

Building, installing and running software

Day one

Bob Dowling
University Computing Service

www.ucs.cam.ac.uk/docs/course-notes/unix-courses

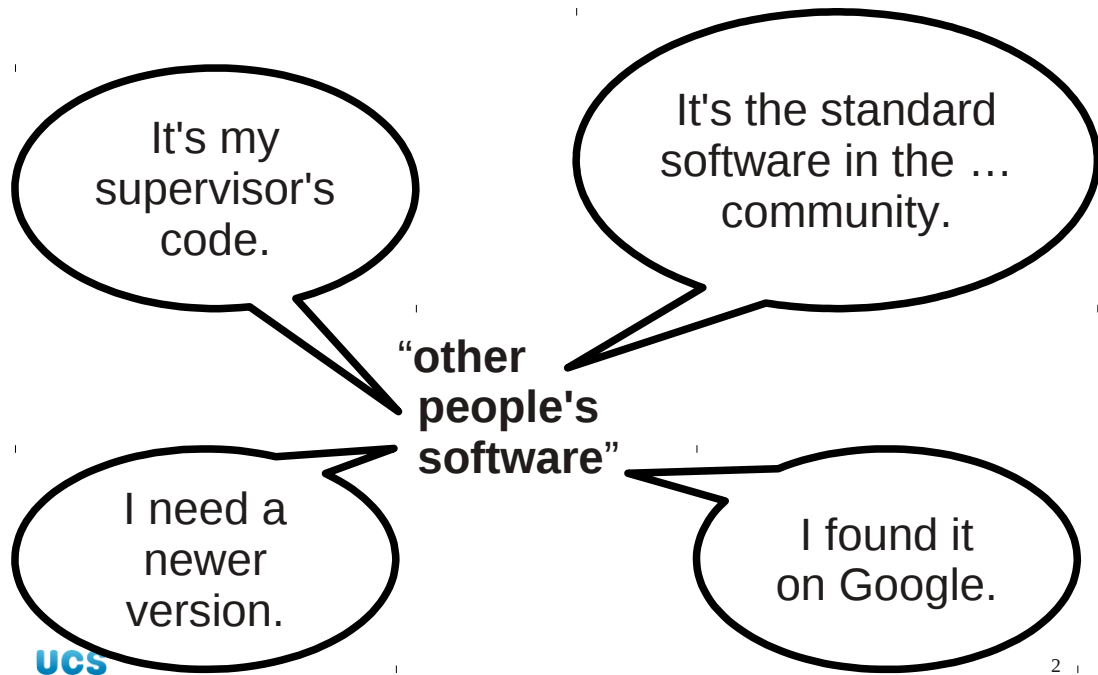
training.cam.ac.uk/ucs/



1

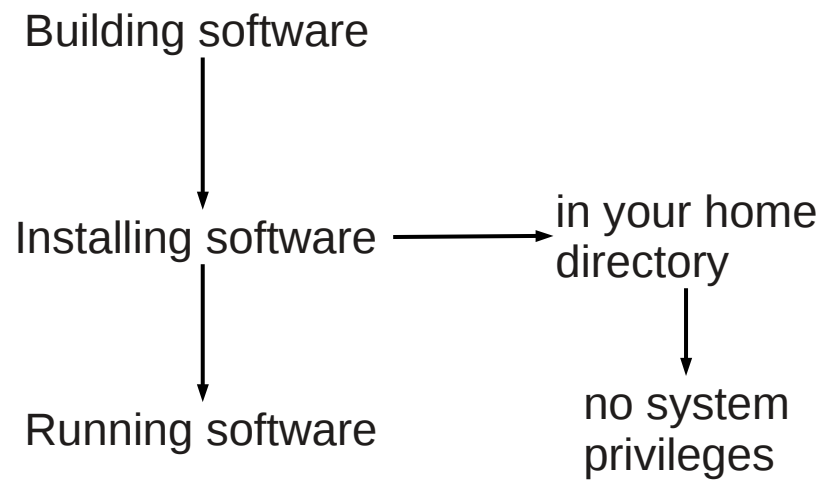
Welcome to the UCS course on “Building, installing and running software”. This course is given on the PWF Linux system though the skills it teaches should be portable to any modern Unix platform and certainly to other GNU/Linux platforms. When this course is given in classrooms students are provided with course accounts. These are not compulsory and you may use your personal account. If you are a seasoned Unix user and have your own personal `.bashrc` file you will need to copy it somewhere safe; it will be overwritten in this course.

Why do this course?



This course is designed to help you with software that's been dumped on you by your supervisor, or that you tracked down because someone told you it would be useful, or that you found when you were looking for something else.

What will you learn?



This course will give you the skills to look after that software yourself without needing to pester your system administrator, without having to have system privileges and without having to leave your home directory!

What is this course *not* for?

System administration

System directories

Writing software



This is not a course for system administrators. The whole point of this course is to avoid using system privileges, system directories or anything “systems-y”. It is also not a course on writing software; we have other courses for that. This course assumes that someone else has written the source code and you just have a copy of it. It's what happens next that we cover, not what happened before.

Course outline

Location Unpacking “Configured” builds	1 st afternoon
make and Makefiles Software libraries	2 nd afternoon
“Real world” example Recursive make	3 rd afternoon

This session will start by discussing where we will install our software and why. Then we will see how to unpack the source code and then, in the easiest and most common cases, build it. This relies on the source code coming with a script called “configure”.

The second afternoon will deal with what to do when the source code doesn't come with a configure script and will introduce “make”, the standard Unix tool for building software. It will also discuss libraries of software.

The third afternoon will take us through an example build which deviates from the good practice we introduced in the 2nd afternoon. We will cope with this and, after learning one last trick, improve it by applying those good practices.

Location?

`${HOME}/bin`

↑
programs

But what about...

documentation?

graphics?

libraries?

So let's get started.

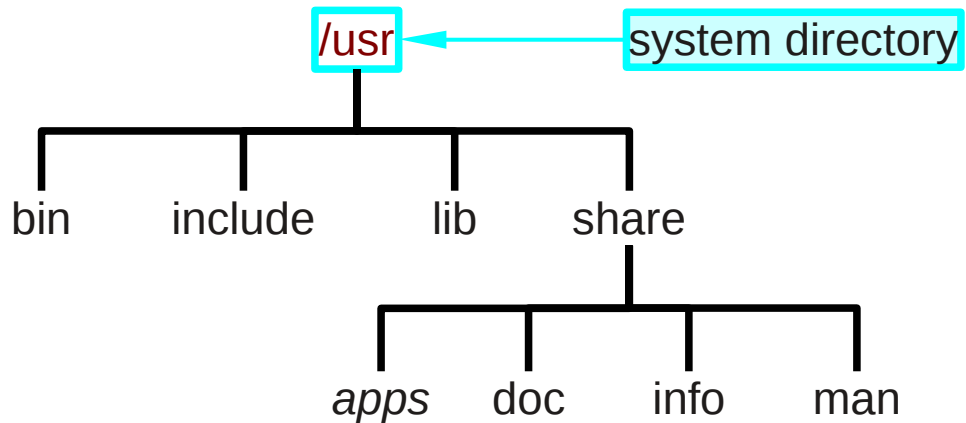
We need somewhere to put our software. This must be somewhere that we control and for which we need no extra privilege. So that will be our home directory, then.

There is a traditional place for personal software, a subdirectory of your home directory called "bin". (By the way, "bin" is short for "binaries" and not "rubbish bin".) However, modern software rarely consists of a single executable any more. Instead, it's a collection of binaries, graphics files, manual pages and other documentation, and perhaps some libraries.

"\${HOME}/bin" doesn't cut it any more.

Incidentally, HOME is an environment variable which evaluates to your home directory, so "\${HOME}" stands for your home directory. "~" is another Unix shorthand for your home directory that some programs recognise.

Mimic the system location



UCS

7

Instead of sticking everything in `/${HOME}/bin` we look at the system hierarchy under `/usr`. It contains a stack of subdirectories designed to cope with all the weird and wonderful software run on a modern computer.

For example:

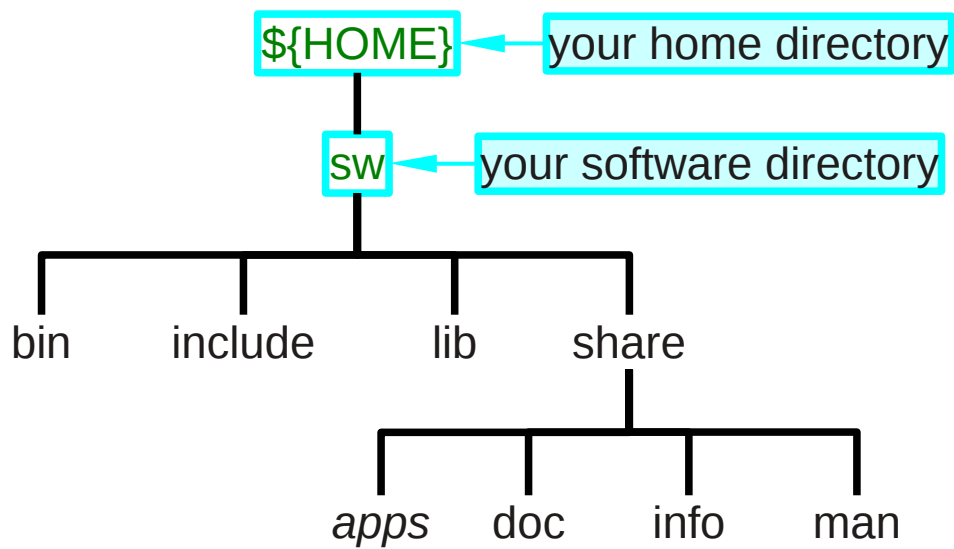
`/usr/bin` contains user executables

`/usr/include` contains the header files used to build C programs

`/usr/lib` contains the libraries

We will emulate that structure ourselves.

Mimic the system location



UCS

8

Instead of /usr we will have a directory in our home directories called “sw” (for “software”). We can call it whatever we like but we must be consistent. Some software can't cope with its location being renamed after it is built and installed. I also strongly advise against putting funny characters (including spaces) in the directory name.

Bug!

Some software needs the tree to exist before it can be installed.



We have built the tree in advance.
(And put one program in it.)

In theory, the installation of programs should create the directory structures they need. However, software is buggy and this includes its creation. Some software requires our `/${HOME}/sw` directory tree to exist prior to installing files into it. To speed things up for this course we have created an empty tree which we can use to get round this problem.

We have also taken the opportunity to put one program in the tree.

Finding programs

Environment variable: PATH

```
$ echo ${PATH}  
/usr/local/bin:/usr/bin:/bin:  
/usr/bin/X11:/usr/X11R6/bin:  
/usr/games:/opt/kde3/bin:  
/usr/lib/mit/bin:/usr/lib/mit/sbin:  
/opt/novell/iprint/bin:  
/opt/real/RealPlayer
```

First of all we need to be able to use programs that live in this directory tree. Whenever you issue a command (“ls” say), the operating system looks through a list of directories looking for an executable file of that name in that directory. This list of directories is stored in an environment variable called “PATH” as a colon-delimited list.

```
/usr/local/bin:/usr/bin:/bin:  
/usr/bin/X11:/usr/X11R6/bin:  
/usr/games:/opt/kde3/bin:  
/usr/lib/mit/bin:/usr/lib/mit/sbin:  
/opt/novell/iprint/bin:  
/opt/real/RealPlayer
```

`/usr/local/bin/ls`

X

`/usr/bin/ls`

X

`/bin/ls`

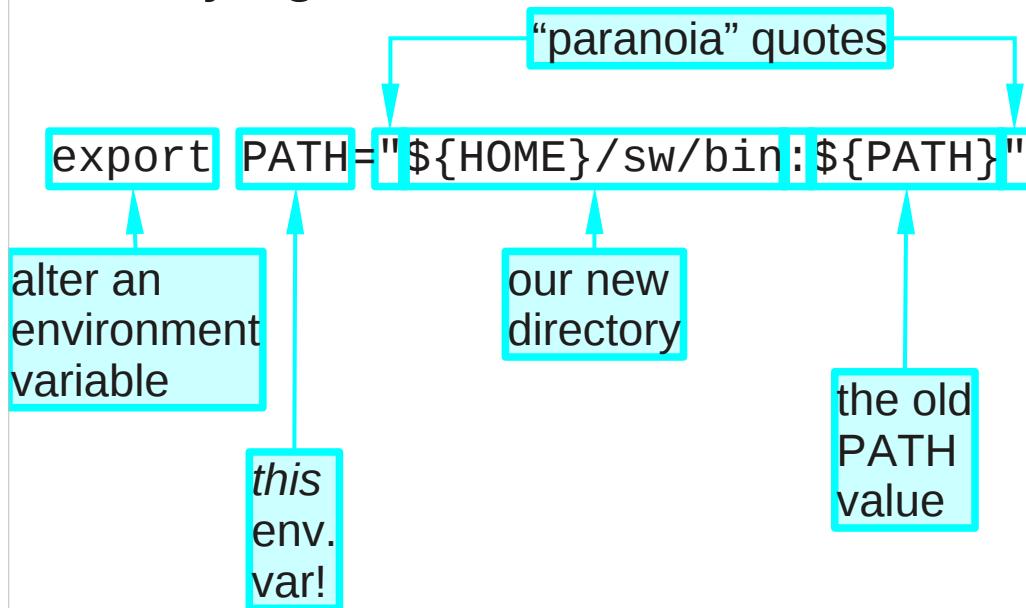
✓

UCS

11

So if you issue the `ls` command, the operating system looks through this list of directories looking for an executable file of that name in that directory. As soon as it finds one it executes that file.

Modifying PATH



So we need to modify the PATH from the system value by adding our directory to the list. The export command specifies that its an environment variable that's being altered. The value is a colon-delimited list and the new value we construct is our new directory, followed by a colon, followed by whatever was there before.

If you are following these notes in class, please **do not run** this command yet.

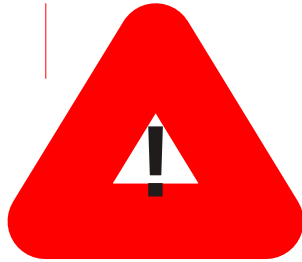
Setting PATH automatically

`${HOME}/.bashrc`

File automatically
run every time
you log in.

We will put the
command there.

NB: **Only** when
you start a session.



UCS

13

But we're lazy. We don't want to type that command every time. Instead we will use a file in our home directory called ".bashrc" (nb the leading dot). This file is run by the shell every time the shell is started. For example, each time you open a terminal window a shell is started to run in it. That shell reads this file.

Note that the file is read only at the start of a session. Changing the file has no effect on any currently running sessions. You must start a new terminal window to reap the benefits of the edit.

Not just PATH !

commands

\$ ls

PATH

\${HOME}/sw/bin

manual pages

\$ man ls

MANPATH

\${HOME}/sw/share/man

information pages

\$ info ls

INFOPATH

\${HOME}/sw/share/info

ucs

14

There is more to life than executables, though. We have already realised that we need a directory tree rather than just `${HOME}/bin` because we have more than executables to install. We need to locate these other files too.

The location of manual pages is changing. Originally they would have been installed under `/usr/man`. More recently they have started moving to `/usr/share/man`. For this course will cover our bases and look in both directories, `${HOME}/sw/man` and `${HOME}/sw/share/man`. The equivalent of PATH for manual pages is called MANPATH. We will adjust that in our `.bashrc` too.

As well as manual pages (given by “`man ls`”) there are “info pages” for the `info` command (given by “`info ls`”; press “q” to quit) and a corresponding pair of directories and an INFOPATH environment variable.

We will be adding to this list of environment variables as we encounter the need for them.

Exercise

1. Copy in a new `${HOME}/.bashrc` file.

```
${HOME}/bashrc1  
└─┬─┘ ${HOME}/.bashrc
```

2. Observe the existence of a program “hello”:

```
$ ls ${HOME}/sw/bin  
hello  
$
```

UCS

15

This is our next, still rather trivial, exercise. It is really here to get you used to needing to start a new terminal window to pick up any changes you make to your `${HOME}/.bashrc` file.

1. Please copy, from the directory `/ux/Lessons/Building`, a version of this file. Copy `bashrc1` to `.bashrc` in your home directory.

```
$ cd  
$ cp /ux/Lessons/Building/bashrc1 .bashrc
```

Note that the file is having its name changed in the process of this copy. It starts out as “`bashrc1`” (a “1” at the end and no leading dot) and ends up as “`.bashrc`” (no trailing “1” and with a leading dot.) Unix starts most of its configuration files with a dot. This means that they won’t appear in a normal “`ls`” output.

2. Also observe that we have slipped in a program into your “sw” directory:

```
$ ls -l sw/bin  
total 10  
-rwxr-xr-x 1 y550 y550 9694 2011-03-17 12:36 hello
```

We will try to run this program next.

Exercise

3. In your existing terminal window...

```
$ hello
```

```
-bash: hello: command not found
```

4. Launch and use a new terminal window...

```
$ hello
```

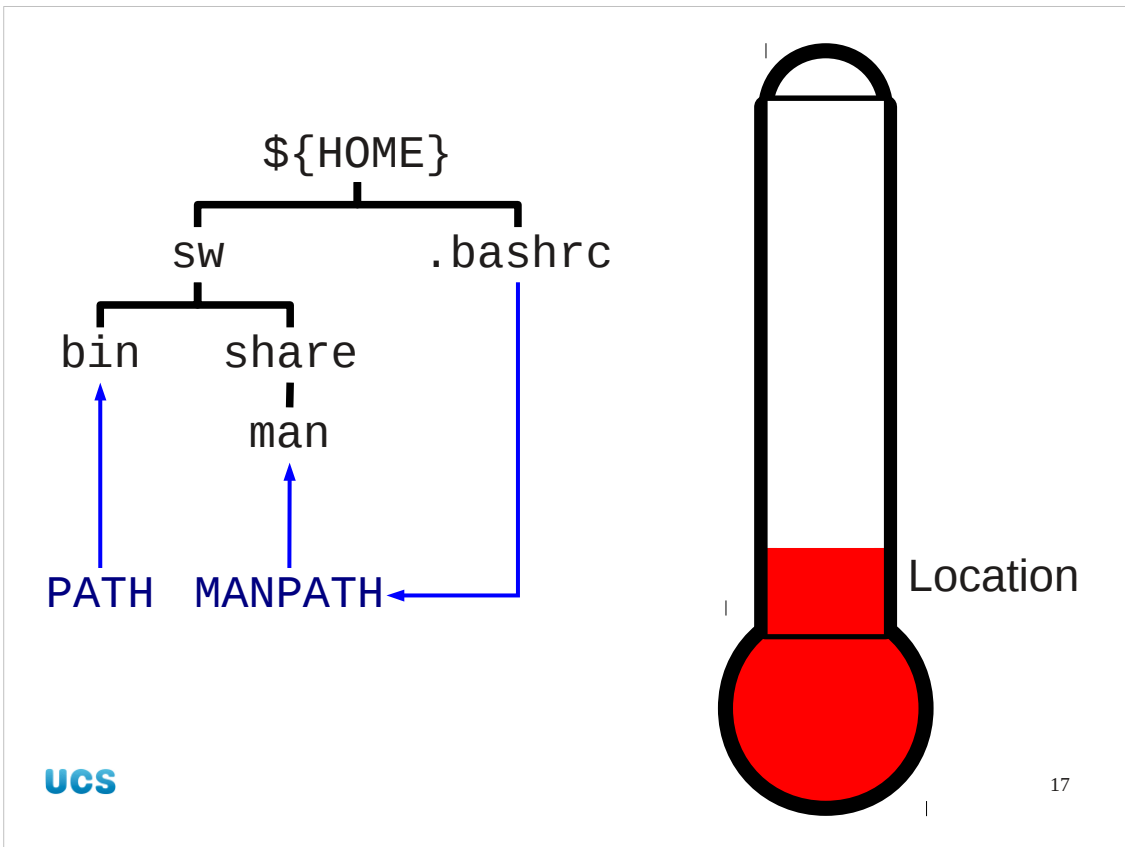
```
Hello, world!
```

5. Close the old terminal window.

3. Then, from the same window as you ran the copy commands, run the command “hello”. This command should fail. Your PATH has not yet been altered by the arrival of the `/${HOME}/.bashrc` file.

4. Next, launch a *new* terminal window. This is a fresh shell so will read the new `.bashrc` file and have the modified PATH. Run “hello” in that second terminal window and it should work.

5. Then close down the first window. We will be working exclusively in an environment with the enhanced PATH from now on.



And that's stage one done!

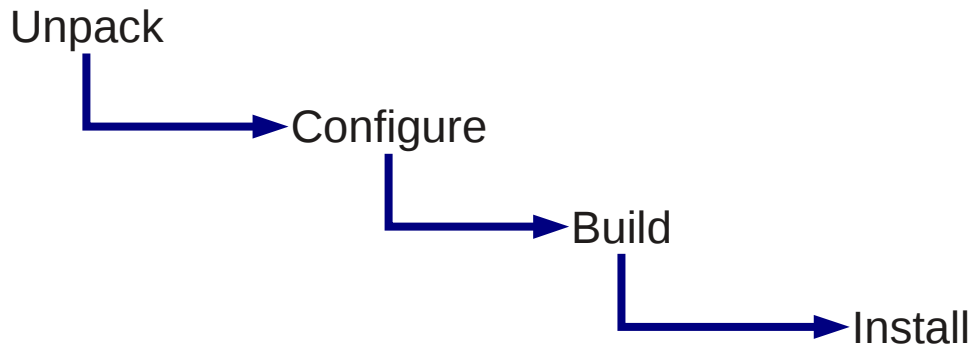
We have somewhere to put our software. What's more we can use it once it gets there by setting environment variables in our `.bashrc` file.

Any questions?

We have a location...

...so let's build
something to
put in it!

The classic model

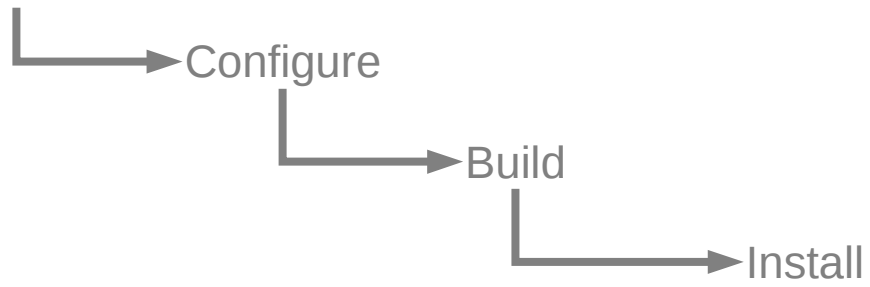


What we will cover for the remainder of this afternoon is the classic software model of “unpack, configure, build, install”. This is by far the simplest way of managing software and is also by far the most common for software managed on the Internet. I don't think I've ever downloaded software from Source Forge (<http://www.sourceforge.net/>, the largest on-line development system for free software), for example, that didn't follow this pattern.

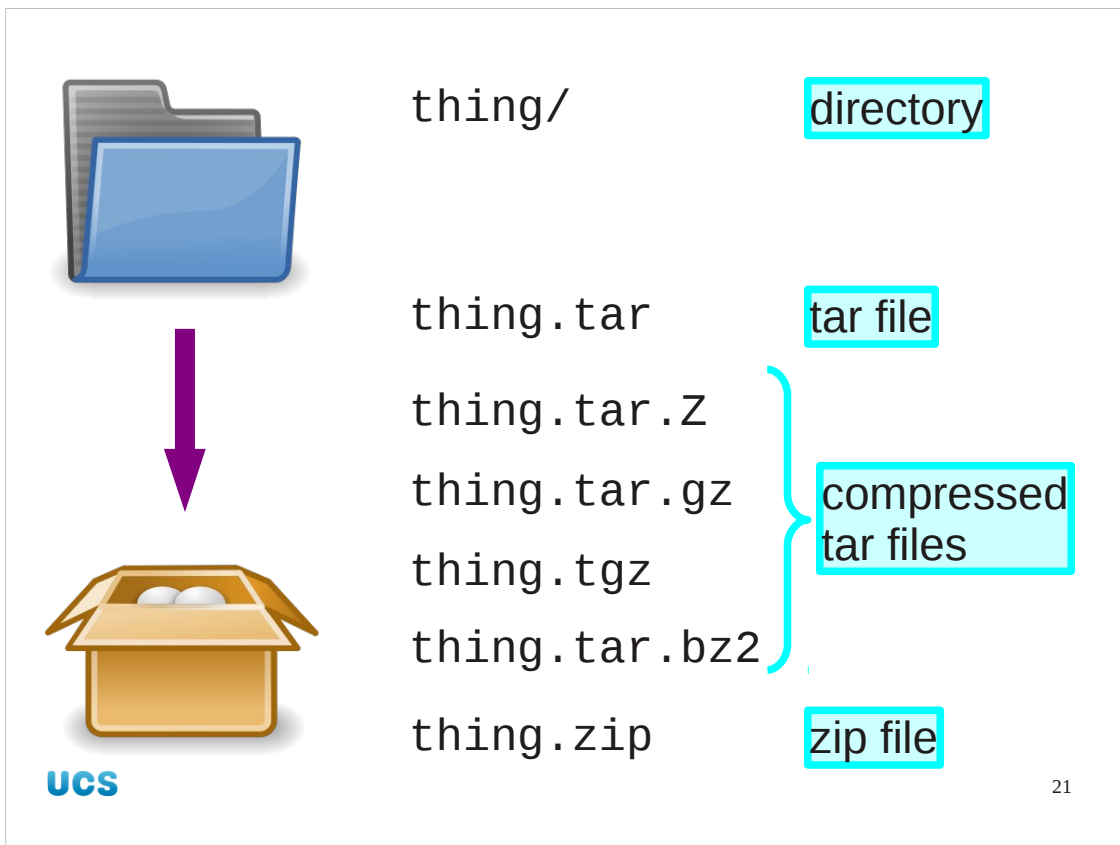
It was designed by the Free Software Foundation as part of their GNU project. While everyone is familiar with GNU/Linux it is important to remember that the GNU project designs their tools to work on a very wide range of platforms. For this to be feasible they developed configuration tools to adapt software source code for specific platforms.

It's not simple to write these configuration scripts, but it is very easy to use them and this is a course about using them, not writing them.

Unpack



The software typically arrives as an archive file which we need to unpack into the directory tree of source files.

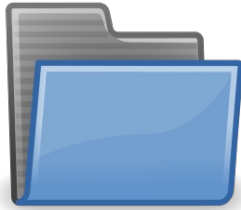


There are typically many files in a source code distribution. Because that is awkward to transfer from site to site these source files are typically bundled together in a directory (or tree of directories and sub-directories) and then converted into a single “archive” file.

The traditional Unix archiving tool is called “tar” (“**t**ape **a**rchive”) originally used for placing a directory tree onto a tape. The files it creates traditionally end in the suffix “.tar”. The tar command does no compression of its own so that the files it creates are typically compressed by a separate compression program. If the old “compress” program is used then the suffix “.Z” is added to the end. If the more common “gzip” program is used then the suffix “.gz” is added. The combination “.tar.gz” is sometimes replaced by “.tgz”. The most recent compressor to be used is the “bzip2” command and it has a traditional suffix of “.bz2”.

Alternatively there is the combined archival and compression program “zip” imported from the PC world. Its archive files need no further compression and typically end with the suffix “.zip”.

tar: unpacking



```
tar -x -f thing.tgz
```

extract

from
file

file
name

To unpack a tar archive, the tar command is used with the option “-x” to extract the files. The archive is unmodified by this action. We identify the archive file to be unpacked with the “-f” option followed by the archive file's name. Modern versions of tar cope automatically with compressed files.

tar: examining



```
thing/  
thing/foo.c  
thing/foo.h  
thing/bar.c  
thing/main.c
```

UCS

```
tar -t -f thing.tgz
```

↑
table of
contents

↑
from
file

↑
file
name

You can also see what's included in an archive without unpacking it. Replacing the “-x” (extract) with “-t” (table of contents) simply lists the files that would be unpacked if you asked it to.

zip: unpacking



```
unzip thing.zip
```

file
name

Zip files are even easier. The command “unzip” unpacks a zip archive.

zip: examining



```
thing/  
thing/foo.c  
thing/foo.h  
thing/bar.c  
thing/main.c
```

UCS

```
unzip -t thing.zip
```

testing

file
name

25

The `zip` command also has a “-t” option to examine the contents of a zip archive. In this case the “-t” stands for “testing” as this is the option to make sure a zip archive is intact. It generates a file listing as a side effect.

Worked example

1. prep

```
$ mkdir /tmp/building
```

```
$ cd /tmp/building
```

```
$ cp ${HOME}/xdalicklock-2.20.tar.bz2 /tmp/building
```

```
$ ls  
xdalicklock-2.20.tar.bz2
```

UCS

26

So let's unpack a file.

We will be running with this example through the whole process so it's important that you do this too. To save load on the Novell server providing the home directories we are going to use use local, temporary files for our building (and finally install them into the home directories managed off the Novell server).

So we are going to build a directory called `/tmp/building` and work in that. We will copy into it the software source archive we need. These have been dumped in your home directory, as they might have been if you had downloaded them from the Internet.

Please note the bent arrow convention we use in the slides to indicate that a line is meant to be a complete, single line and that it has just run on to a line below because the screen isn't wide enough.

Worked example

2. unpacking

```
$ tar -x -f xdaliclock-2.20.tar.bz2
```

```
$ ls
```

```
xdaliclock-2.20  xdaliclock-2.20.tar.bz2
```

We use the tar command with its “-x” option to extract the files from the archive. Note that tar is silent as it unpacks unless told otherwise. (It has a “-v” for “verbose” option to list files as it unpacks them if you want to see them.)

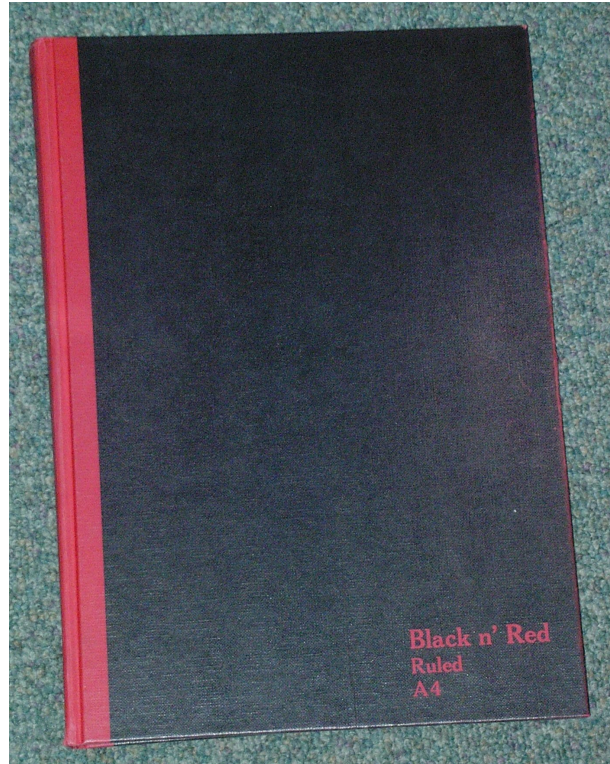
Keep records

Software:
version, source...

Details:
platform, options...

Results:
Success / Failure ?

UCS



We've started on a worked example.

Before we go further I will introduce you to a concept that might be a bit alien. Building and using the software for your scientific research is equivalent to building a laboratory instrument or to running an experiment. And it should be treated as such. Just as you record your scientific progress in a lab book you should also record your software progress.

You should record where you got the software from and what version it is. You should also record the system you use to build it. Later we will see how you pass options to the build process. You should record these too. Finally, you should record whether the build succeeded or failed and if it failed why it failed.

You are issued with toy lab books for this course. Please fill them in for all the worked examples and exercises during this course. Filling in these lab books and presenting them at the end of the third afternoon is required for anyone seeking signed "transferable skills" forms.

Worked example

3. lab book

10th March 2008

xdaliclock v2.20

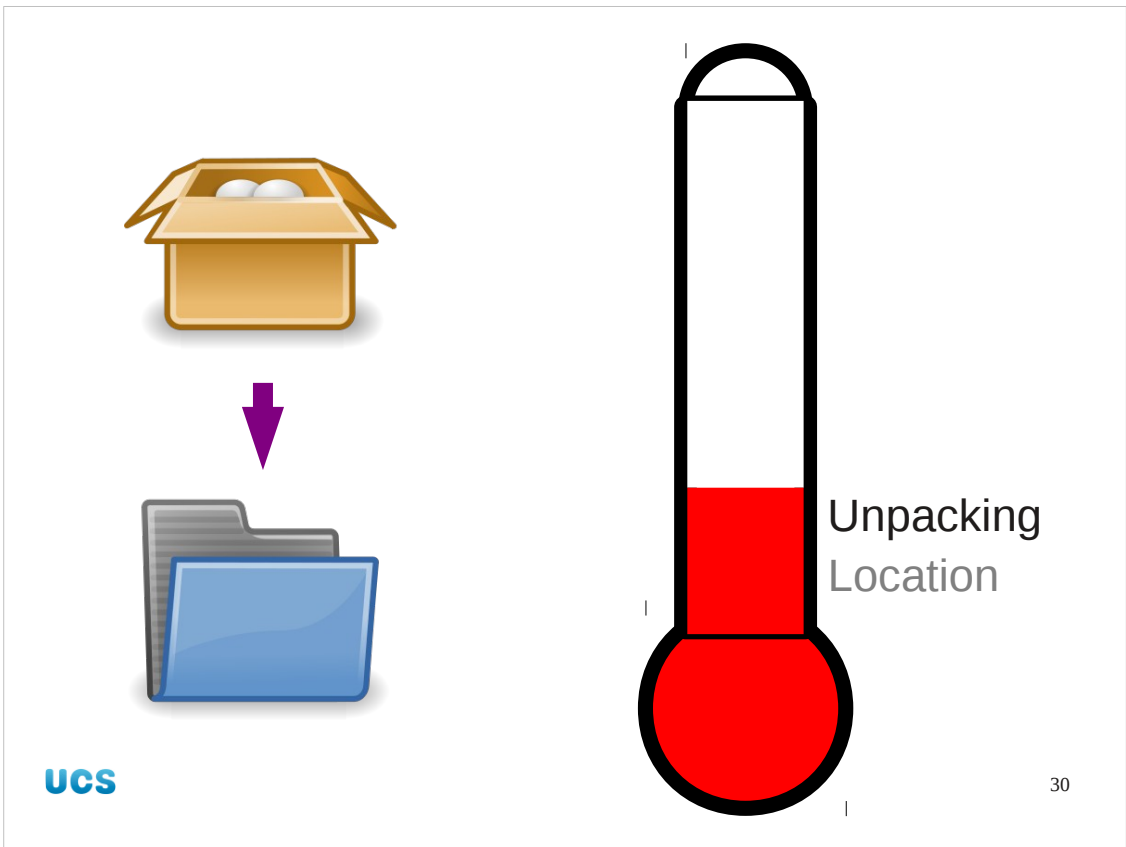
Source: UCS PWF Linux
xdaliclock-2.20.tar.bz2

Unpacks OK (tar -xf ...)

UCS

29

You don't need to go overboard. A few simple notes are sufficient.



So now we have learnt how to unpack our software source code from the files we download.

Coffee break

Five minutes break

Spines
Wrists
Eyes

Brains!

UCS



Generally speaking you should take regular breaks when you work at a computer. You need to let your spines unslouch, your wrists relax, your eyes refocus and your brains get some well-earned caffeine — I mean rest — once in a while.

If you work long periods at a computer it is better to have a one minute break every ten minutes than a ten minute break every hour, but that won't work in a course setting.

The README file

```
...  
To build for the X Window System:  
  
    cd xdaliclock/X11/  
    ./configure  
    make  
    make install  
  
...
```

Within the unpacked directory is a file called README. Sometimes it is called README.txt or comes as a set of platform-specific files: README.irix, README.linux, README.solaris, etc.

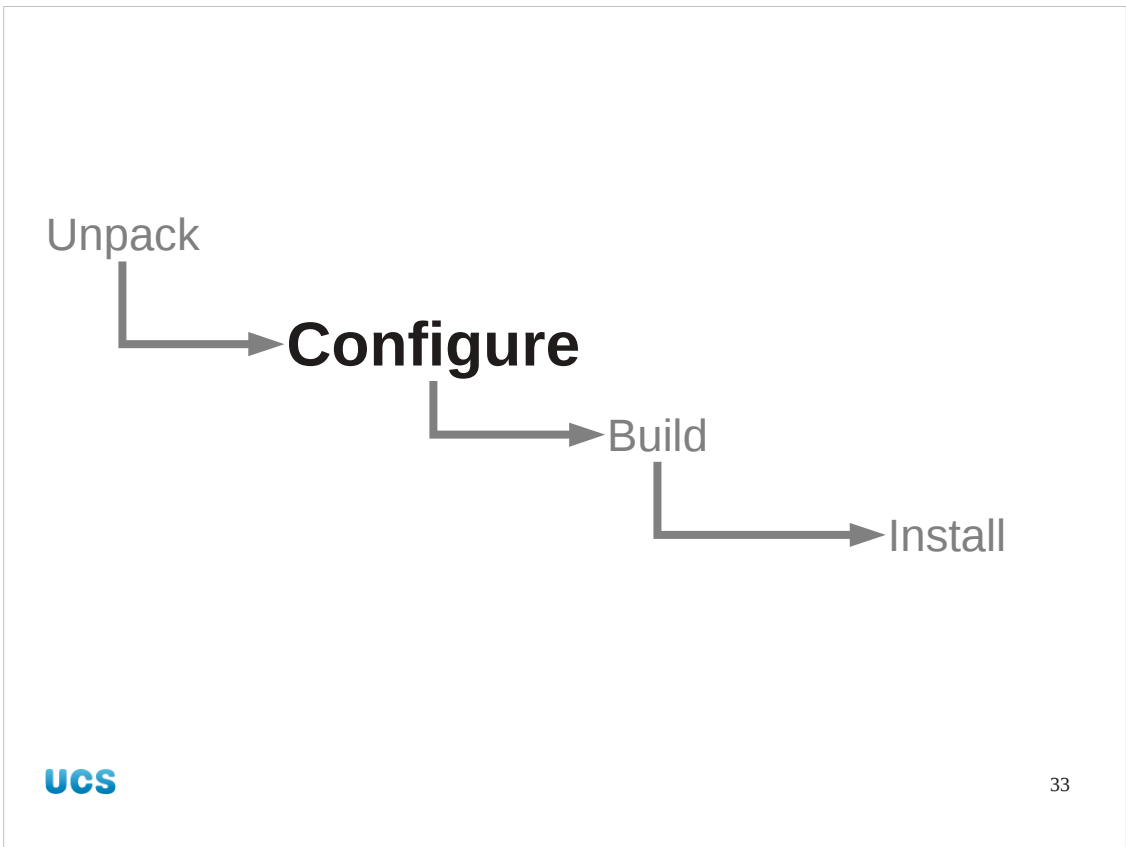
This file — by one name or another — is almost always there. If it isn't there go looking for it.

We're not kidding about this file's name: *read it*.

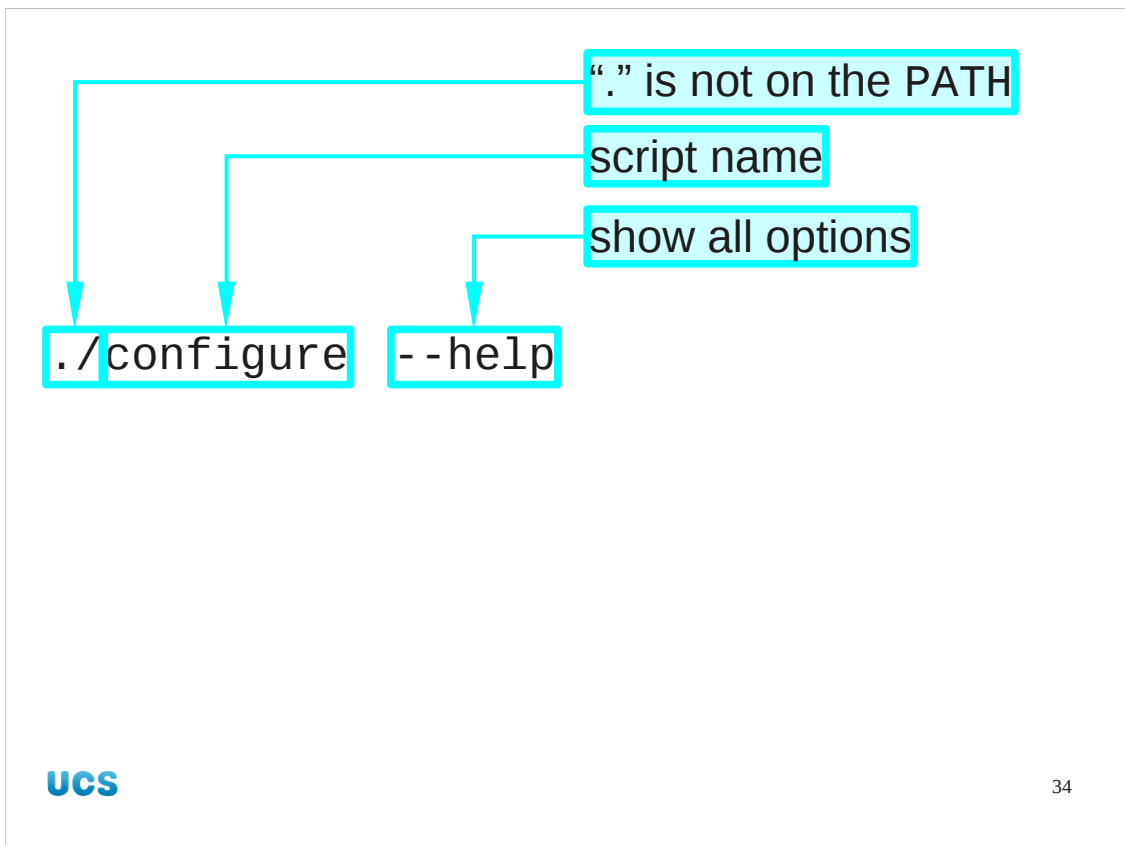
In the xdaliclock README file we find instructions for how to build it. Specifically, we find the instruction telling us what subdirectory to go to first.

Sometimes the useful file is called something else. A common alternative is "INSTALL".

Note that the README's instructions start with running the configure script. It's time to move to the second stage of our build process.



So, we've unpacked our source file. Next we have to configure it for our particular platform. (It is possible to cross-configure but it's much harder. In this course we will show how to configure a build for the system configure is being run on.)

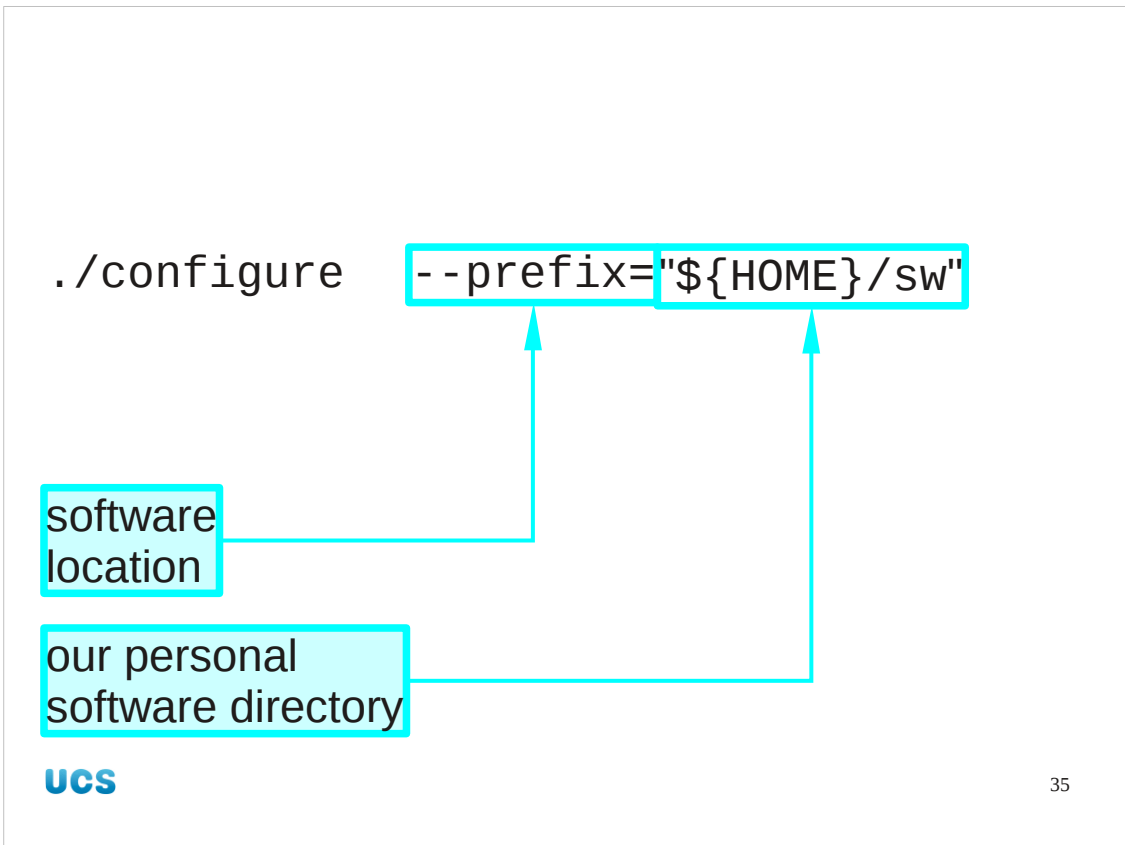


So let's look at `configure`.

Note that because `“.”`, the current working directory or the directory you are currently in, is not on the `PATH` it is not searched for commands by default. So simply typing `“configure”` as a command won't work. Instead, we type `“./configure”` to tell the system explicitly where to find the command.

The `configure` program is a large, complex shell script written in a way that makes it extremely portable across all flavours of Unix but which also makes it very hard to read. Fortunately we don't need to read it.

It has a very large collection of command line options of which only two are of direct interest to us right now. It has an option `“--help”` which lists all its various options. Fortunately for us we usually need only one.



The “- -prefix” option specifies the software tree that the software should be built for and ultimately installed in. By default, `configure` builds software for `/usr/local`. If you are building system software it might be set to `/usr`. But we are building for ourselves so we set this option to the top of our personal directory tree, `HOME/sw`. The quotes around the value are just a safety precaution in case `HOME` (or something you chose to use instead of “sw”) has strange characters or spaces in it. It's unlikely, but it's a good habit to get into.

Compiler choice

`./configure --prefix=... CC=gcc`

specify
C compiler

Use the gcc
C compiler

CC=gcc

The final set of options, that is occasionally useful specify the compilers to use and the options to pass to them. Typically these can be left blank, but if you are building the software for yourself because you have a spiffy compiler on your system then this is how you specify that the build system should use it.

Compiler options

CC	C compiler
CFLAGS	C compiler options
CXX	C++ compiler
CXXFLAGS	C++ compiler options
FC	Fortran compiler
FFLAGS	Fortran compiler options
LDFLAGS	Library options

We will be meeting these variables in depth tomorrow when we discuss the `make` utility in its own right, but for now we just need to understand that we can modify what `configure` is setting up by setting these values.

Worked example

4. configuration

```
$ cd /tmp/building/xdaliclock-2.20
```

You are probably
here already

```
$ cd X11
```

README's instructions

Configure for
our location

```
$ ./configure --prefix="${HOME}/sw"
```

UCS

38

So let's get on with our worked example of xdaliclock.

If you have been following along then you are probably already in `/tmp/building/xdaliclock-2.20`. The README there told us to move into the `X11` subdirectory. Then we run `configure`. We use its “`--prefix`” option only and set it to the software tree we are creating for ourselves.

```
$ ./configure --prefix="${HOME}/sw"
```

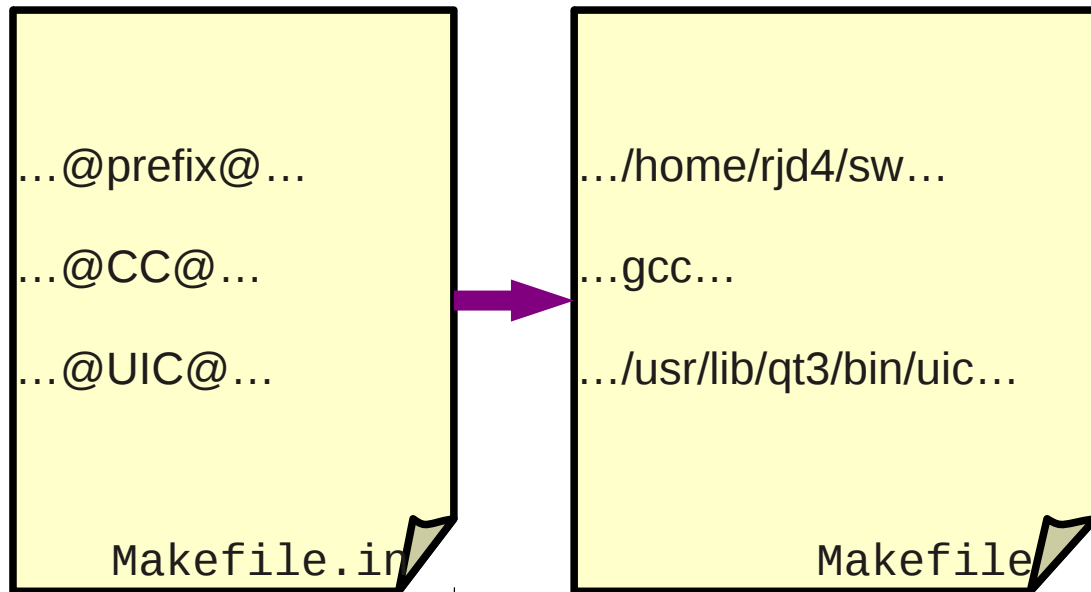
```
current directory: /tmp/building/xdaliclock-2.20/X11
command line was: ./configure --prefix=/home/rjd4/sw
checking build system type... i686-pc-linux-gnu
```

```
...
```

```
checking for remove... yes
checking for shmat... yes
checking for IceConnectionNumber in -lICE... yes
checking X11/extensions/shape.h usability... yes
checking X11/extensions/shape.h presence... yes
checking for X11/extensions/shape.h... yes
configure: creating ./config.status
config.status: creating Makefile
config.status: creating config.h
```

```
>
```

What configure does



So what is it that configure actually does?

The configure script takes a bunch of files called *thing.in* and creates files called *thing*. A list of these was given at the end of configure's output. For *xdaliclock* two files were involved:

Makefile.in → Makefile
config.h.in → config.h

Each of the *thing.in* files contains a number of *@word@* expressions. These get substituted for various values.

@prefix@ is replaced by whatever was specified by `--prefix` on the command line.

@CC@ is replaced by the name of a C compiler. By default this is "cc" (which on a Linux system typically points to gcc) but can be overridden by setting the CC environment variable as discussed earlier.

Others, such as *@UIC@* depend on the system itself. Unless a setting is specified on the command line, the configure script will go looking for the *uic* program and replace the *@UIC@* expression with whatever it finds.

Sometimes configure can't find things as we will see later. In these cases it stops with an error message saying so if it needs the facility. If the facility is optional and the package simply comes with extra bells and whistles if the dependency can be found then a warning is generated and configure sets up the build not to use that extra feature.

Worked example

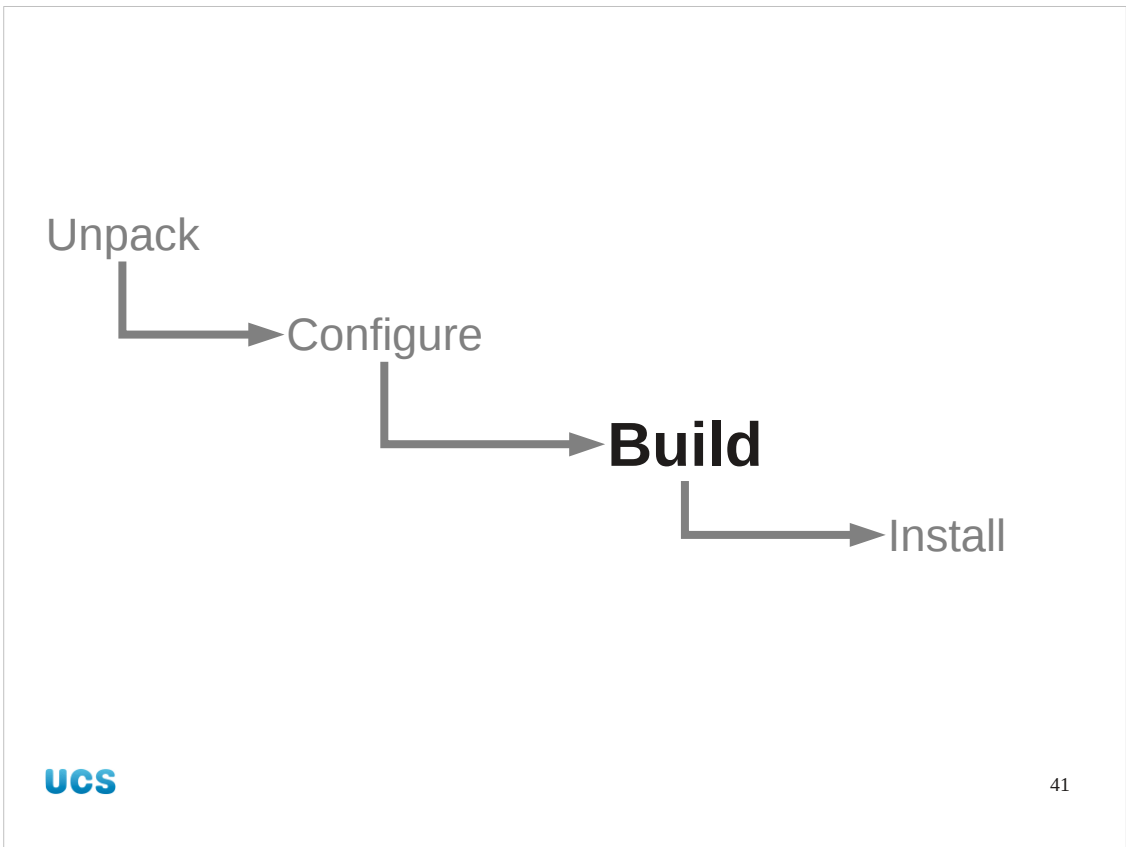
5. lab book

Source: UCS PWF Linux
/ux/Lessons/Building/xdaliclock-2.20.tar.bz2

Unpacks OK (tar -xf ...)

./configure --prefix="\${HOME}/sw"
Configures OK.

All the modification of the software is done by `configure`, either by command line option or environment variable. Record them in your lab book and follow them with a report on how `configure` behaved. In the case of success that's really all you need record.



So now we have configured our software we need to actually build it.

make

fubar.c  fubar.o

If:

exists

missing

or:

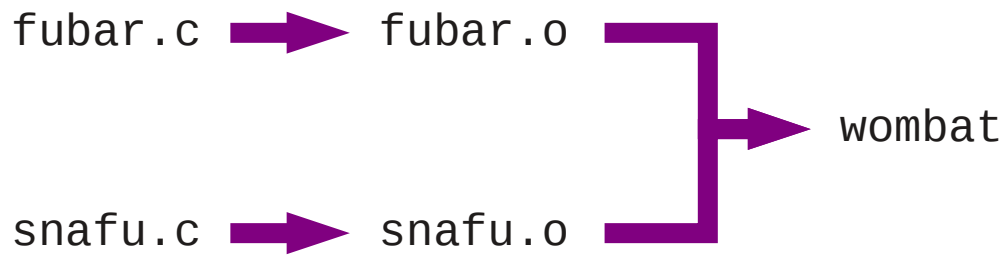
newer

older

The actual building of our program will be done by the make program. This program will fill our next two afternoons so I will skip over it lightly now.

The make program knows how to build one file (fubar.o, say) from another (fubar.c, say). If fubar.o exists and was created or updated more recently than fubar.c then make leaves it alone. Under all other circumstances make will rebuild it from fubar.c.

make

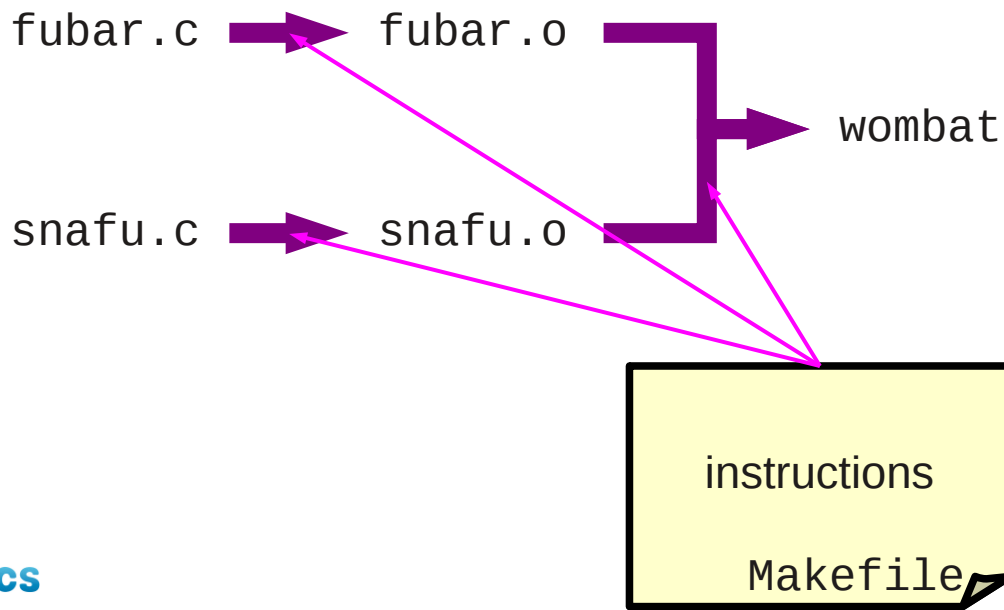


The make program takes this idea further into chains of dependencies.

Suppose the program `wombat` was built from two files called `fubar.o` and `snafu.o` and that these were built from `fubar.c` and `snafu.c` respectively. Suppose further that we start with `fubar.c` and `snafu.c` and that `make` is asked to build `wombat`.

It knows that `wombat` needs `fubar.o` and `snafu.o` and that they don't exist. So it builds them. When it's asked to build `fubar.o` it knows how to because `fubar.c` exists so it builds `fubar.o`. In the same way it builds `snafu.o`. Now that `fubar.o` and `snafu.o` exist it can finally build `wombat`.

make



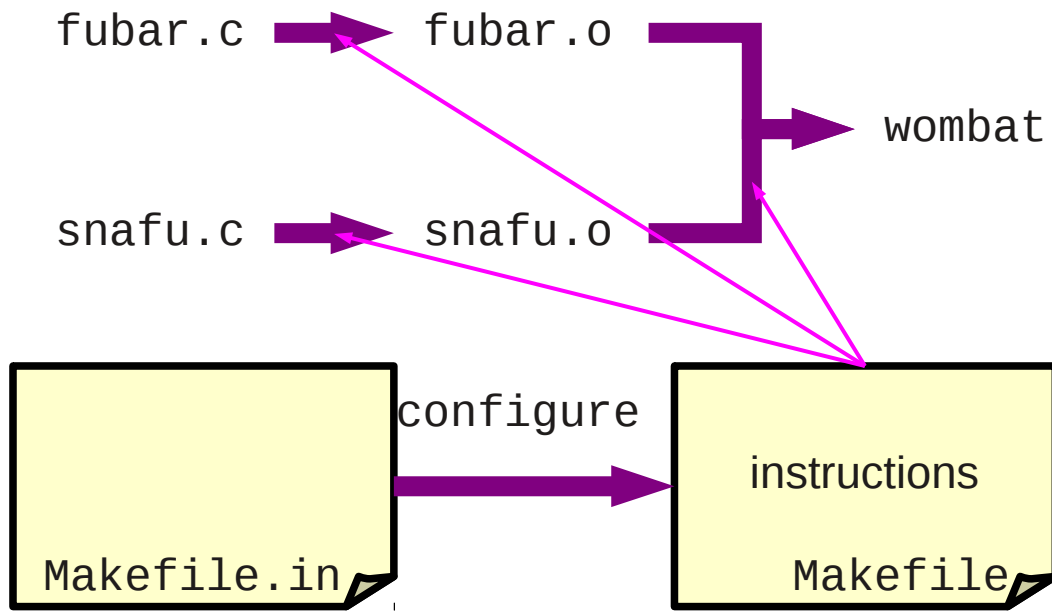
UCS

But where do the instructions come from? They reside in two places.

The default instructions for the simple but common cases are built into make itself. For example the rule to get from fubar.c to fubar.o and from snafu.c to snafu.o is built in as the general rule for something.c → something.o.

The more complex rules such as how to link fubar.o and snafu.o together to create wombat need explicitly stating and that is done in the make configuration file called Makefile (with an upper case "M").

make



But where do the instructions in the `Makefile` come from? They come from the template instructions in `Makefile.in` which were filled in with local knowledge and our location instructions by `configure`.

```
$ make
```

Given that all the instructions have been encoded in the `Makefile`, we don't need to give any extra instructions to the command itself. All we type is the command itself: `make`.

Worked example

6. make

```
config.status: creating Makefile
config.status: creating config.h
```

```
$ make
```

```
gcc -Wall -Wstrict-prototypes
-Wnested-externs -std=c89
-U__STRICT_ANSI__ -c -I. -I. -I./..
-I/home/rjd4/sw/include -DHAVE_CONFIG_H
-g -O2 xdaliclock.c
```

```
...
```

UCS

47

Back at our worked example, we recall that at the very end of configure's output it confirmed that it had created a `Makefile`. This contains everything make needs to do its job so we can just type `make`. Note that make echoes all the commands it is running to perform the build. You do not need to understand these commands, and you are not expected to.

Worked example

7. confirmation

```
$ ls -l xdaliclock
-rwxr-xr-x    ...    xdaliclock
```

Obviously we should check the build succeeded. It did for me.
If it didn't for you, shout!

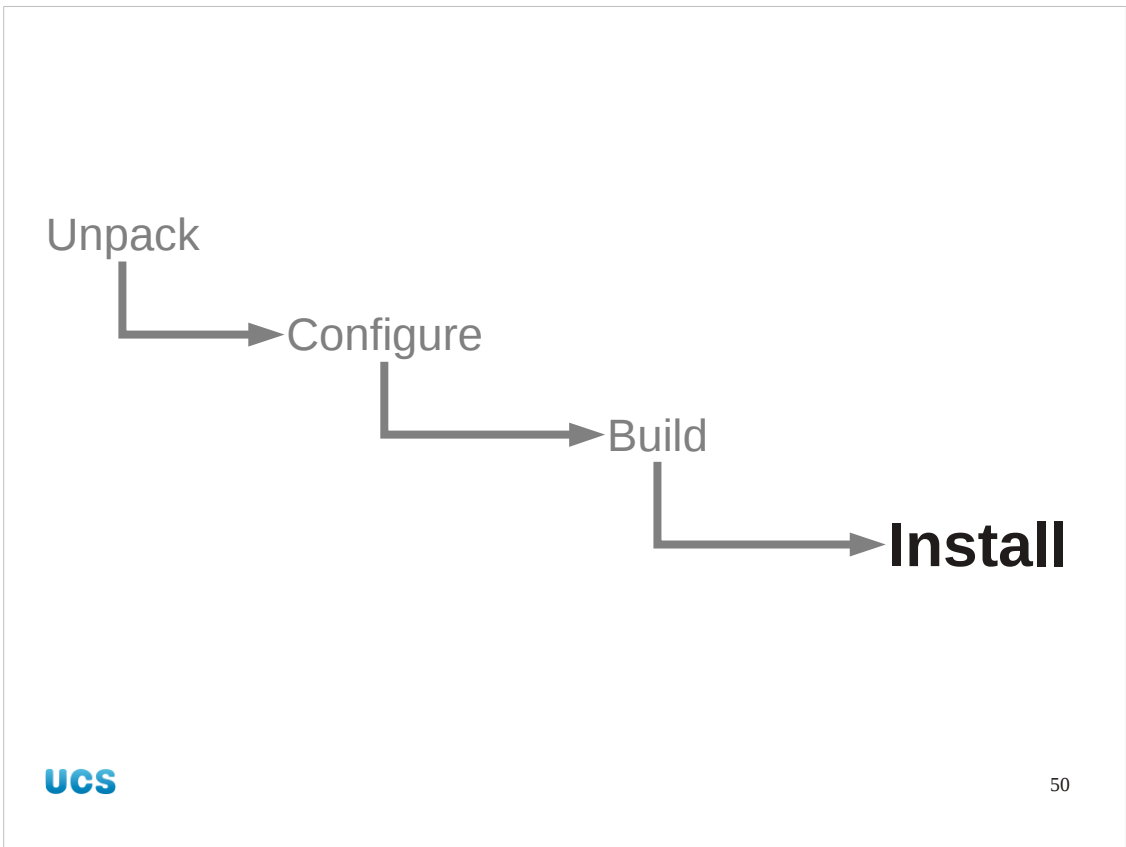
Worked example

8. lab book

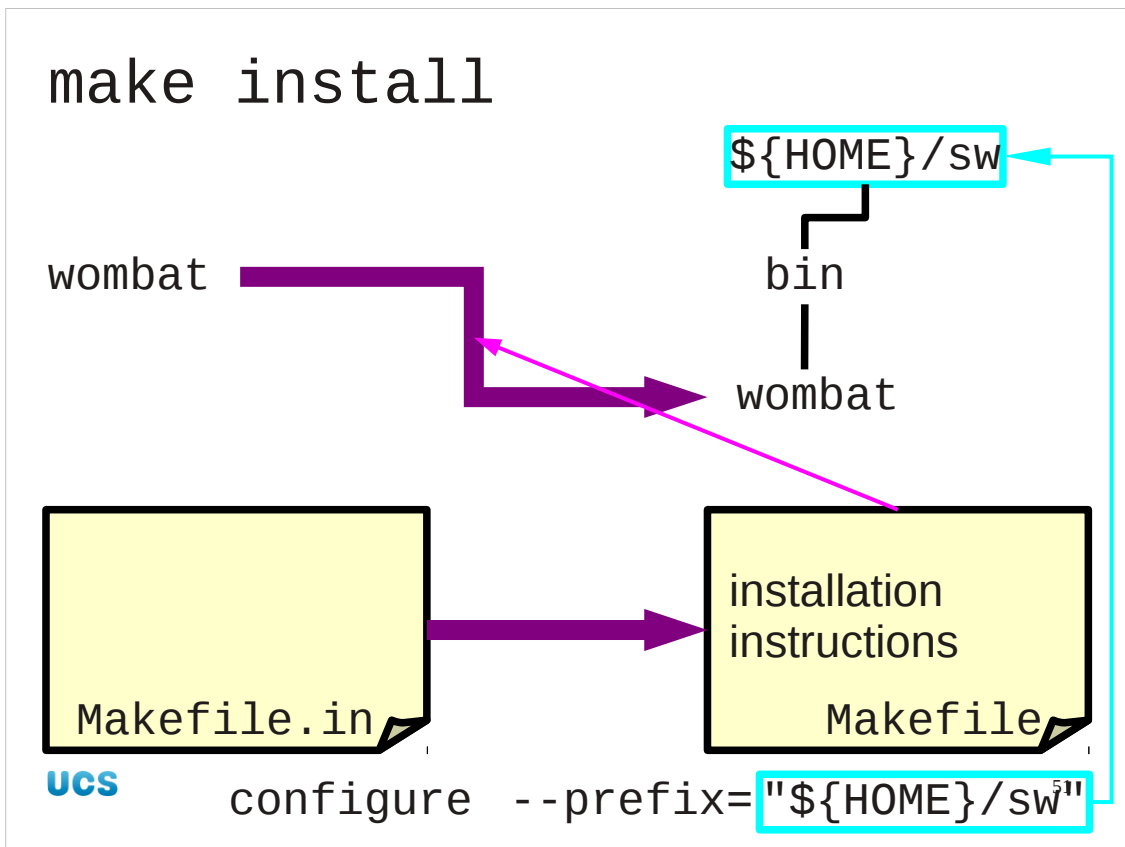
```
./configure --prefix="$HOME/sw"  
Configures OK.
```

```
make  
Builds OK.
```

Don't forget your lab book.



So we have built our software. Now we have to move it into place in our directory \$ {HOME}/sw.



The instructions for how to install the software are already built into the `Makefile`. The command “`make install`” will safely copy all the various files to be installed into the tree under `${HOME}/sw` in their appropriate locations. The knowledge of where to install the software was built into the `Makefile` (which controls `make`) by the `configure` script, using the location passed to it by its `--prefix` option.

```
$ make install
```

So, once again, the command we need to run is quite trivial:
“make install”.

This is typical of the `configure/make/make install` model. All the complexity is contained in the first phase, configuration. After that it should be trivial. If anything goes wrong at the build or installation phase then it's the *configuration* phase that should be looked at for errors.

Worked example

9. installation

```
$ make install
```

```
install -c xdaliclock /home/rjd4/sw/  
bin/xdaliclock
```



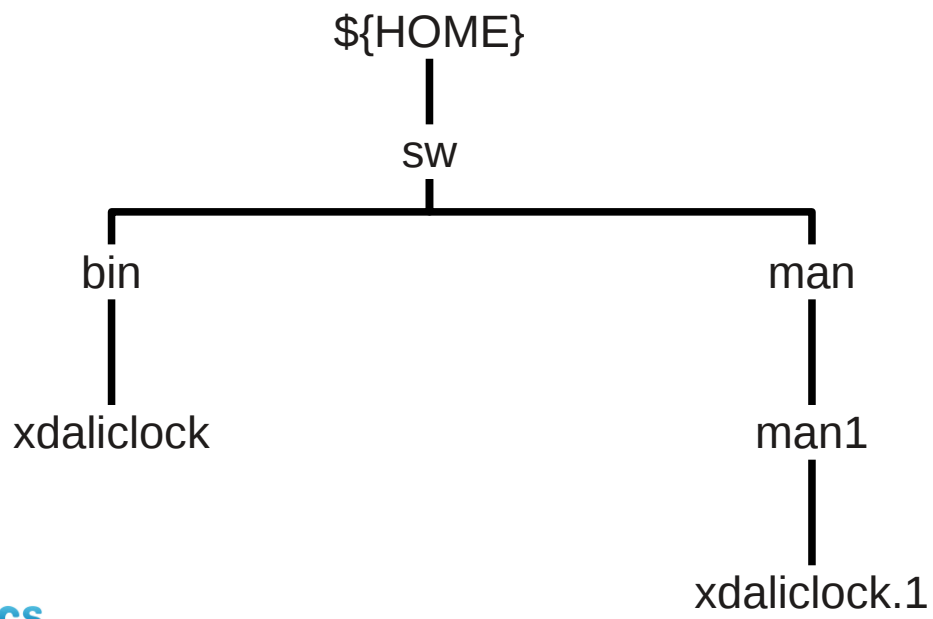
```
install -c ./xdaliclock.man /home/rj  
d4/sw/man/man1/xdaliclock.1
```



In our worked example, just two files get installed.

Worked example

9. installation

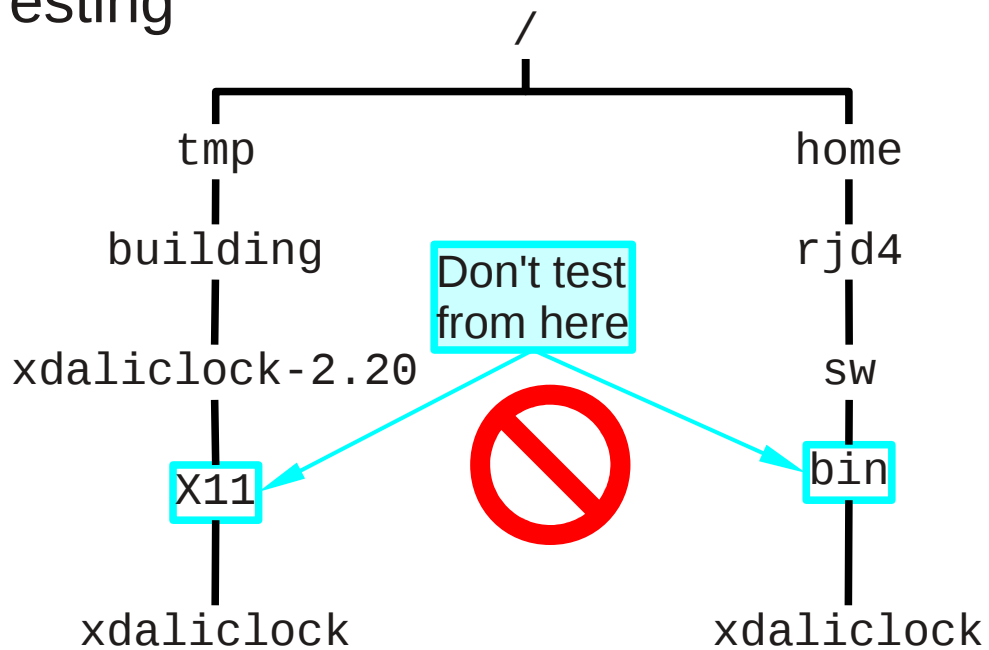


ucs

54

This is the structure that has been installed.

Testing



It's important that when you test to see that the binary works that you not be in the directory where the binary exists. There are two of these directories, `/${HOME}/sw/bin` where it has been installed and `/tmp/building/...` where it was built. The latter is quite likely to be your current working directory at this point in the process. Go back to your home directory before testing.

Worked example

10. testing

```
$ cd
```

go to home directory

```
$ xdaliclock &
```

new command

run in background



UCS

56

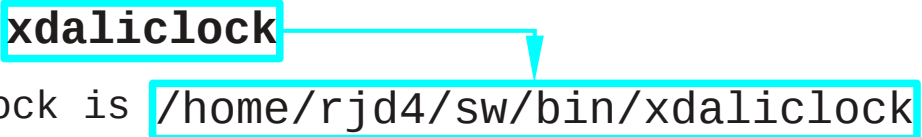
So we will go back to our home directories to do our tests. (The home directory is the default destination for cd.)

We give the command "xdaliclock" and it will be helpful to us to background the process by following the command with an ampersand ("&"). This lets us get on with other work while the program runs.

Worked example

11. testing

```
$ type xdalicklock  
xdalicklock is /home/rjd4/sw/bin/xdalicklock
```



There is one last thing you ought to check. Sometimes you may be installing your own version of a program that already exists. You might be installing a more recent version, for example. You ought to check that the program you get when you type its name is the version you have just installed.

Linux and Unix shells (the command line interpreters you type commands at) have a command called “type” which tells you where the program was found.

Worked example

12. lab book

make
Builds OK.

make install
Installs OK.
Works from home
directory.

Don't forget your lab book.

Long builds & installs

`make`



`make install`

ucs

`make && make install`

First

and if it works

Second

59

Here's a useful trick.

The build process can take a long time. So can the installation process under certain circumstances. If you give the “make” instruction then you have to keep an eye on it to know when to give the “make install” instruction. This uses up your time.

Instead we can issue both instructions on the same command line with the caveat that the installation should only be attempted if the build succeeded. To do this we use a bit of shell magic.

The double ampersand (“&&”) with no space between the two symbols can be used between two commands on the command line. The second command is only run if the first command succeeded. If the first command fails for any reason the whole line is abandoned and the second instruction isn't even tried.

So instead of

```
$ make
```

```
...wait...
```

```
$ make install
```

```
...wait...
```

we can give the command

```
$ make && make install
```

```
...wait...
```

and only come back when the whole thing is finished.

Exercise

openbabel

`${HOME}`  `/tmp/building`

`openbabel-2.2.3.tar.gz`

1. unpack
 2. configure
 3. build
 4. install
- } &&

Lab
book!

UCS

60

Now you get to do one on your own.

In your home directory there is a compressed archive file `openbabel-2.2.3.tar.gz`. Unpack it, configure it for installation in `${HOME}/sw`, build it and install it.

You won't know how to test it. We will do that next.

If you get stuck, please feel free to ask.

Don't forget to fill out your lab books.

```
$ cd /tmp/building
$ tar -x -f ${HOME}/openbabel-2.2.3.tar.gz
$ cd openbabel-2.2.3
$ ./configure --prefix="${HOME}/sw"
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
...
config.status: executing src/babelconfig.h commands
$ make && make install
Making all in data
make[1]: Entering directory `/tmp/building/openbabel-2.2.3/data'
...
make[1]: Nothing to be done for `all-am'.
make[1]: Leaving directory `/tmp/building/openbabel-2.2.3'
```

Coffee break

Ten minutes

Don't just stare
at the screen!

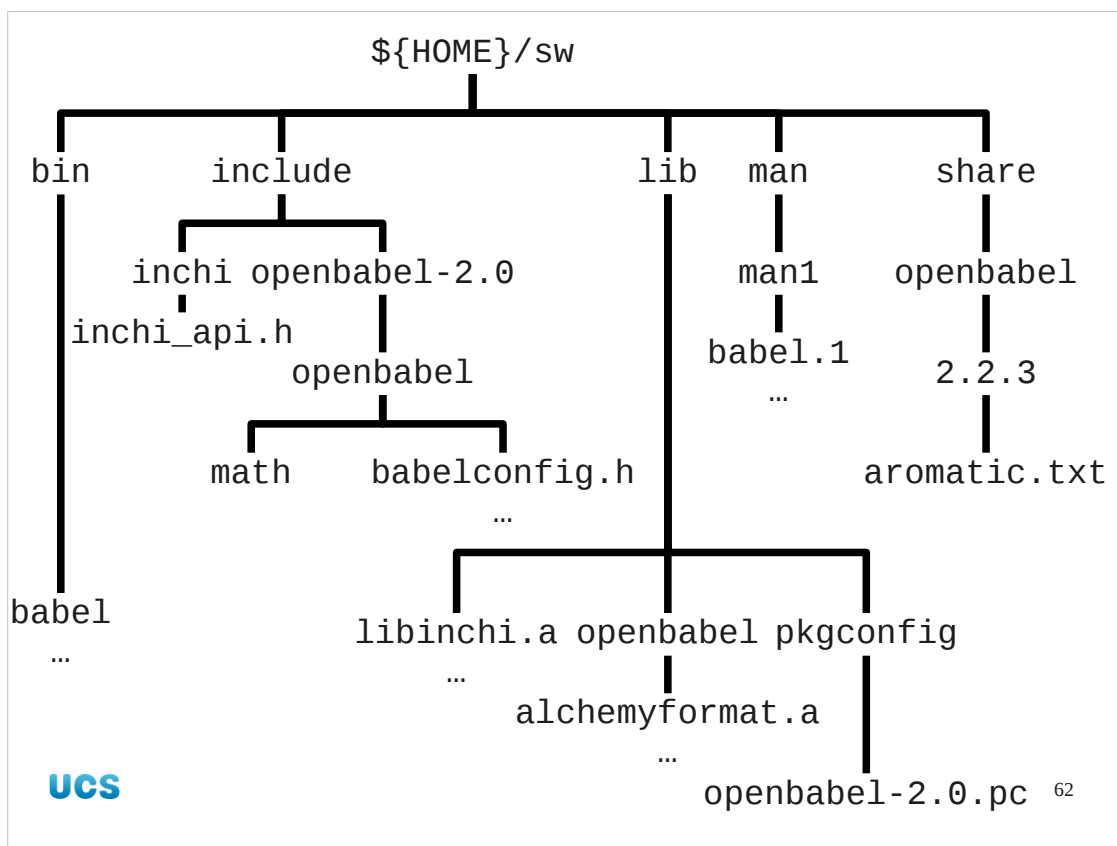
UCS



The build and installation will take about ten minutes. Have some coffee once you have the build going. Because you have coupled the build and the install both will (should!) happen while you are enjoying your coffee.

It's a bad habit to stare at the build logs scrolling by. To start with they're not particularly informative like that. Secondly and more importantly, it's a complete waste of your time. The best plan is to see if the build gets started. If a configured build is going to fail it will most often fail early.

This is your chance to do that fifteen minute chore you've been putting off. Failing that, ten minutes is usually about the right time to go to the coffee machine, get your caffeine fix, wander back and resume your trance-like state before the LCD altar.



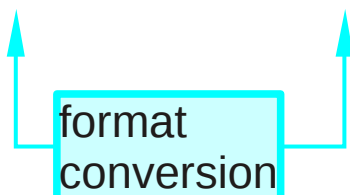
I want to emphasise just how much has been installed. The xdaliclock package installed two files. The openbabel package installed 236!

Exercise

openbabel

```
$ cd
```

```
$ babel ethanol.cml ethanol.xyz
```



UCS

63

Let's test the openbabel software.

The openbabel software includes a command line program called "babel" which converts one chemical description format into another. We will demonstrate it (and crudely test our software installation) with a couple of files.

We have put a file in your home directory called ethanol.cml. This is a description of the ethanol molecule in **C**hemical **M**ark-up **L**anguage. We will run the babel program to convert ethanol.cml into ethanol.xyz. This latter file is a much simpler representation of ethanol, mostly just listing the (x,y,z) locations of its constituent atoms.

```
$ cd
```

```
$ babel ethanol.cml ethanol.xyz
```

```
1 molecule converted
```

```
$
```

Feel free to look into any of the files; they are all just plain text.

Exercise

libghemical
liboglappth

libghemical-2.99.2.tar.gz
liboglappth-0.98.tar.gz

And now, to help us relax, we have a couple more builds which both “just work”. These will build a pair of libraries we need for the next part of the course.

The source files are in your home directories already.

Please build them in `/tmp/building`.

Please install them in `${HOME}/sw`.

It should take about 5 minutes to build both.

Dependencies

ghemical

needs


**openbabel
libghemical
liboglappth**

needs

base system

Now we will turn our attention to another chemistry package called `ghemical`, a graphical tool for building molecular diagrams. We will discover that `ghemical` needs `openbabel`, which we have just installed. (It also depends on the two libraries we have just installed but we won't trip over them the way we will with `openbabel`.) Of course, there is always a hidden dependency; `openbabel` and the other two libraries need certain elements of the base system to be installed, but we will ignore that dependency for now.

Worked example

`${HOME}`  `/tmp/building`
`ghemical-2.99.2.tar.gz`

We will copy `ghemical` from the usual location and unpack it. Normally we would read the `README` file but it's empty. There is also an `INSTALL` file. It tells us we need some libraries:

<code>glib</code>	<code>glib-2.0</code> version 2.6.0 or newer
<code>gtk+</code>	<code>gtk+-2.0</code> version 2.6.0 or newer
<code>gtkglext</code>	<code>gtkglext-1.0</code> version 1.0.5 or newer
<code>libglade</code>	<code>libglade-2.0</code> version 2.4.0 or newer
<code>gthread</code>	<code>gthread-2.0</code> version 2.6.0 or newer (optional)

(It also mentions a command called `autogen.sh`. In actual fact we won't need it but it wouldn't do any harm to run it. This is the script that builds the `configure` script. On some systems you may not have the necessary packages installed to run it properly.)

Failed dependency

```
$ ./configure --prefix="${HOME}/sw"  
...No package 'openbabel-2.0' found...
```



```
pkg-config  
PKG_CONFIG_PATH
```



UCS

67

If we try to configure the software it fails, complaining that it can't find openbabel-2.0!

```
$ ./configure --prefix="${HOME}/sw"  
checking for a BSD-compatible install... /usr/bin/install -c  
checking whether build environment is sane... yes  
...  
checking for GTK... yes  
checking for OPENBABEL... configure: error: Package requirements  
(openbabel-2.0) were not met:
```

No package 'openbabel-2.0' found

Consider adjusting the PKG_CONFIG_PATH environment variable if you installed software in a non-standard prefix.

I do like error messages that tell you what to do!

pkg-config

What are the library options for...

```
$ pkg-config --libs gtkglext-1.0
```

```
-Wl,--export-dynamic -lgtkglext-x11-1.0  
-lgdkglext-x11-1.0 -lGLU -lGL -lXmu  
-lXt -lSM -lICE -lgtk-x11-2.0  
-lpangox-1.0 -lX11 -lgdk-x11-2.0  
-latk-1.0 -lgio-2.0 -lpangoft2-1.0  
-lgdk_pixbuf-2.0 -lpangocairo-1.0  
-lcairo -lpango-1.0 -lfreetype -lz  
-lfontconfig -lobject-2.0 -lgmodule-2.0  
-lglib-2.0
```

UCS



68

How do we describe the contents of one package to another?

The way it is done in the configure, build, install world is through a command called `pkg-config`. This command can be used to reveal the options that need to be passed to the build system for `gchemical`, say, to tell it about the already built system of `gtkglext`, say.

This example requests the options needed for the `gtkglext` libraries to be used. It is also a good example of why we need tools like `pkg-config` to assist us with this sort of thing.

pkg-config

Our software



```
$ pkg-config --libs openbabel-2.0
```

```
pkg-config --libs openbabel-2.0
Package openbabel-2.0 was not found
in the pkg-config search path.
Perhaps you should add the directory
containing `openbabel-2.0.pc' to the
PKG_CONFIG_PATH environment variable.
No package 'openbabel-2.0' found
```



UCS

69

And here's it not working for openbabel:

```
$ pkg-config --libs openbabel-2.0
```

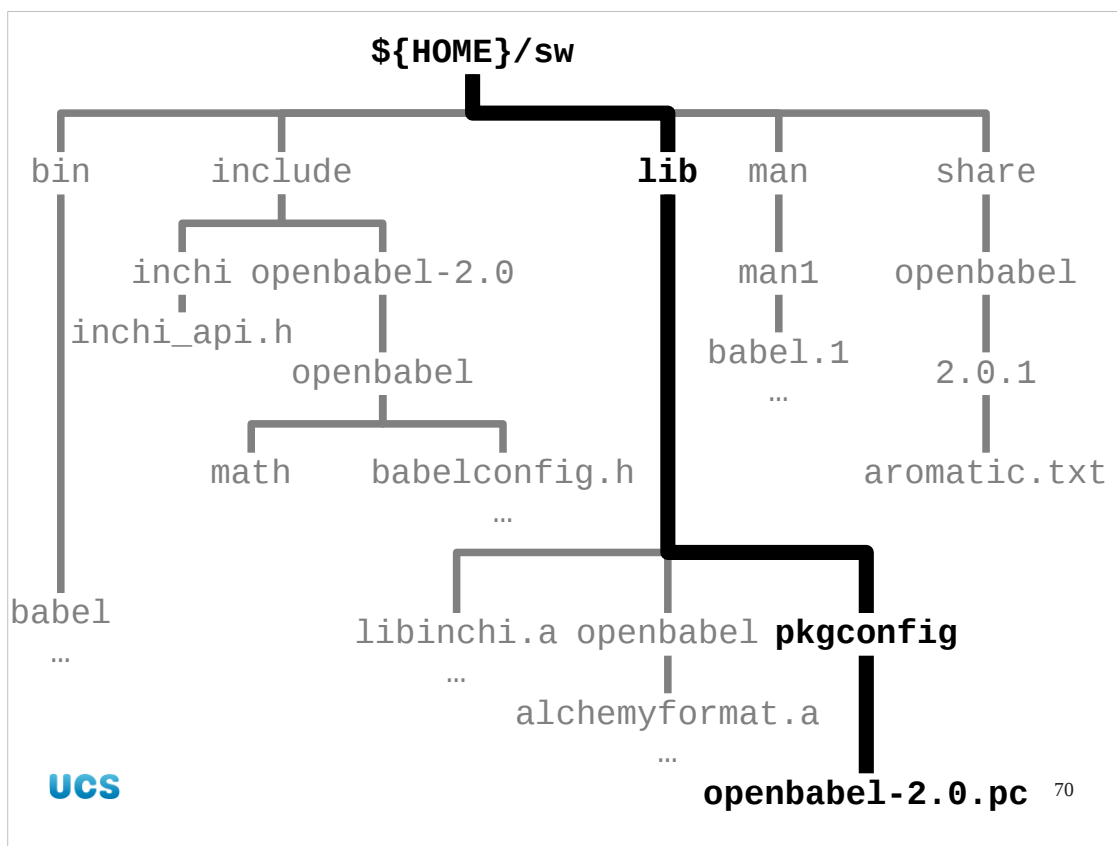
```
Package openbabel-2.0 was not found in the pkg-config search
path.
```

```
Perhaps you should add the directory containing `openbabel-
2.0.pc' to the PKG_CONFIG_PATH environment variable
```

```
No package 'openbabel-2.0' found
```

This is a particularly useful error message, though. It tells us exactly what we need to do to fix the problem.

First we have to find what directory `openbabel-2.0.pc` is in and then we have to add it to another `...PATH` environment variable, just like we did for `PATH` and `MANPATH`.



We can find it under our `${HOME}/sw` tree with `find`:

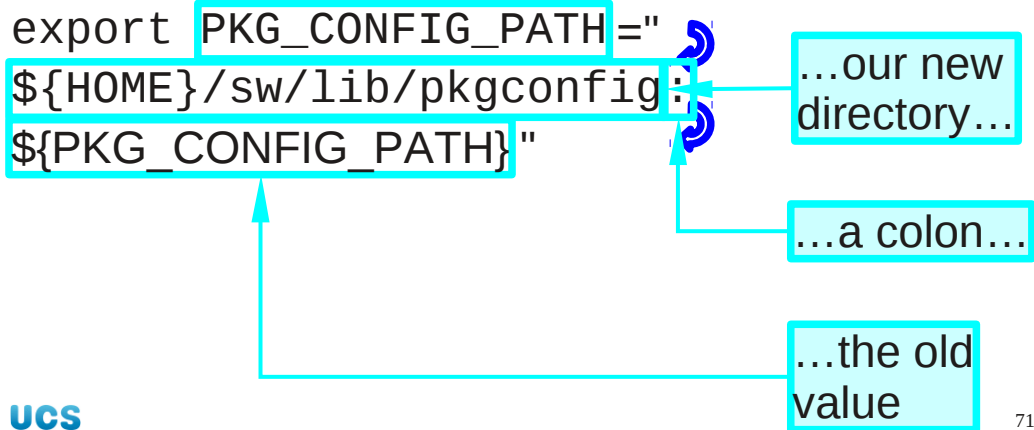
```
$ find "${HOME}/sw" -name openbabel-2.0.pc
/home/y550/sw/lib/pkgconfig/openbabel-2.0.pc
```

The standard place for `configure` to have these package configuration files put is in `lib/pkgconfig` relative to the directory specified by the option `--prefix`. We will add `${HOME}/sw/lib/pkgconfig` to `PKG_CONFIG_PATH`.

This matches the equivalent location for system packages: `/usr/lib/pkgconfig`. This contains the `gtkglext-1.0.pc` file used in the working example.

`${HOME}/.bashrc`

Set this environment variable to be...



We will update our `PKG_CONFIG_PATH` environment variable in our `${HOME}/.bashrc` files, just as we did for `PATH`, `MANPATH` etc.

It's a colon-delimited list as ever, so its new value will be our new directory, `${HOME}/sw/lib/pkgconfig`, followed by a colon, followed by whatever value it had before, `${PKG_CONFIG_PATH}`.

Exercise

1. Copy in a new `${HOME}/.bashrc` file.

```
${HOME}/bashrc2  
└─┬─┘ ${HOME}/.bashrc
```

2. In your existing terminal window...

```
$ pkg-config --libs openbabel-2.0  
Package openbabel-2.0 was not found  
in the pkg-config search path.
```

```
...
```

UCS

72

Again, there is a prepared `${HOME}/.bashrc` file in the course directory as `bashrc2`.

1. Copy it into your home directory as `.bashrc`.

```
$ cp /ux/Lessons/Building/bashrc2 ${HOME}/.bashrc
```

2. Check that `pkg-config` still cannot find `openbabel-2.0`. Recall that this file is only read at the *start* of a session.

```
$ pkg-config --libs openbabel-2.0  
Package openbabel-2.0 was not found in the pkg-config search  
path. Perhaps you should add the directory containing  
'openbabel-2.0.pc' to the PKG_CONFIG_PATH environment variable.  
No package 'openbabel-2.0' found
```


Exercise

3. Launch and use a new terminal window...

```
$ pkg-config --libs openbabel-2.0  
-L/home/rjd4/sw/lib -lopenbabel
```

4. Close the old terminal window.

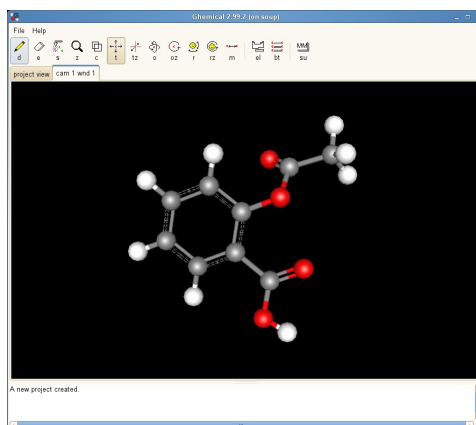
3. Launch a new terminal window and try the `pkg-config` command again. This time the fresh session should have read your new `${HOME}/.bashrc` file and have set the `PKG_CONFIG_PATH` environment variable appropriately. So this time the command should work.

4. Close the old terminal window. It will only confuse matters.

Exercise

ghemical

1. configure
2. build
3. install
4. launch



UCS

74

So now we have our dependencies sorted, and pkg-config knows we have them sorted, we can return to the task of building ghemical.

```
$ cd /tmp/building/ghemical-2.99.2
```

```
$ ./configure --prefix="${HOME}/sw"
```

...

```
config.status: executing depfiles commands
```

```
config.status: executing default-1 commands
```

```
config.status: executing stamp.h commands
```

```
config.status: executing po/stamp-it commands
```

```
$ make && make install
```

...

```
$ cd
```

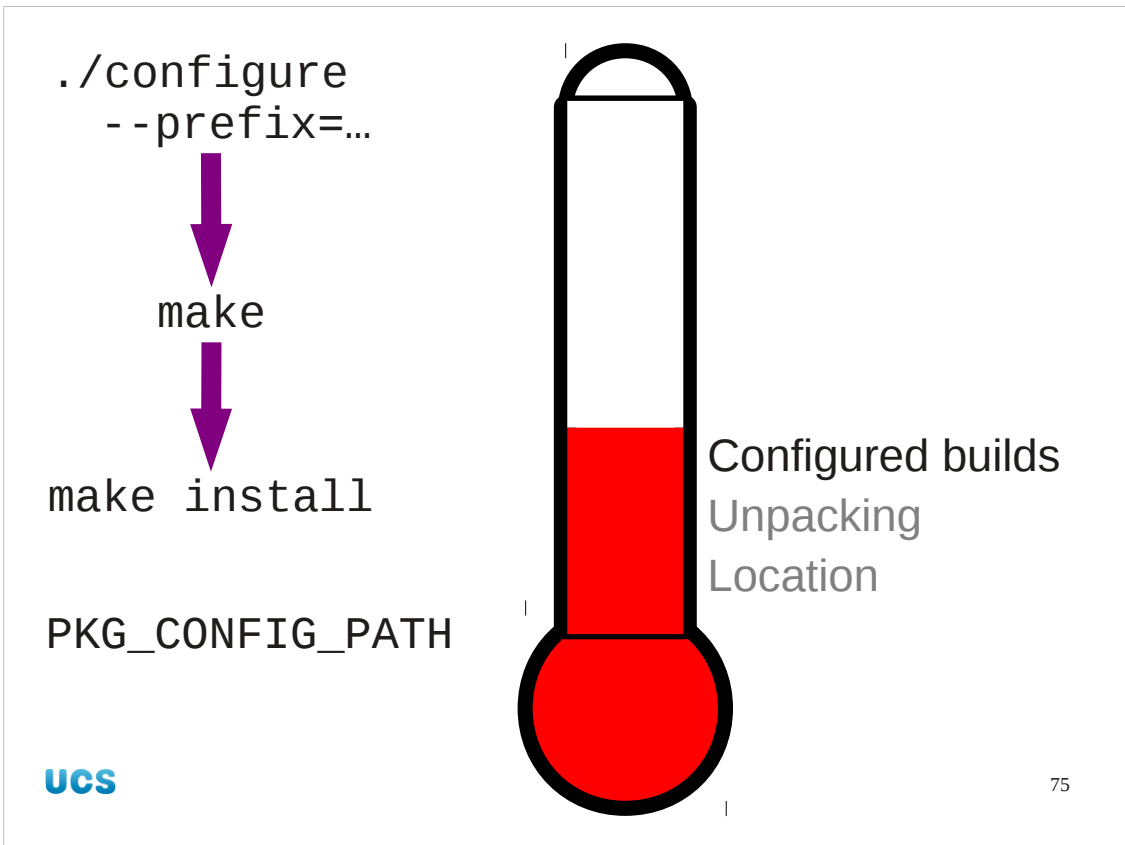
```
$ type ghemical
```

```
ghemical is /home/rjd4/sw/bin/ghemical
```

```
$ ghemical
```

There are some example files in

/tmp/building/ghemical-2.99.2/examples. The one shown in the slide is acetylsalicylic acid.



So now we have completed our tour of “configured” builds. They're all much the same. All the work goes into the configuration. After that it should be “make” and “make install”. If the `PKG_CONFIG_PATH` environment variable contains your `~/sw/lib/pkgconfig` directory then configure's dependency checking should be aware of your personal repository too.

Python packages

Equivalent for Python

“distutils”

```
./configure  
--prefix=...  
make  
python3 setup.py build  
python3 setup.py install --prefix=...
```

ucs

I want to end by illustrating that the idea of your own personal software repository, identified as a “prefix” to build instructions is not restricted to configure scripts. While configure/make/make install is the most common (by far) approach to compiled software there are alternatives out there which also use the same core idea. One that you might meet is the Python build system for its packages. Python has a system called “distutils” (distribution utilities) which it uses to help distribute Python modules. It too uses a prefix approach. The last parts of this course will be a quick demonstration of the Python analogue of what has gone before. If you don’t use Python don’t worry; it will be over in a very few minutes. The key to spotting a distutils build is the presence of a file called setup.py. This is analogous to the configure script. There are two phases to distutils: build (which includes configuration and compilation) and install.

Python example

Plotting library

`matplotlib`

Only a pre-release version available for Python 3.

```
>>> import matplotlib
```

```
...
```

```
ImportError: No module named matplotlib
```

So let's build it!

UCS

77

My worked example will be the matplotlib Python module, which is a graphics module that has only just started to be ported to the new Python3 system. So it is currently not installed. We are going to build our own.

Unpacking Python packages

matplotlib-1.2.0rc2.tar.gz



Same old same old...

matplotlib-1.2.0rc2

Unpacking is unpacking is unpacking...

Building Python packages

```
$ python3 setup.py build
```

```
basedirlist is: ['/usr/local', '/usr']
```

```
=====
```

```
BUILDING MATPLOTLIB
```

```
    matplotlib: 1.2.0rc2
```

```
        python: 3.2.3
```

```
        platform: linux2
```

```
...
```

Unlike a configure script, we don't specify the prefix at build time.

Installing Python packages

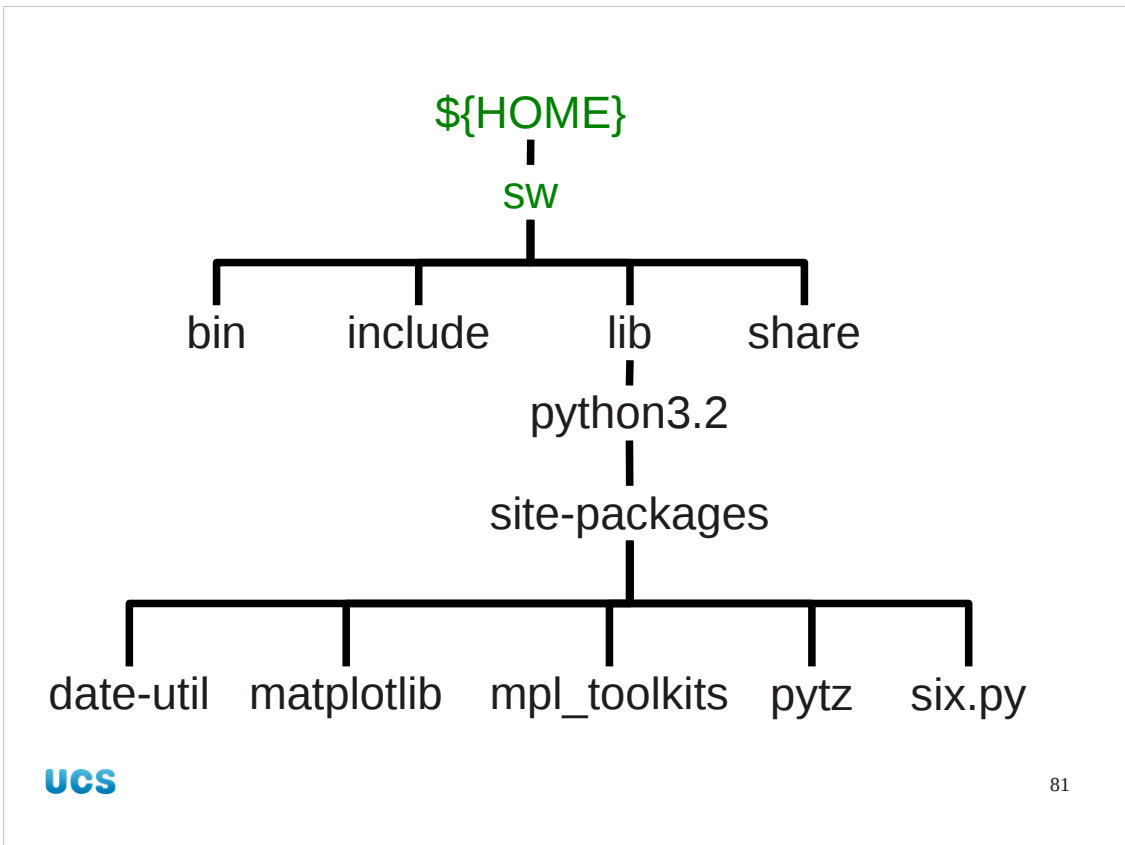
```
$ python3 setup.py install
--prefix="${HOME}/sw"

basedirlist is: ['/usr/local', '/usr']
=====
BUILDING MATPLOTLIB
      matplotlib: 1.2.0rc2
        python: 3.2.3
      platform: linux2

...

```

And then we install, specifying the directory it should be installed in with a “--prefix” option.



It installs into the private software repository in an exactly analogous way to how it would have installed into the /usr system repository.

Running Python packages

```
$ export PYTHONPATH=  
"${HOME}/sw/lib/python3.2/site-packages"
```

```
$ python3  
Python 3.2.3
```

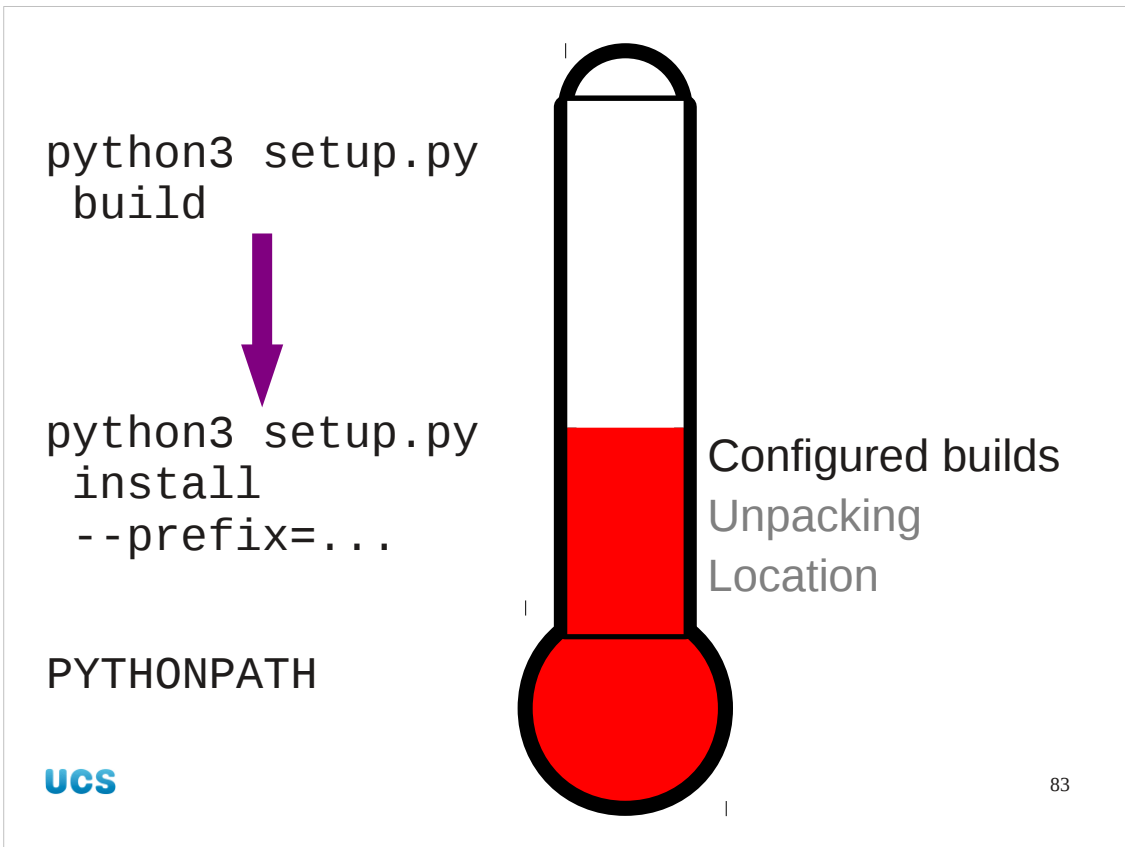
```
>>> import matplotlib  
>>> help(matplotlib)
```

```
...
```

```
FILE
```

```
ucs /home/y550/sw/lib/python3.2/  
site-packages/matplotlib/___init___.py82
```

We need to set an environment variable (also ending "...PATH") to tell Python to look there, but then we're done.



So the ideas of configured builds and their support for private repositories carries over to Python too.

With this, we end today's session. See you all next time.