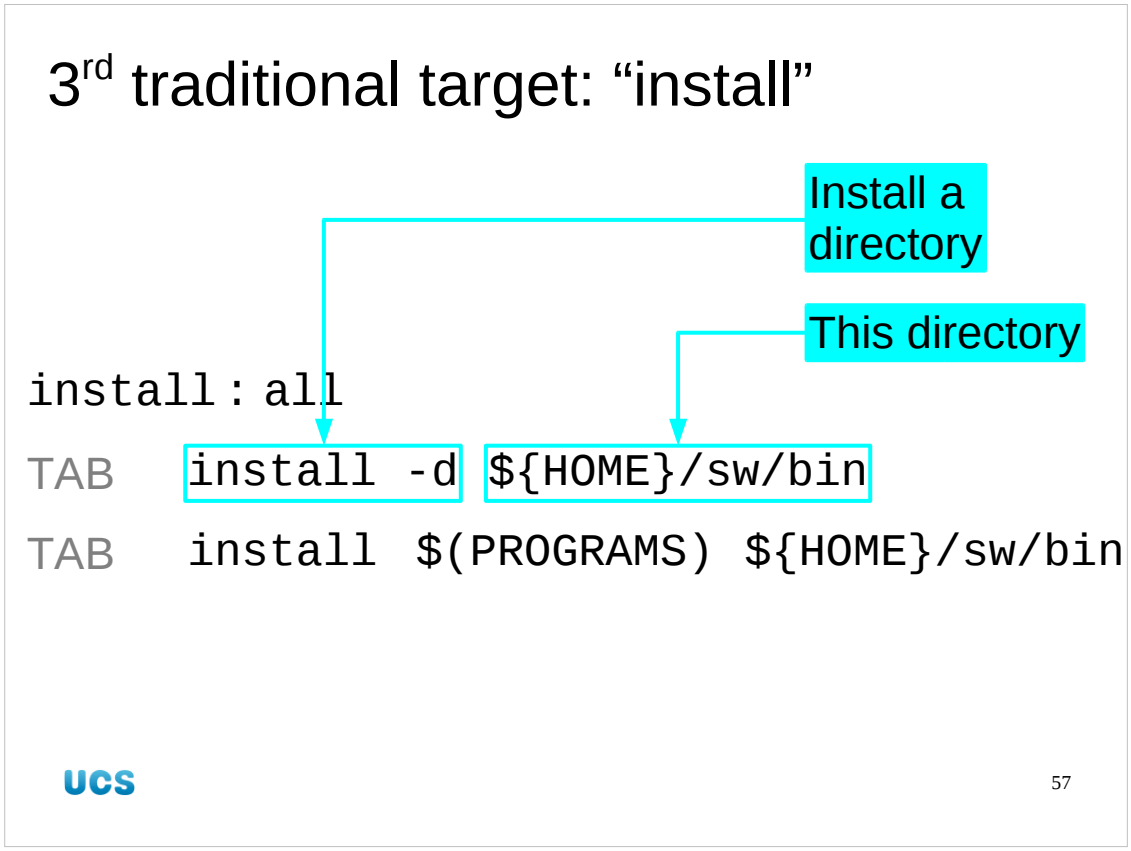

Because we have changed the “`all`” rule as well as adding the “`clean`” rule we should test both.



There is one final traditional target, the “`install`” target that takes the freshly created programs and installs them where the users are going to run them from.

In our case we are installing in `${HOME}/sw/bin`.

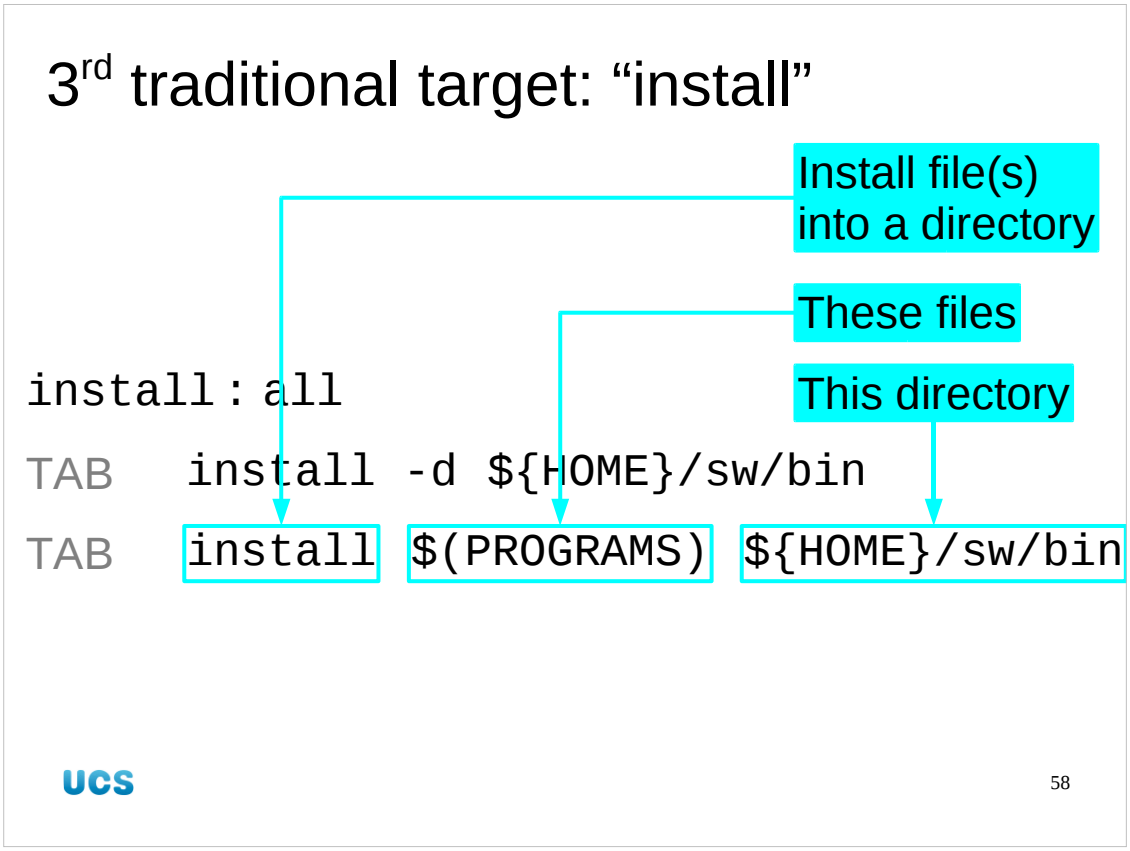
The “`install`” target has a single dependency. This is the “`all`” target. The idea behind this dependency of one dummy target on another is that the actions for the “`install`” target will not start until all the activity for the “`all`” target has been completed. In other words, the installation of software won't start until all of the software has been built. We won't build a bit, install it, build the next bit, install it, and so on.



The actions are an example where multiple lines make sense.

The first line makes sure that the directory we want to install into exists. It is the omission of this line or its equivalent that we worked around with our `mkswtree` script in the previous session.

The `HOME` environment variable is in braces (curly brackets, `${…}`) which expand environment variables, and not parentheses (round brackets, `$(…)`) which expand `Makefile` macros.



The second line installs the programs into the directory (which must already exist). Note the re-use of the PROGRAMS macro to avoid writing in the list of executable programs.

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
```

UCS

If we were building libraries, manual pages, info pages, graphics files etc. then we would have more lines beyond these two. A more fully formed installation rule might look like this.

Repetition

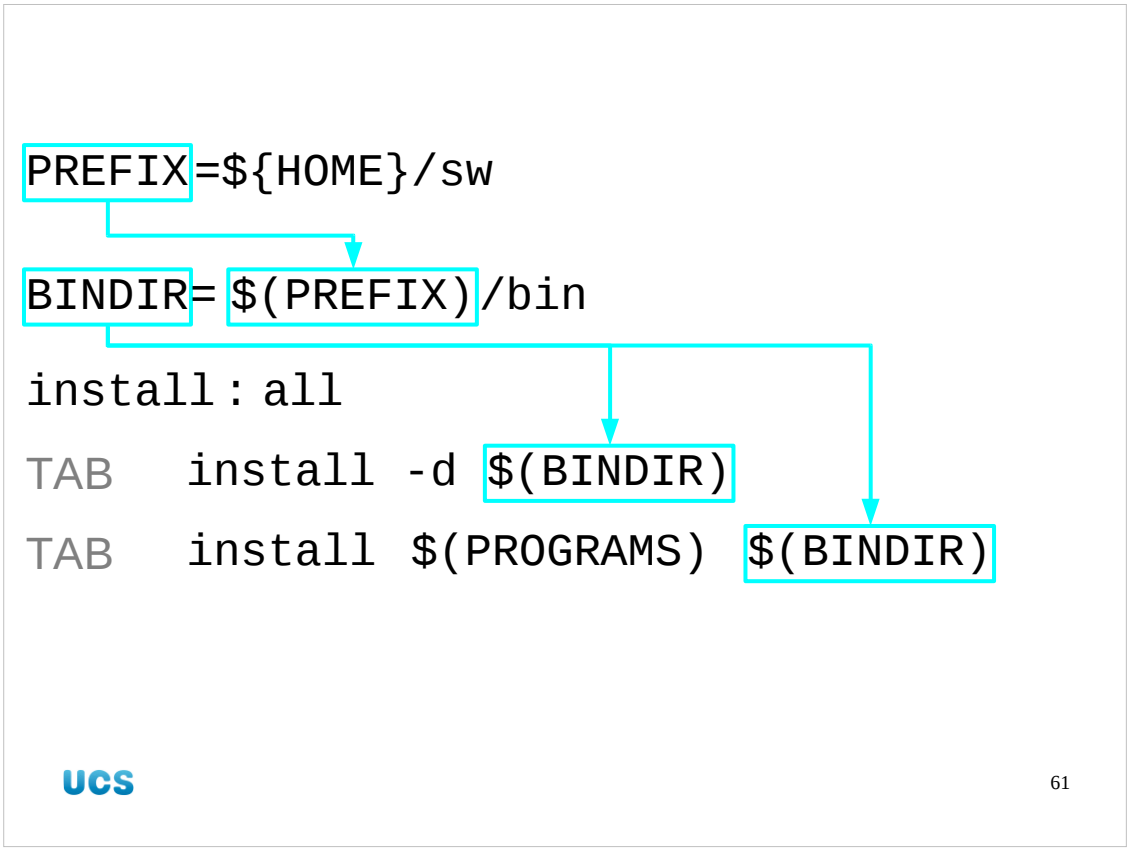

```
install : all
TAB    install -d ${HOME}/sw/bin
TAB    install $(PROGRAMS) ${HOME}/sw/bin
```

UCS

Once again, however, we have strayed into repetition. This time we are repeating the installation directory.

Repetition is bad, still, but this time for different reasons.

It's unlikely that the two instances so close together will ever get out of step. But if we, or the person we pass this build system on to, ever want to change the installation location we have to change two items. If the makefile is long, these may take some hunting for. Also, consider how much work would be required for our larger example.



Again, we introduce a static macro. But we do it in two stages.
What we are most likely to want to change is the directory `${HOME}/sw` rather than how things are allocated within it. So we will create a macro, `BINDIR`, for `${HOME}/sw/bin` but we will define it in terms of another macro, `PREFIX`, which identifies that top-level directory. We take its name from our use of the "`./configure --prefix`" option in the previous session.

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)
...
```

UCS

Again, the benefits of this two-level approach increase with the complexity of the set of targets.

Updating our Makefile

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

PROGRAMS=hello zog

all: $(PROGRAMS)

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


Now we will update our `Makefile` with the installation instructions we need.

It doesn't matter whether this target comes before or after “`clean`”, so long as it comes after “`all`”.

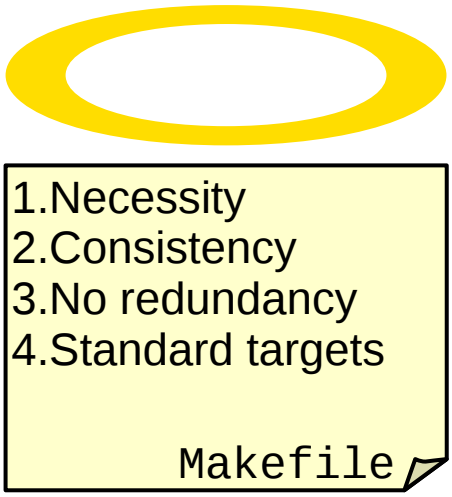
Worked example

```
$ 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
```


We can test this rule easily.

Note that because we tested "`make clean`" most recently, we also get to test the implied "`make all`" from the "`install`" target's dependency.

Principles for a good Makefile



Now we are done with `make` and `Makefiles` for now. So let's have a quick recap of what makes a good `Makefile`.

Necessity: We have added rules only where we needed to. For example, we didn't need to add rules for hello, or for any of the object files, so we didn't.

Consistency: We have copied and adapted the system rules wherever possible and used the same static macros as they have so that all our activity is consistent.

No redundancy: We have ruthlessly stamped out any redundancy, defining our own static macros where necessary.

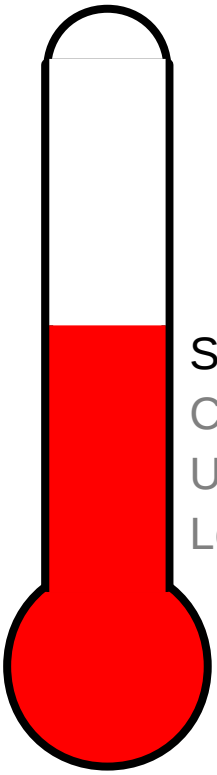
Standard targets: We have defined the three standard targets: `all`, `clean` and `install`, with `all` being the first, default target.

`make`

built-in rules

`Makefile`

static macros
explicit rules
automatic macros
standard targets



Simple make
Configured builds
Unpacking
Location

UCS

And that completes our work on `make`, for now. We will return to it in the next session.

Coffee break

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

Alphonse Allais

Ten minutes

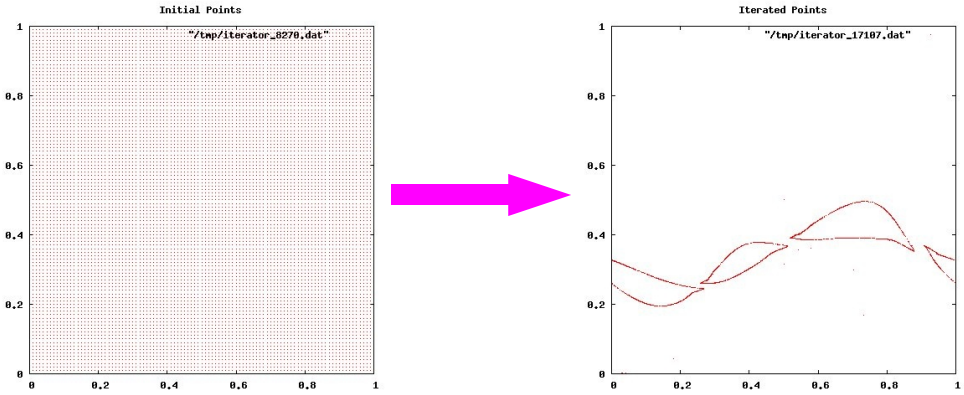
UCS

Let's take a break.

Using libraries

Iterate points

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



UCS

While we are going to meet make in the second half of this session, we will just be using it as a tool to drive the building of libraries. What we will be investigating in depth is how to get executables built that use external libraries.

We will start with a simple example. We will build a simple program that iterates a set of points in the unit square a thousand times under a simple iteration formula as shown. It's vaguely interesting from a mathematical perspective, and the points converge to a simple pattern for values of the parameter ε around 0·5.

From the software perspective what is interesting is that it uses the sin() function. This function is found in a library that is not linked in by default. We will have to take extra measures to have it used.

Worked example

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

   `iterator.c`

   `iterator.sh`

UCS

Let's have a worked example. Please copy into position these two files. The shell script, `iterator.sh`, is just a wrapper to give us pretty graphical output. The computational component is in the to-be-compiled file, `iterator.c`. That's the file we will be concentrating on for the purposes of learning how to build software.

```
$ cp /ux/Lessons/Building/iterator.* /tmp/building
```

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!

UCS

The file `iterator.c` is a complete program except for a catch we are about to see.

We can treat it like a complete program and attempt to make to executable in one leap with the command "`make iterator`".

The build fails with an error message saying that the program referred to the sin() function without defining it. Its definition lies elsewhere in the maths library.

[I need to make a quick apology. The maths library was written by Americans. Its real name is, therefore, the "math library" with no "s". But I'm a Brit and no matter how hard I try my tongue is programmed to use the word "maths". I'm sorry if this confuses. When you are searching the documentation (and you do all read the fine manuals, don't you?) you need to remember that the word is written in the singular. Damned colonials.]

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

By the way, this is our first serious error out of a make-driven build. We ought to spend at least a couple of minutes looking at it. We will cover how to read make logs systematically in the next session when things get far uglier.

```
cc --ansi --pedantic   -march=i686 iterator.c   -o iterator
```

By default `make` prints out the commands it is running. In this case it is building straight from the source file.

```
/tmp/cco7S3A.o
```

Remember that I said that when the compiler seems to skip the object file step all that's really happening is that the object file is hidden out of the way. Well, here it is. It's a temporary file with a randomly chosen name.

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

`In function `main': iterator.c`

Now the compiler tells us that it was working on a function called `main()` in the file `iterator.c`.

Every C program needs a function called `main()`. It's the function that is run when the program is called.

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

UCS


```
undefined reference to `sin'
```

Here's the actual error message in the midst of the full error report.

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")

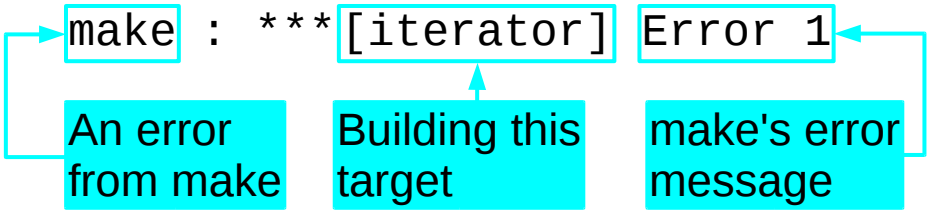
UCS

`collect2: ld returned 1 exit status`

"`ld`" is the linker. It's also called the loader (though the two functions are typically merged these days) so its command is "`ld`" for "load". Recall that it's the linker that has the responsibility of tying together uses of functions and definitions of functions, so it's the linking stage that's actually failing.

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
```



UCS

Finally we get `make` reporting that there was an error from the compiler that it launched.

```
[iterator]
```

The term in square brackets is the target that `make` was trying to build.

```
Error 1
```

The error code is `make`'s less than helpful summary of what went wrong. We will return to make error messages in the third session.

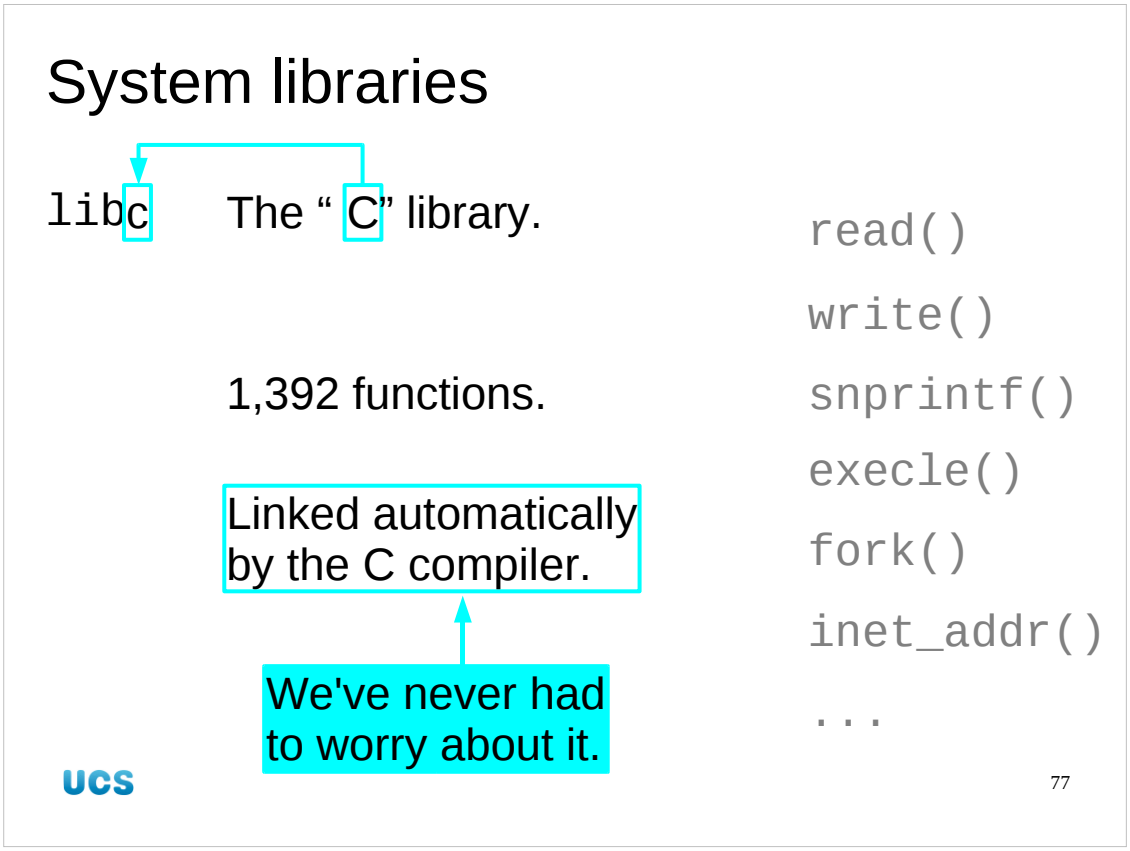
Worked example

```
$ 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!

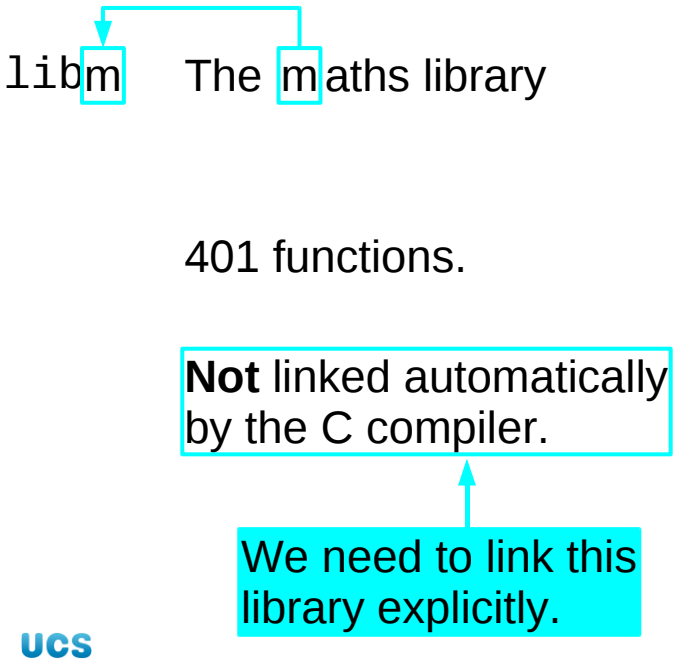
UCS

Back at the chase, we have an error we need to deal with. We are using the `sin()` function defined in the maths library but we aren't telling the compiler to use the maths library. It's dumb. It doesn't know `sin()` is “obviously” a mathematical function. So we need to be explicit.



We are, in fact, already using a system library. The “C library” contains well over a thousand standard functions that any C program might want to use. These are not part of the C language itself but are provided by a library of functions that is included by default. Because it is automatically linked in without any explicit instruction from us we have never had to worry about it in the past. It is unique in this regard, though. Any other library we want to add will need explicit instruction.

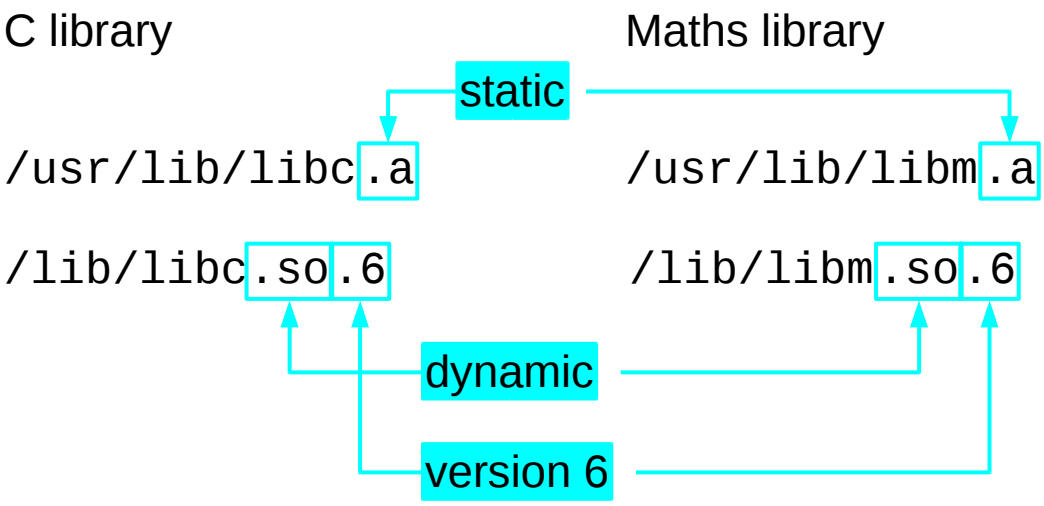
System libraries



UCS

The maths library is an example of a “typical” library in this regard. We will need to give explicit instructions to link it into our executables that need it.

System libraries: the files



The libraries are stored in files, of course, with different files for the static and the dynamic versions. The two standard directories for system libraries are `/lib` and `/usr/lib`.

One of the advantages of the dynamic library approach is that the library can be patched (have bugs fixed or other improvements made) without having to rebuild every executable that relies on them. All that matters is the set of functions defined by that library (and what arguments those functions take and return). This is specified by a version number. For example, the current C library (and maths library) is at version six. This is reflected in the naming of the library file: `libc.so.6` for version six of the shared object version of the C library. It's quite possible to have `libc.so.5` and `libc.so.6` sitting side-by-side with version six there for modern programs and version five there for back compatibility.

System libraries: cc options

C library

`/usr/lib/libc.a`

`/lib/libc.so.6`

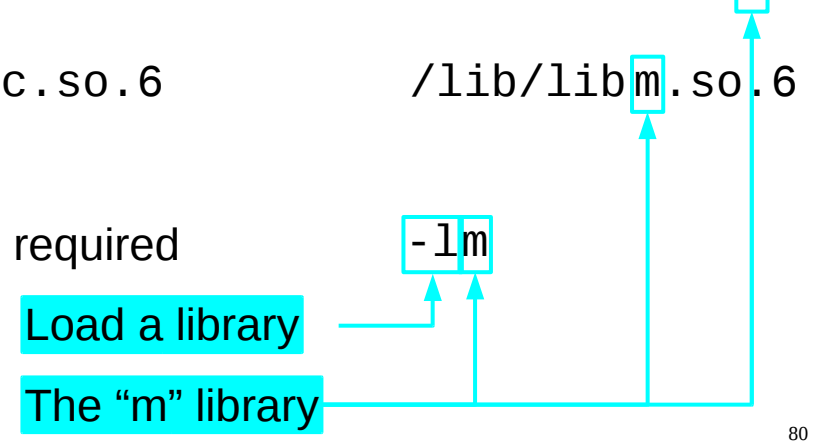
No options required

Maths library

`/usr/lib/libm.a`

`/lib/libm.so.6`

`-lm`



UCS

So what options do we need to use the library in a file?
For `libc` no extra option is required. For the maths library, `libm.so.6`, we use the option “`-lm`” where the “`m`” identifies which library to use.

System libraries: the Makefile

C library

```
LDLIBS=
```

Empty value: the default

Maths library

```
LDLIBS=-lm
```

Link in lib**m**

Of course, we won't be using the compiler directly; we'll be using make. The corresponding macro in a Makefile is `LDLIBS` and we set *it* to be "`-lm`".

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.

So we make that change to our Makefile in the ongoing worked example. Note that this will add the maths library to every program built from this Makefile, regardless of whether or not they need it. This is why you should have one Makefile per project, rather than shovel everything together as we are for this course. Nonetheless, add `iterator` to the `PROGRAMS` macro to get used to doing it.

Worked example

4. `$ make iterator`

   `cc … iterator.c -lm -o iterator`

5. `$ ls -l iterator`

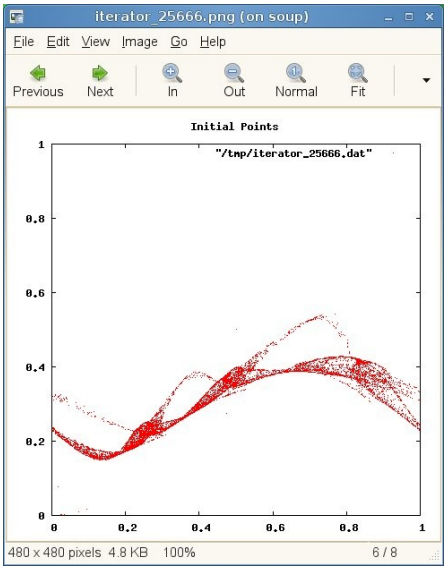
   `-rwxr-xr-x 1 … iterator`

UCS

Next, make the `iterator` program.

Worked example

6. `$ ./iterator.sh 0.60`



UCS

The `iterator` program simply generates large numbers of point definitions. There is a wrapper script in `iterator.sh` which surrounds `./iterator` with the programs to generate graphical output.

(See the UCS Gnuplot course if you want to understand the graphical generation system.)

Other libraries

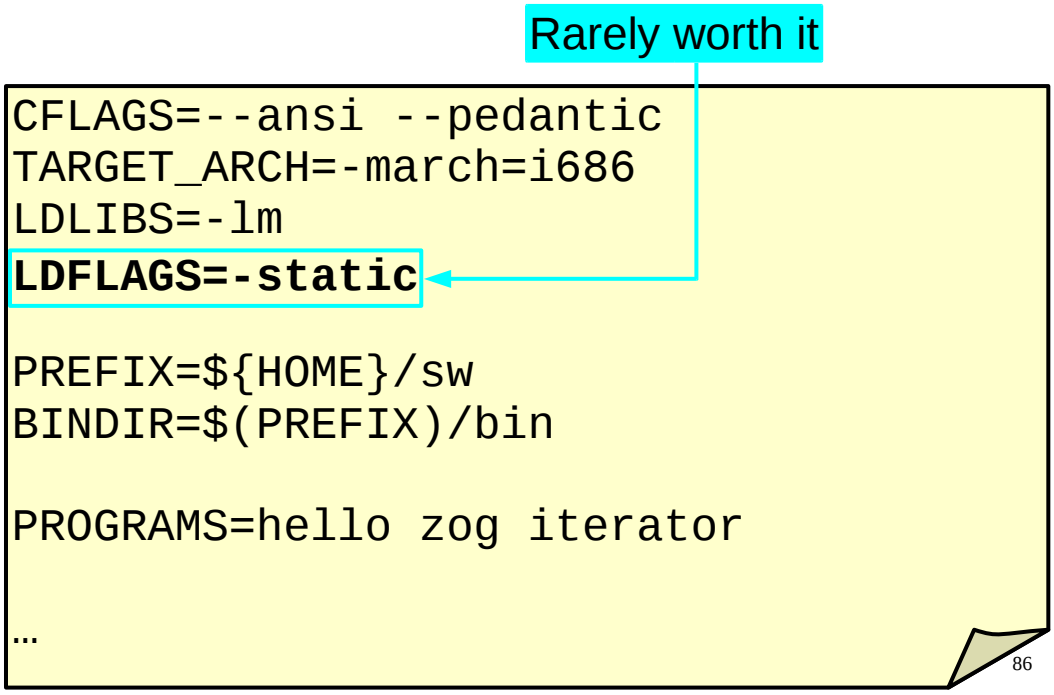
Maths library	Other library
`/usr/lib/libm.a`	`/usr/lib/libthing.a`
`/lib/libm.so.6`	`/lib/libthing.so.`*`n`*
`-lm`	`-lthing`

UCS

The approach of "`-lm`" linking in the library "`libm`" can be extended to any library whose name follows the convention of starting with "`lib`" followed by its name followed by either "`.a`" or "`.so.`" and a version number.

So if we had a library (either in `/lib` or `/usr/lib`) called "`libthing.so.4`" then we would link it in with the option "`-lthing`".

Static linking



We still have the choice of static or dynamic linking. Which will it be?

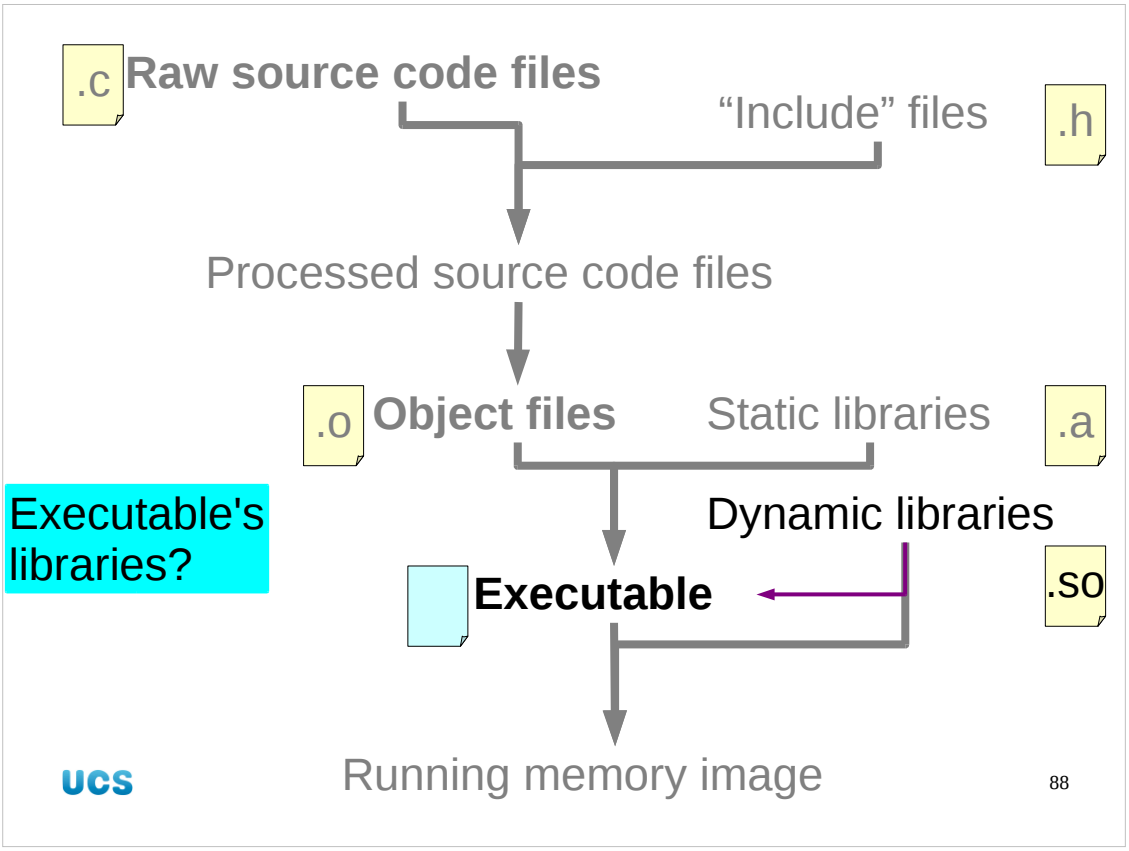
By default, and you are advised not to override it, the default is to link dynamically if possible. If you do want to then add the "`-static`" (single dash) option to the compiler. This should be done through the `LDFLAGS` `Makefile` macro.

Size difference

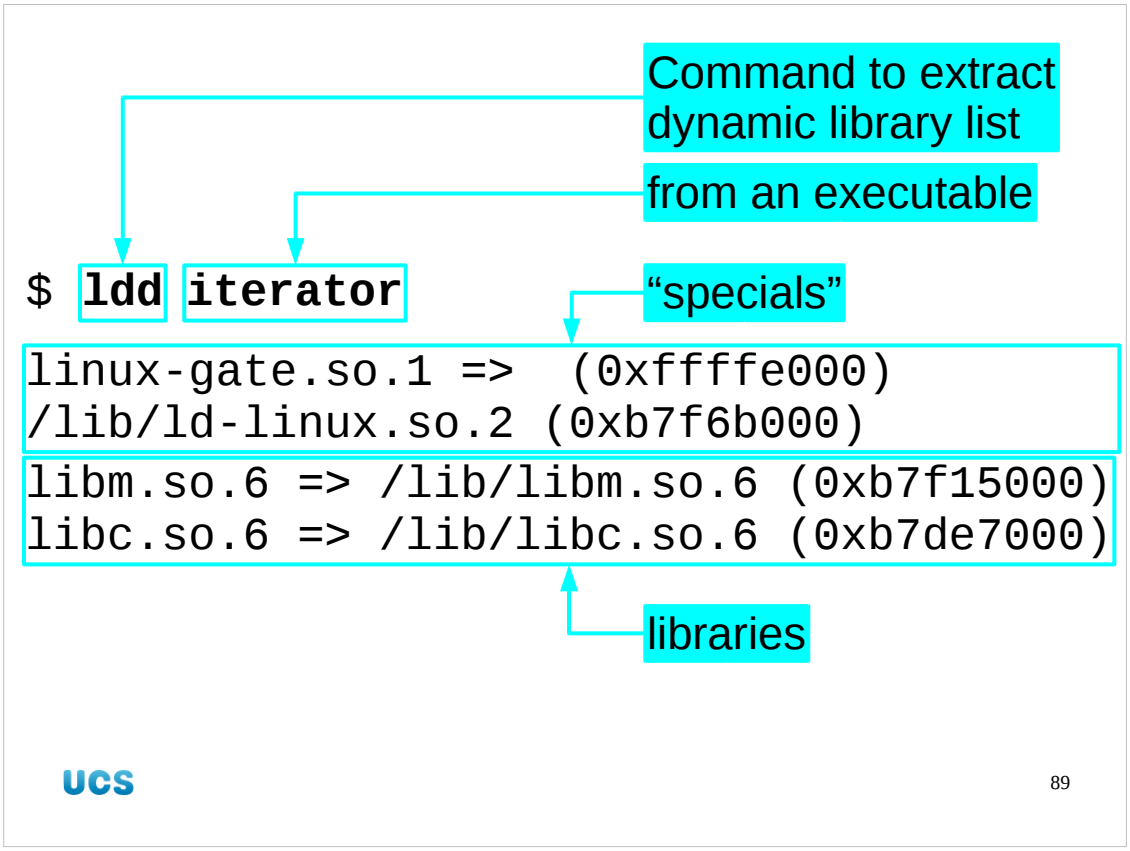
```
$  ls -l iterator
```

Dynamic:	11,065 bytes
Static:	2,338,249 bytes!

Static linking adds all the necessary maths functions into the executable itself. This has dramatic implications for the size of the executable.



Let's reverse the question for a moment? Instead of asking how we link libraries into an executable, let's ask how we can ask what dynamic libraries have been linked into a given executable.

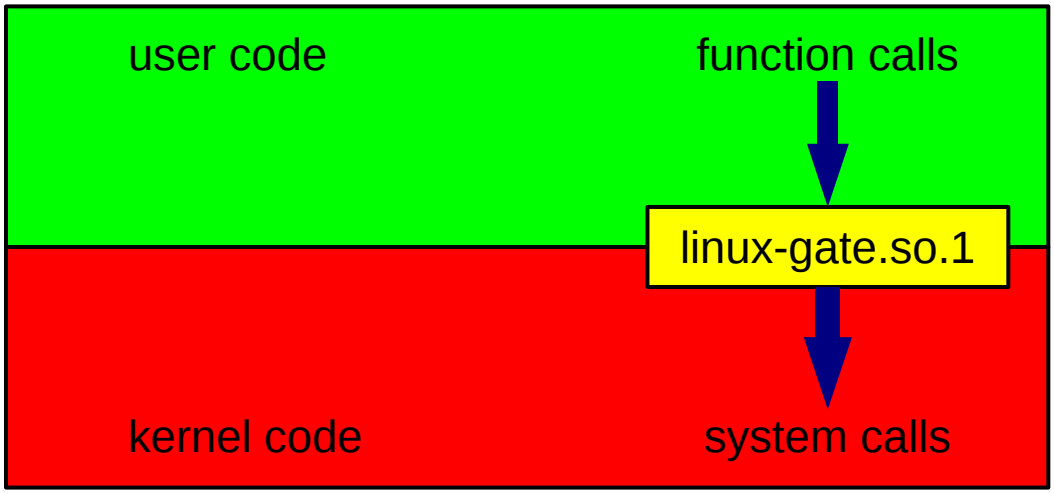


This is done with the command “`ldd`”. This reveals the dynamic library names linked into an executable and the files those library names currently correspond to.

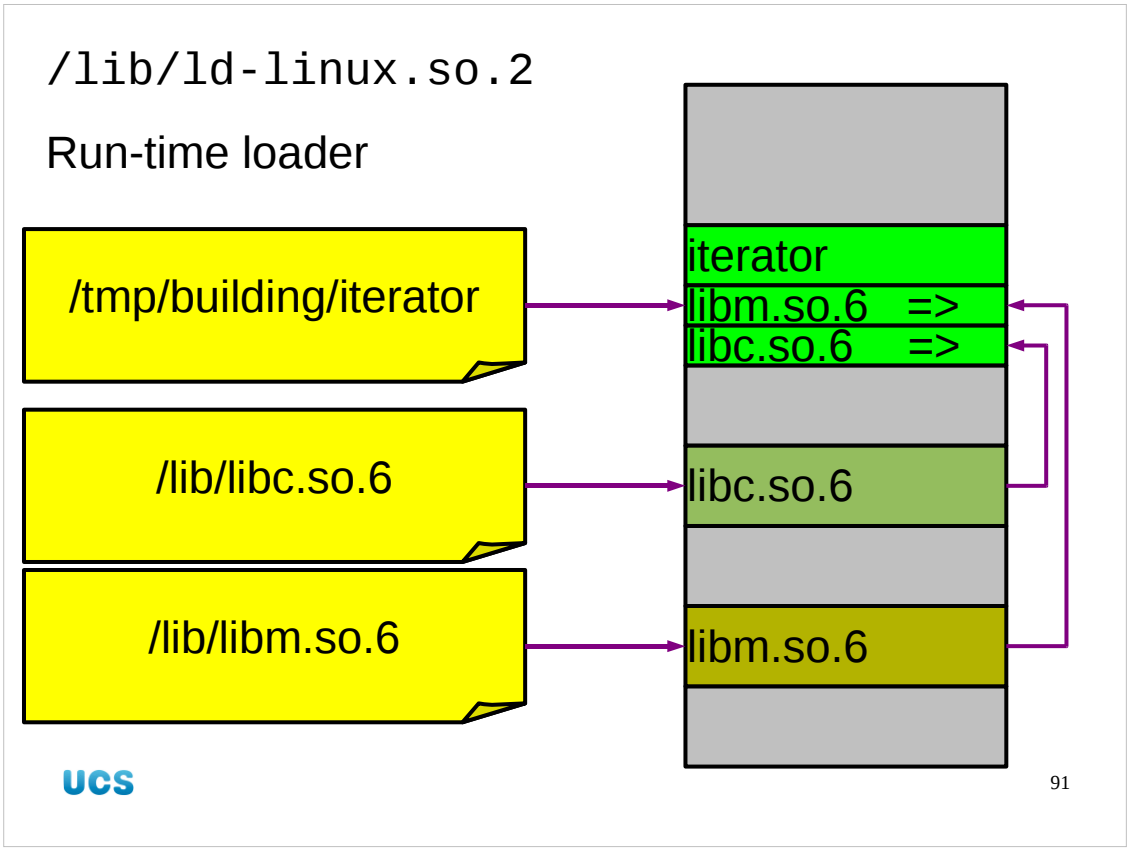
Note that there are a couple of “specials” in the output that look a bit like libraries but which don't match to file names. We will address those first.

`linux-gate.so.1`

“Virtual dynamic library”



The dynamic library name `linux-gate.so.1` is wired into the executable by the compiler and provides the set of hooks to call the kernel, the base operating system at the heart of the computing environment.



The entry `/lib/ld-linux.so.2` is the hook into the dynamic linking system. The run-time loader is responsible for making sure the relevant library files are mapped into memory and that the library references in the in-memory copy of the executable are set to point to these memory copies.

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

So we need to see this in practice.
We will build another library. This is called the "GNU Linear Programming Kit" or "GLPK" for short.
Linear programming is the mathematical process of taking a set of variables, x_j, and varying them subject to a set of linear constraints, $C_{ij}xj \geq K_i$, to minimize a linear function $B_j x_j$.
This is a standard four-step installation as we saw in the first session. Building and installing it should give you no grief at all.

```
$ cd /tmp/building
$ tar -xf /ux/Lessons/Building/glpk-4.40.tar.gz
$ cd glpk-4.40
$ ./configure --prefix="${HOME}/sw"
…
$ make && make install
…
```

Coffee break

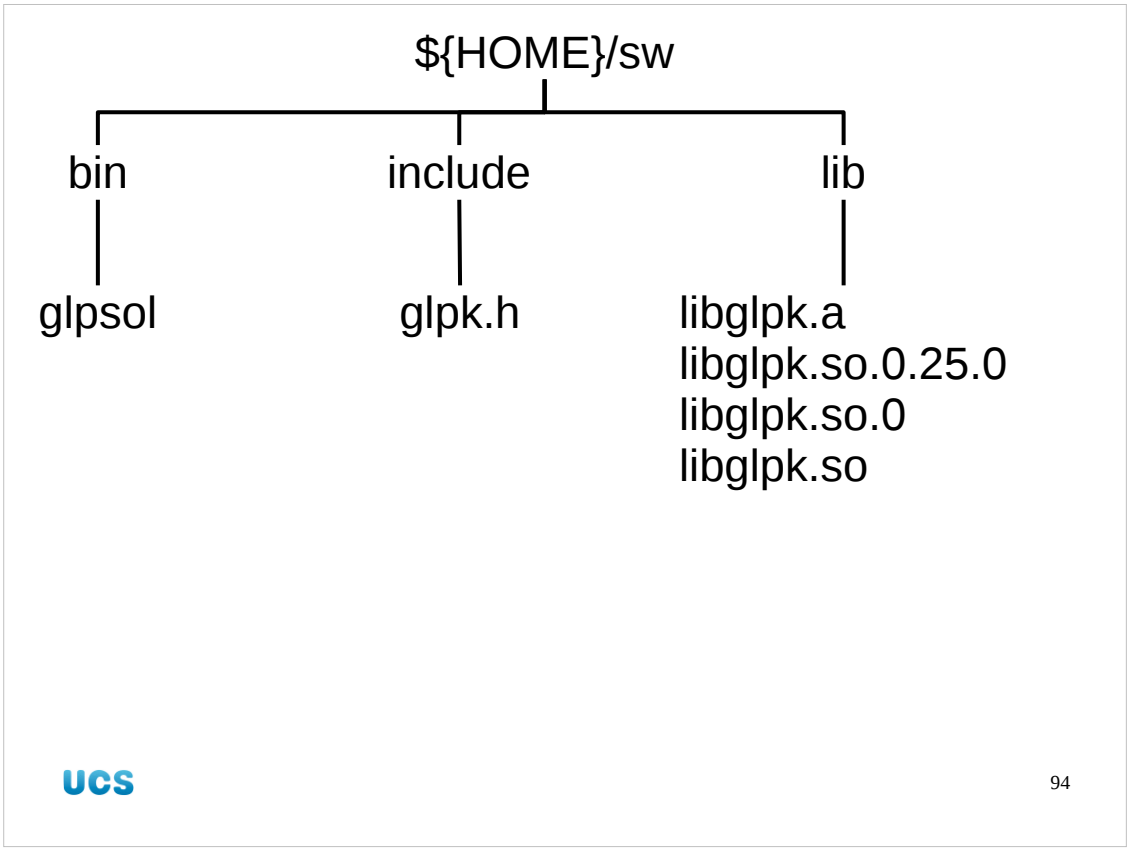
“In Seattle you haven't had enough coffee until you can thread a sewing machine while it's still running.”

Jeff Bezos

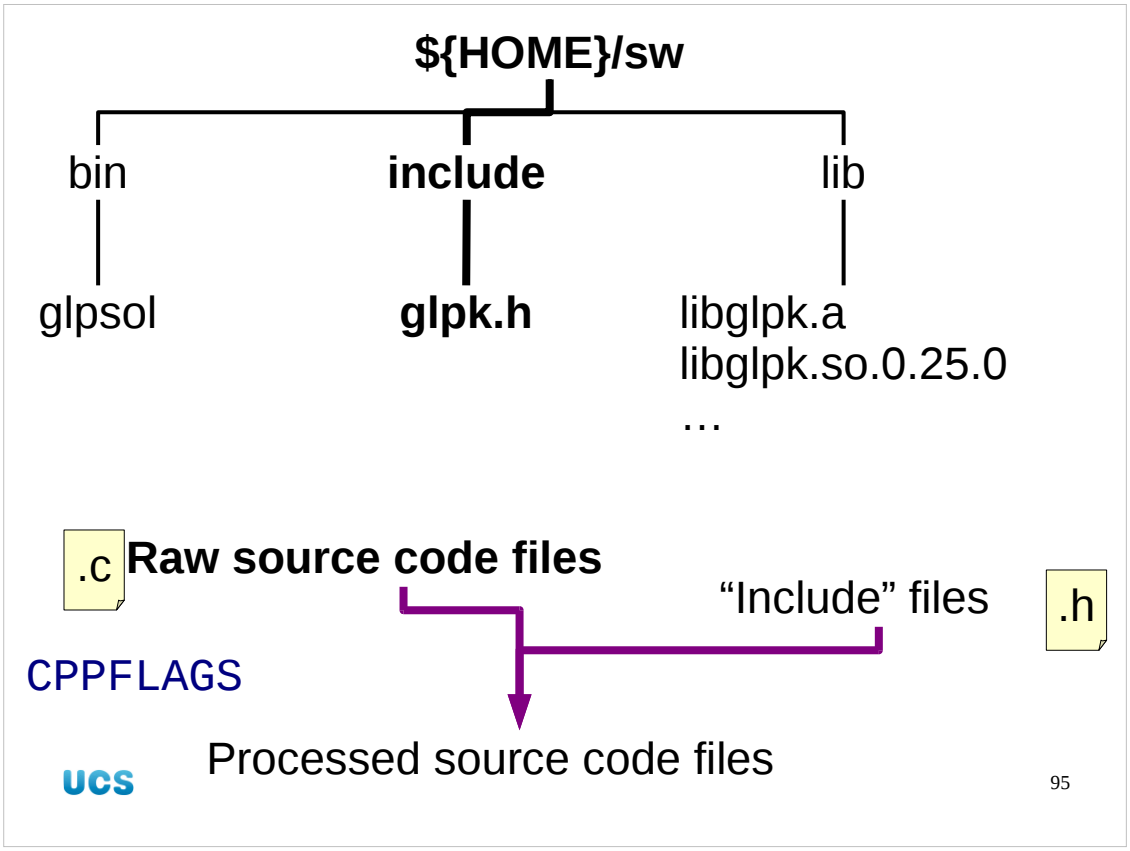
Ten minutes

UCS

This is an easy build so it should be over in less than ten minutes.
There's plenty more caffeinated wisdom at
http://www.quotegarden.com/coffee.html

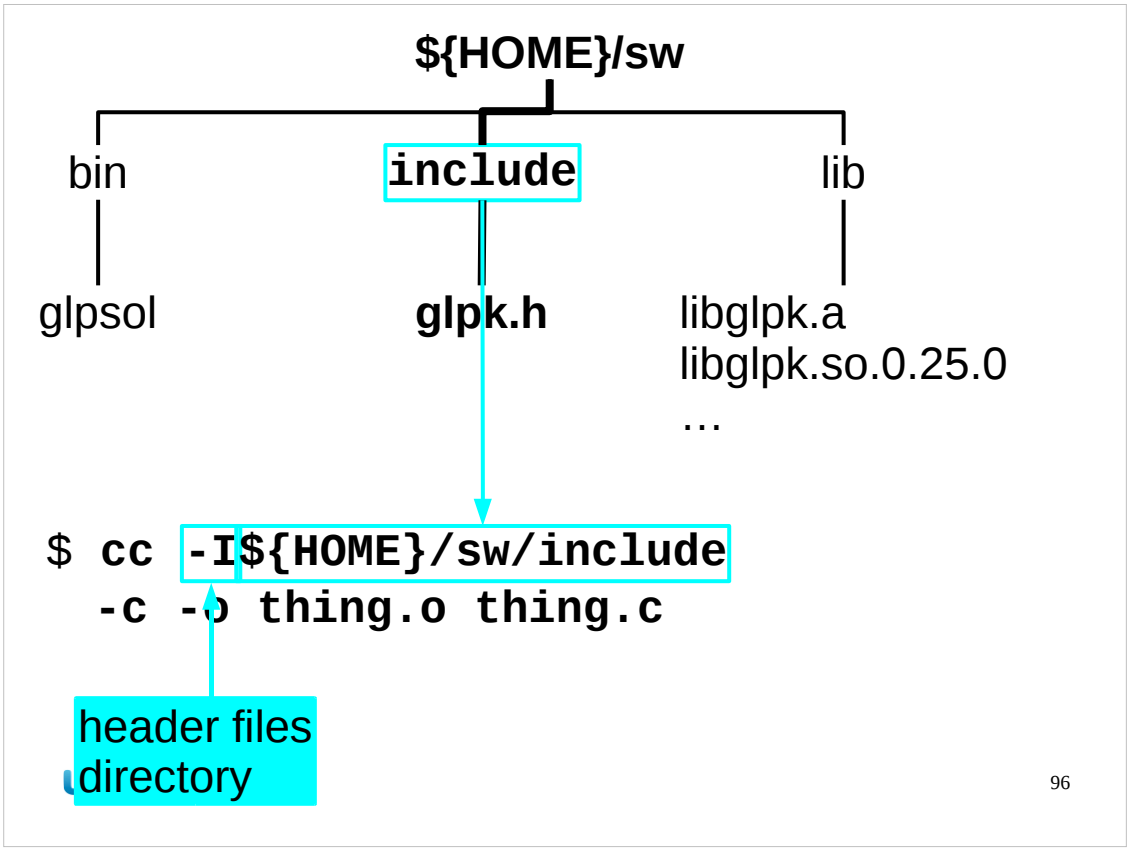


We will take a look at what has been installed.



There is a header file installed in `${HOME}/sw/include`. Header files are used by the pre-processing phase that takes raw C source code files and converts them into “pre C” ready for compilation into machine code.

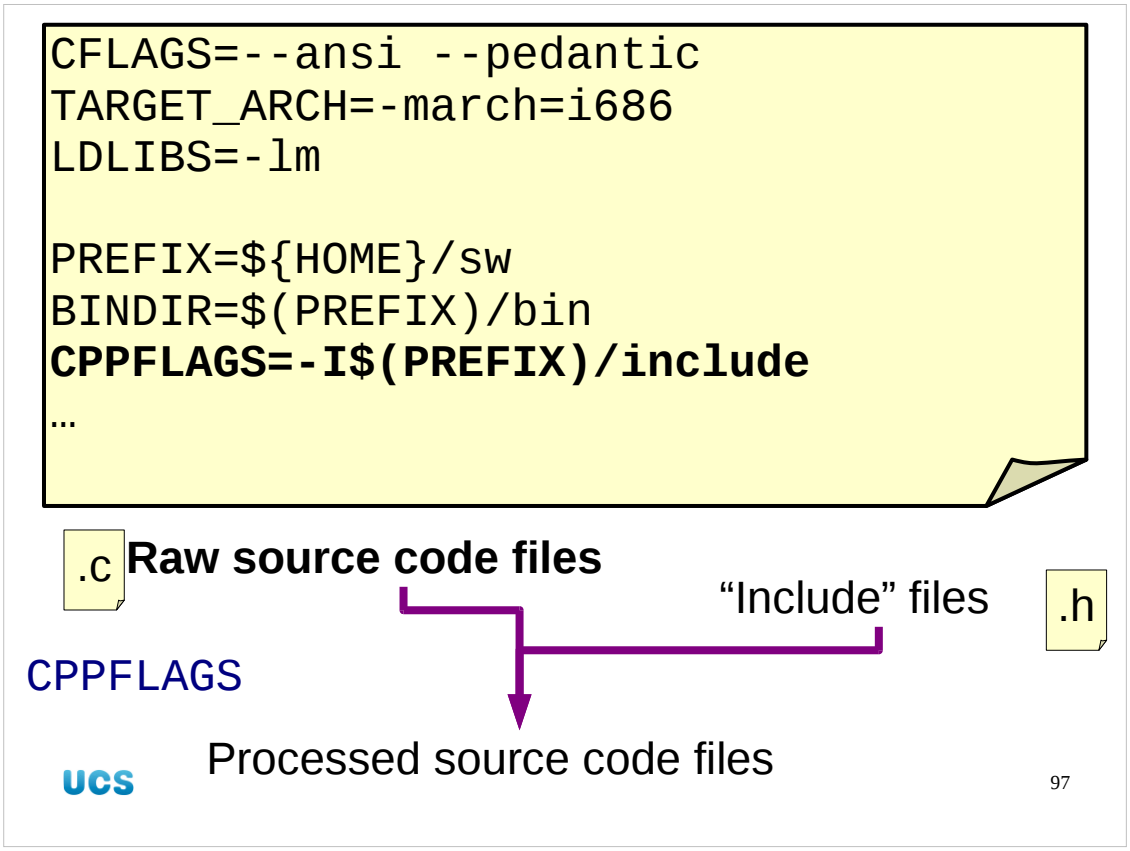
Recall that any options needed for this phase are set in the `CPPFLAGS` `Makefile` macro.



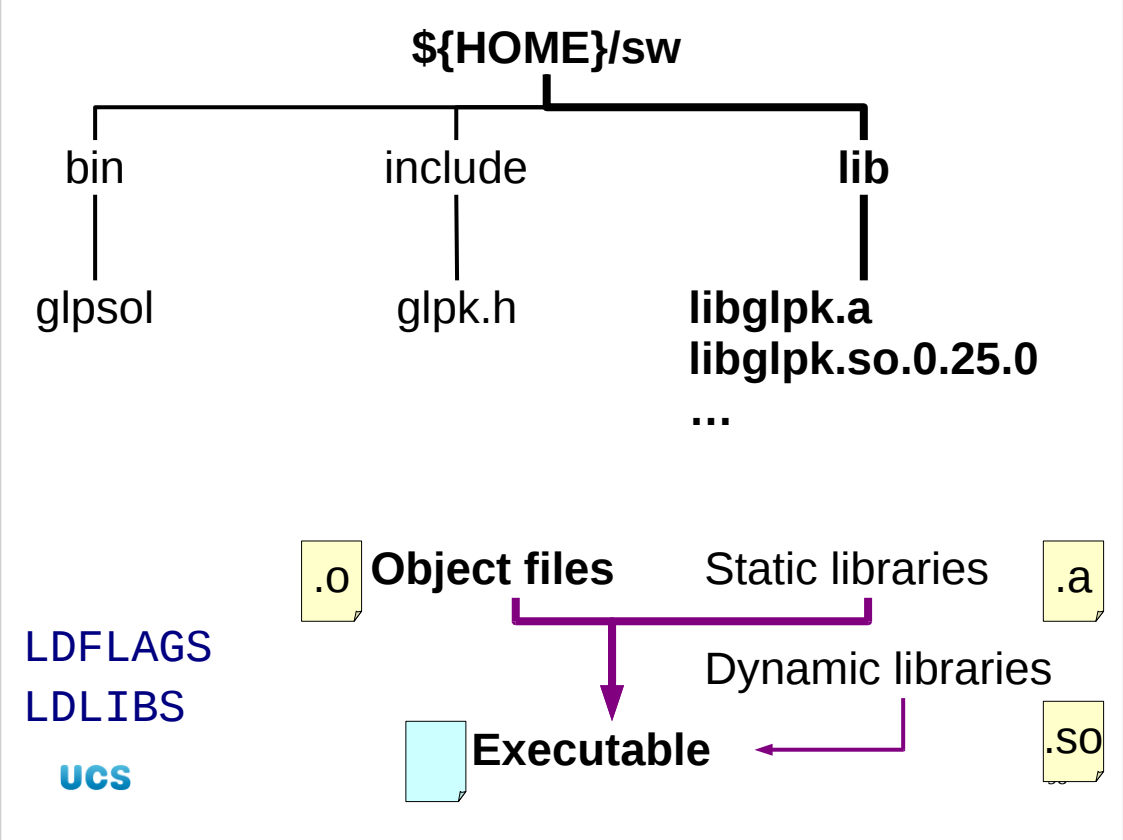
THE COMMAND IN THE SLIDE IS ILLUSTRATIVE. DO NOT RUN IT!

If we are going to use these header files we need to tell the compiler where to find them. They are not in any standard system location. Build-time locations like this are typically not done with environment variables but with command line options instead. The "`-I`" option adds a directory to the set searched for include files.

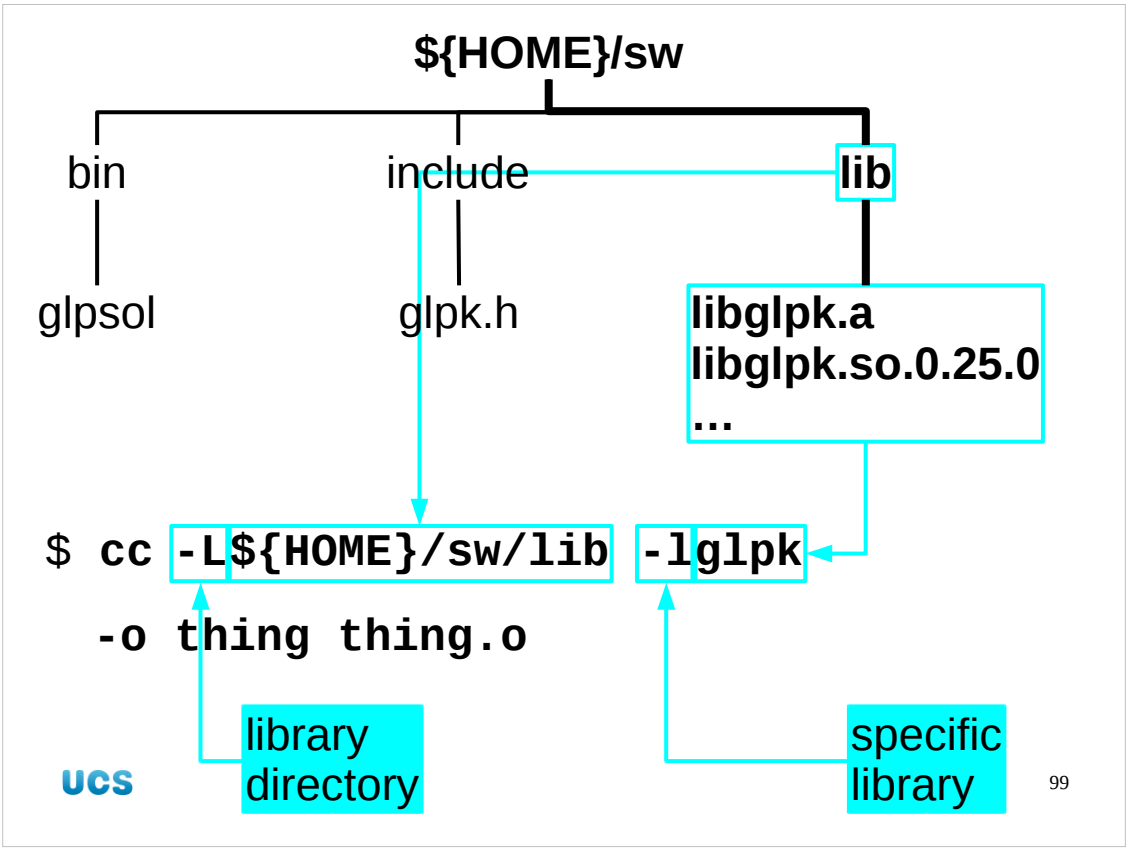
We will not be running the compiler directly, of course, but will use `make` instead.



In practice we set the `CPPFLAGS` macro in our `Makefile`. Note how we re-use the `PREFIX` macro to save a little bit of repetition.



Now we will look at the libraries themselves.
We need to get these linked into any executable that will use them
Note that this package does not provide a `pkg-config` file!



THE COMMAND IN THE SLIDE IS ILLUSTRATIVE. DO NOT RUN IT!

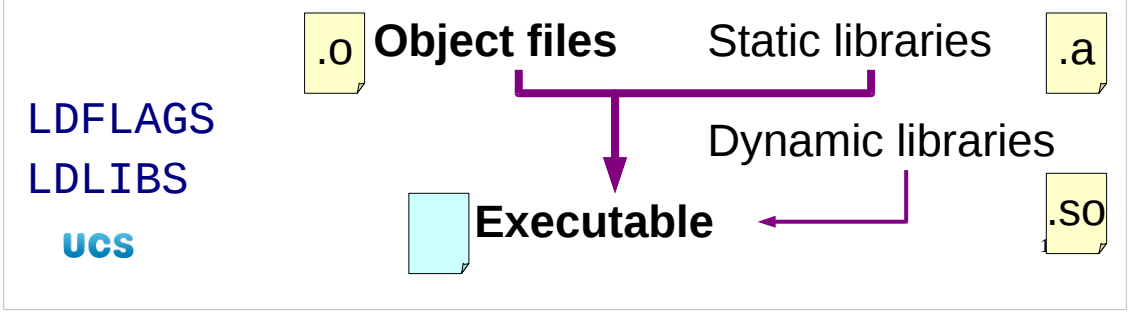
If we were linking with the compiler then we would need to add some more options to our command line. We need two options, one of which we have met before. They differ only by case, so you need to be careful with your typing.

The "`-L`" option adds a directory to the set searched for libraries. It is analogous to "`-I`" for include files.

The lowercase "`-l`" option is the one we have seen already. This identifies the libraries we need.

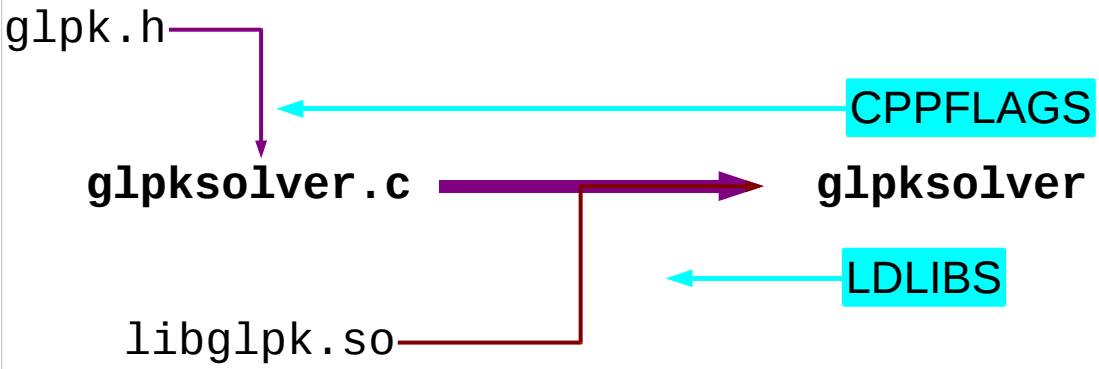
Of course, we are not going to run this command manually. We are going to use `make`.

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

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



To link in the GLPK libraries we modify the `LDLIBS` macro in the `Makefile` we have. We are keeping the default behaviour of linking dynamically so we don't need `LDFLAGS`.

Exercise: Using the GLPK



UCS

So let's try to use these new libraries of ours.

There is a simple program in /ux/Lessons/Building/gplksolver.c which solves a simple linear programming problem. This will need to use our new library's header file and be linked against the library itself.

Copy in the source file to /tmp/building and adjust your Makefile to have the appropriate CPPFLAGS and LDLIBS variables as described in the previous slides.

Then build the software. You don't need to install it, but you can add those rules to the Makefile if you want.

```
$ cd /tmp/building
$ cp /ux/Lessons/Building/gplksolver.c .
$ make gplksolver
cc --ansi --pedantic -I/home/rjd4/sw/include  -march=i686
glpksolver.c  -L/home/rjd4/sw/lib -lglpk -o glpksolver
```

Program seems to run OK

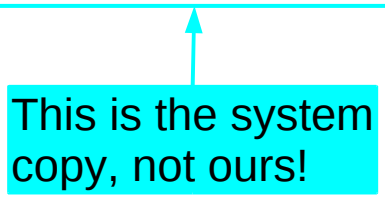
```
$ ./glpksolver
…
OPTIMAL SOLUTION FOUND

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

UCS

But we aren't quite done yet. We have successfully built the `glpksolver` executable and it seems to run. However, looks can be deceiving.

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
```



The executable is using the “wrong” library. There is a system version of the library in `/usr/lib` and despite all our build options the executable seems insistent on using it.

If the system didn't have that library at all this would have failed completely!

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

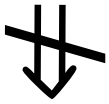
We have a newer version that we want to use.

UCS

Recall that one of the reasons we might want to build our own software is that it gets us more recent versions of the software than are installed in the system set. This is one of those cases.

We can see that there are two copies of the library. The system version is at 0.16.0 and ours is at 0.25.0. We want to use our, more recent version.

Finding libraries at build time



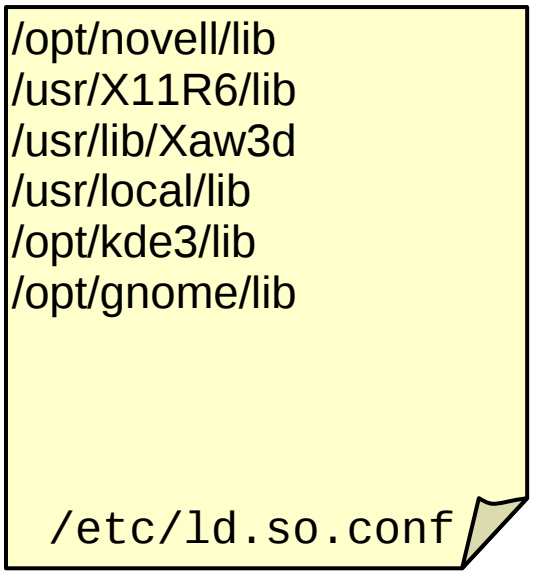
Finding libraries at run time

This tells us that being able to resolve the libraries at build time is insufficient to resolve them at run time. There are two ways round this.

One is to tell the executable (`glpksolver` in our case) at build time where the libraries will be at run time. (And note that they may not have been installed there yet at build time.) This means you will never be able to move your software tree, because its location is hardwired into its own programs. This is not a good idea in the general case as it means you cannot move libraries without having to rebuild all your executables.

The second is to tell the run time system where to look. This is the approach we will take.

Finding libraries at run time — 1



List of directories

No way to use ${HOME}

System libraries

Let's examine how libraries are located at run time.

The first place to look is in the file `/etc/ld.so.conf`. This contains a list of directories which are searched for matching file names. This has no mechanism to expand environment variables so we cannot add `${HOME}` expressions here. It is fixed across the entire system.

Finding libraries at run time — 2

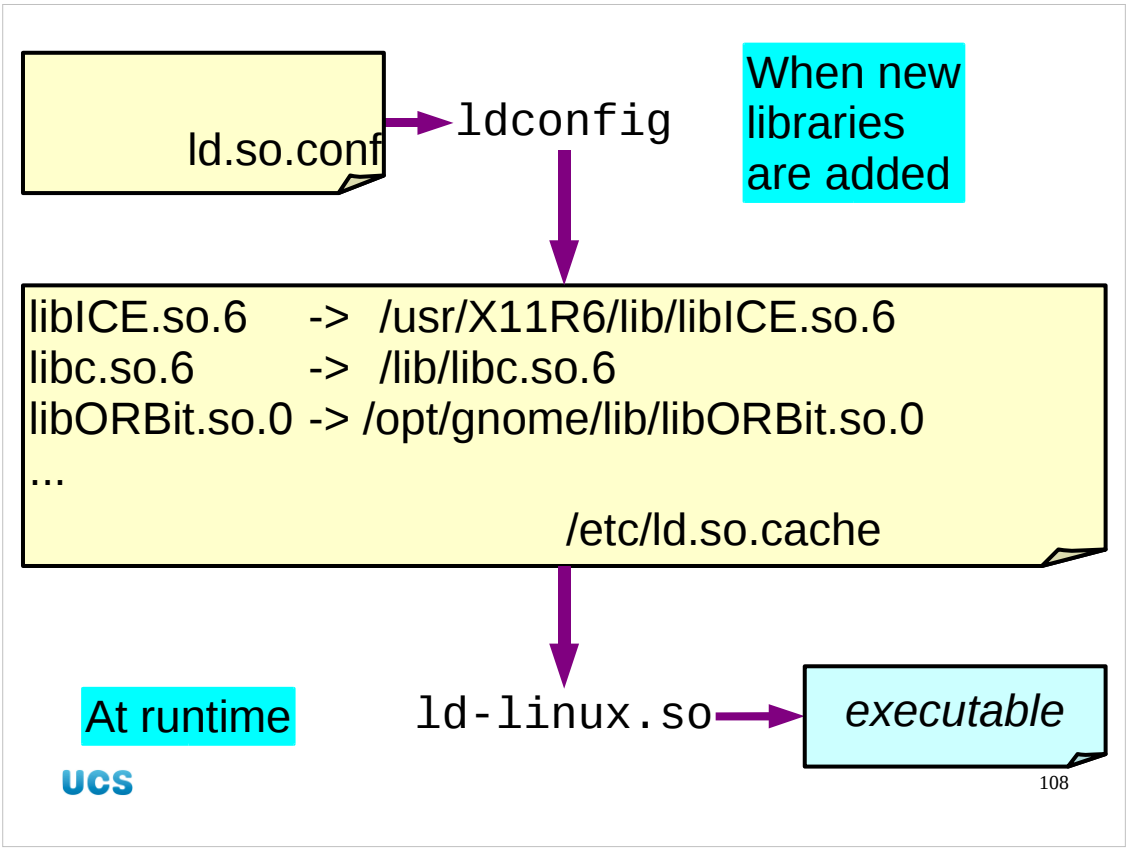
`/lib/ld-linux.so.2`

Built-in list of directories

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

Still no way to use ${HOME}

Second there are some directories wired into the run time linker itself. Again, there is no way to use `${HOME}` here.



Because the search for libraries is performed every time an application launches it needs to be very fast. To assist with this, the directories we have just mentioned are not searched individually every time a library is needed. Instead, the system maintains a (binary format) database in the file `/etc/ld.so.cache` which is a lookup from library name to file name. This has to be updated every time a system library is added or removed. At run time the linker uses this cache to set up the executable.

Personal extras

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

So how do we add our own personal libraries? We do it in the traditional fashion, with a "`*_PATH`" environment variable called `LD_LIBRARY_PATH`. This will contain a colon-delimited list of directories to be searched for libraries. It is searched before the cache is used and has no cache of its own so it should be kept short if possible. We will have only a single component on it.

Exercise: Update .bashrc

1. `bashrc3` → `${HOME}/.bashrc`

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

So we have one final update to our `${HOME}/.bashrc` file. Copy in `bashrc3` to get a start-up script that sets this extra environment variable. Note that, for the first time, we set its value absolutely rather than append our directory to a system default. The system uses a different mechanism to set libraries (`/etc/ld.so.conf`) so it should not be passing you an `LD_LIBRARY_PATH` at all.

Exercise: Update .bashrc

2. Existing terminal window:

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

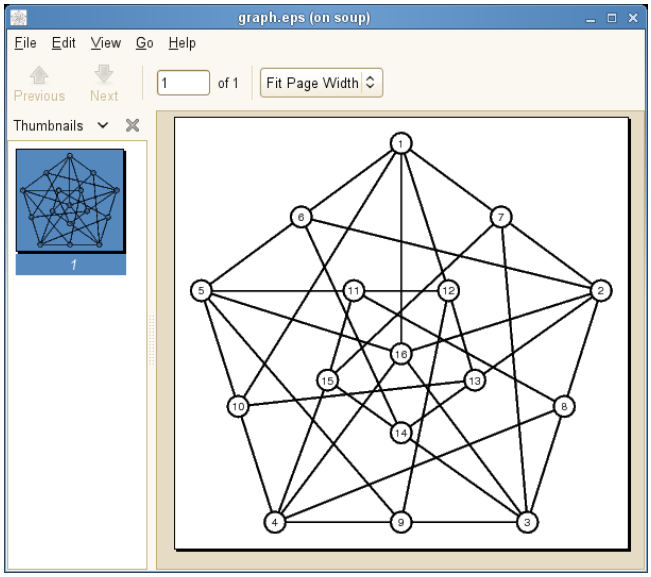
3. New terminal window:

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

Launch a new terminal window and change directory to `/tmp/building`. Then try `ldd` again. This time it will (should!) point to our copy of the library.

Viewing the graph

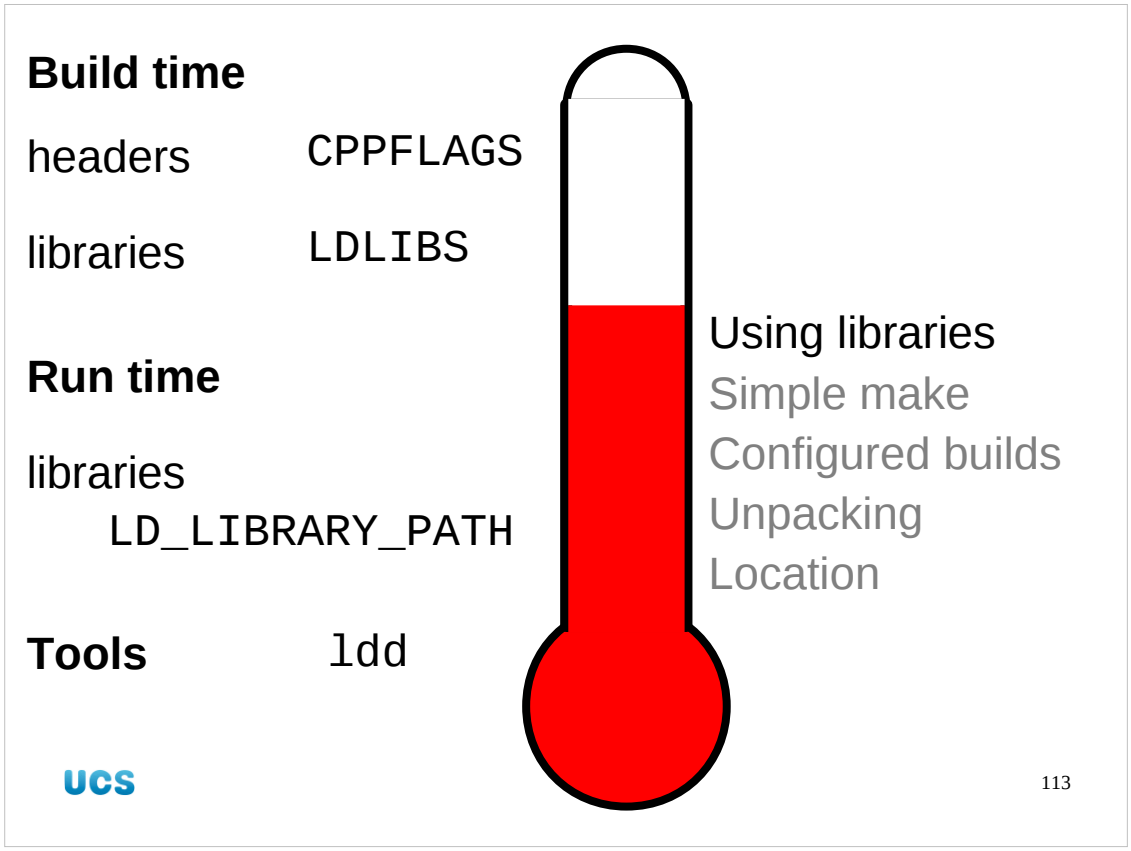
```
$ evince graph.eps
```



UCS

There's a chance that you might actually want to see the graph this program generates. Use the `evince` program to view it.

What we have used the library to do is to find an optimal path round a set of points.



And we're done for this session.

We have covered a lot of ground in the second half of the session. We have seen how to use libraries both at build time and at run time, with the critical understanding that the two are not the same. We've also used a new tool, `ldd`, which gives us insight into how an executable is using its dynamic libraries.