The University Of Sheffield.

The University of Sheffield CiCS Research IT

Development Tools For HPC Applications

Michael Griffiths
IT Services
The University of Sheffield
Email m.griffiths@sheffield.ac.uk,

The University Of Sheffield.

The University of Sheffield CiCS Research IT

Outline

- Building Applications
  - Gnu compilers, g++, g++, g77
  - Portland compilers, pg++, pgf77
    - Calling fortran from C and C from fortran
- Using, Building and installing libraries
- Using the make utility
- The Eclipse Development Environment
- Links

The University Of Sheffield.

The University of Sheffield CiCS Research IT

BUILDING LARGE APPLICATIONS

- Typically compile program using
  - g++ –o myprog myprog.c –lm –g
- Large programs
  - Modularized
  - Combine into a single executable
- Building large applications is a multi step process
  - Compile each source file
  - Link resulting objects into an executable

The University Of Sheffield.

The University of Sheffield CiCS Research IT

EXAMPLE MULTI SOURCE PROGRAM:1

- To build the Monte-carlo model, mc, we do the following.
  - g++ –c –g mc.cpp
  - g++ –c –g mc_system.cpp
  - g++ –c –g mc_particle.cpp
  - g++ –c -g mc_statistics.cpp
  - g++ –o mc mc.o mc_system.o mc_particle.o mc_statistics.o –lm
- Note: only one of the sources has a main function

EXAMPLE MULTI SOURCE PROGRAM:2

- If mc_system.cpp is edited we don't need to recompile
  - mc_statistics, mc_particle or mc
- Rebuild the application as follows
  - g++ –c –g mc_system.cpp
  - g++ –o mc mc.o mc_system.o mc_particle.o mc_statistics.o –lm
- Automate these steps using make

LIBRARIES

- Libraries are packaged collections of object files
  - Standard library contains printf… etc..
  - Maths library contains sin, cos etc..
- Specify additional libraries with –l<name>
  - Only standard library is provided automatically
- To compile a program with a maths library
  - g++ –c myprog myprog.c -lm

The University Of Sheffield.

The University of Sheffield CiCS Research IT

BUILDING YOUR OWN LIBRARY

- Benefits of building libraries
  - Share standardised functions with community
  - Separate functionality from detailed code
  - Good way of packing up your most useful routines and reusing them
- How to build
  - Build libraries using
  - Named as lib<name>.a or lib<name>.so
  - http://www-cs.canisius.edu/PL_TUTORIALS/C/C-UNIX/libraries

EXAMPLE

- Example my util library
  - g++ -c vec.cc
    - Generates vec.o
  - g++ -c mat.cc
    - Generates mat.o
- Add object files to library
  - ar r myutillib.a vec.o
  - ar r mylibutil.a mat.o
- Don’t use –l for your own libraries link as follows
  - g++ myprog.cc mylib.a –o myprog

The University Of Sheffield.

The University of Sheffield CiCS Research IT

INSTALLING A LIBRARY

- General steps
  - Download and uncompress source
  - Read documentation and build e.g. using configure
  - make and make install to build and install
  - Update your environment
    - Set LD_LIBRARY_PATH
    - Compile with -lMyNewLib

The University Of Sheffield.

The University of Sheffield CiCS Research IT

USING THE MAKE UTILITY

- Used to compile and link programs
- Makefile tells make how to perform link and compilation
- Consists of rules with the following shape

```
target …… : dependencies ……
        command
        ……
```

The University Of Sheffield.

The University of Sheffield CiCS Research IT

MAKE

- target name of file generated by a program
- dependency used as input to create target
- Target files are created whenever a dependency has changed
- Commands can include
  - cc, CC, g++, f77, f95, mpf77
- make
- make clean

MAKE TARGET

- Perform actions to obtain a target from a set of dependecies
- Make checks when dependencies last updated

target : dependencies

rule

The University of Sheffield CiCS Research IT

SIMPLE MAKEFILE …..
ALMOST TRIVIAL!

```
game : game.o
	gcc -o game game.o

game.o : game.c
	gcc -c game.c

clean :
	rm game game.o
```

SIMPLE MAKEFILE

- Generates executable called game from a single source file called game.c
- Has a sequence of rules
  - game
    - Rule for building target executable file
  - game.o
    - Rule for building object files
  - clean
    - Rule for cleaning executable and object files

MAKE MULTIPLE SOURCE FILE PROJECT

```
project : main.o data.o io.o
        gcc -o project main.o data.o io.o

main.o : main.c io.h data.h
        gcc -c main.c

data.o : data.c io.h data.h
        gcc -c data.c

io.o : io.c io.h
        gcc -c io.c

clean :
        rm project main.o data.o io.o
```

The University Of Sheffield.

The University of Sheffield CiCS Research IT

HINTS FOR BUILDING MAKEFILES

- Use # at the start of a line for comments
- Use \ at the end of a line for line continuation
- The line defining the rule that follows the definition of target and dependencies should normally be indented using a tab character and NOT whitespace characters

MAKEFILE WITH IMPLICT RULES FOR COMPILING A STATIC LIBRARY

```
objects = vec.o vecpair.o mat.o

flags = -fast -tp k8-64
libmyutil.a : $(objects)
        ar -r -o myutil.a $(objects) $(flags)

vec.o : vec.c
        pgCC -c vec.c $(flags)
vecpair.o : vecpair.c
        pgCC -c vecpair.c $(flags)
mat.o : mat.c
        pgCC -c mat.c $(flags)

clean :
        rm myutil.a $(objects)
```

MACROS USED WITH MAKEFILES

$@	Full name of the current target .
$<	The source file of the current (single) dependency .
$*	The part of a filename which matched a suffix rule.
$?	The names of all the dependencies newer than the target separated by spaces.
$^	The names of all the dependencies separated by spaces, but with duplicate names removed.

SUFFIXES

- **Make** uses a special target, named .SUFFIXES to allow you to define your own suffixes.
- For example, the dependency line:
  .SUFFIXES: .foo .bar
  - tells **make** that you will be using these special suffixes to make your own rules.

The University of Sheffield CiCS Research IT

CUSTOM SUFFIX RULE

•Similar to how **make** already knows how to make a *.o* file from a *.c* file, you can define rules in the following manner:

.foo.bar: tr '[A-Z][a-z]' '[N-Z][A-M][n-z][a-m]' < $< > $@ .c.o: $(CC) $(CFLAGS) -c $<

•The first rule allows you to create a *.bar* file from a *.foo* file. (Don't worry about what it does, it basically scrambles the file.)
• The second rule is the default rule used by **make** to create a *.o* file from a *.c* file.

MAKEFILE WITH SUFFIX RULE

```
objects = blastest.o

flags = -fast -tp k8-64

mk4 : $(objects)
	pgCC -o mk4 $(objects) $(flags)

.c.o:
	pgCC -c $(flags) $<

clean :
	rm mk4 $(objects)
```

The University Of Sheffield.

The University of Sheffield CiCS Research IT

LINKS

- http://www.eclipse.org/
- http://www.cplusplus.com
  - Very useful reference section