Porting between Operating Systems

or

how to increase your customer base

Harry Powell MRC-LMB

Why bother to port (doesn't everyone use MS-Windows)?

- support a wider community (some people use other platforms) - more people will use your software
- future proofing (remember DEC VAX or PDP-11? or IBM 370/168?)
- leads to more standard code (which should be more maintainable)

Which platforms/operating systems to port to? This depends on:

- who you want to use your software (most important)
- what hardware you have available for building/testing

Serious developers will usually have access to:

- PC / Linux (no excuse for not having it!)
- PC / MS-Windows (probably)
- Mac / OS X (becoming more popular)
- SGI / Irix (historically important)
- Alpha / Tru64 (historically important)
- Sun /Solaris or SunOS (niche product in crystallography)
- HP-UX (niche product in crystallography)

Three basic types of porting;

1. existing software
2. new graphical interfaces
3. functionality

Porting existing software

Porting Existing Software

Assumption that a complete re-write is not necessary/appropriate/desired

Reasons include:
this program is

- mature;
- does its job;
- will not be developed much further;
- enormous (any re-write will take many man-years & introduce many bugs)
- obsolescent (*i.e.* it will be rewritten when time permits)

Porting Existing Software

Used to be expensive/difficult - only readily available compilers were commercial and tied to a platform, had useful platform dependent features (e.g. VAX FORTRAN)

Now platforms exist which have a “free” development environment (e.g. Linux, Mac OS X, Cygwin/MinGW)

“Free” compilers which adhere to the standards now exist (gcc/g77 for most platforms, Salford FTN77 for MS-Windows, icc/ifc for Intel Linux)

Porting Existing Software

Why use a commercial compiler?

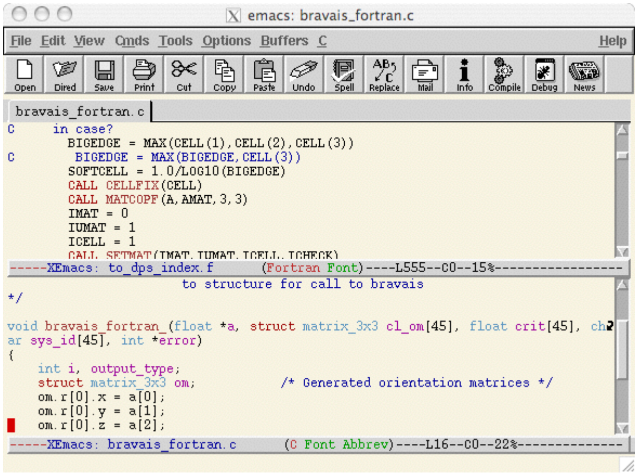
- “You get what you pay for”
  - user support
  - often better optimization (for specific platforms)
  - good development environments (e.g. Visual Studio)
  - your institution/company may already have paid for and own a licence

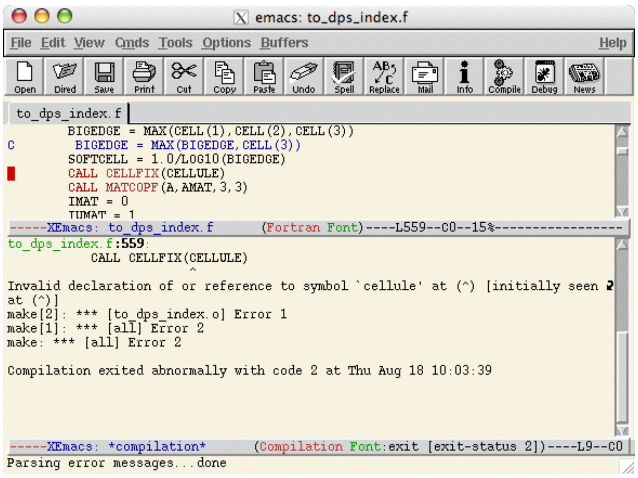
Porting Existing Software

Why use a “free” compiler?

- low initial cost
- modern ones often produce executables about as fast as commercial ones
- same compiler across multiple platforms
- readily available cross compilers (build on Linux, run on Windows - or *vice versa*)
- good development environments (e.g. XEmacs)

but - you get what you pay for, so it can be buggy - e.g. gcc 2.96 distributed with Red Hat Linux should never have been released





Porting from MS-Windows to other systems

- need a system to build on
- need a system to test on

Possible to cross-develop on MS-Windows but the target windowing system will be different.

Two easy routes -

1. install Linux and dual boot (best environment)
2. install Cygwin (easiest since you can run MS-Windows simultaneously)

Each of these provides a fully-featured windowed environment *via* X11 windows, but Cygwin will produce MS-Windows executables unless you cross-compile!

Porting from any UNIX to other systems

- “easy” if the other system is UNIX-based
- can be an opportunity to modularize functionality
- for MS-Windows need a system (probably not a problem for most people):

  Choices:
  - MinGW (Minimalist GNU for Windows), provides compilation tools under Windows
  - Cygwin; easy to migrate a UNIX build (essentially identical to Linux on PC)
  - Cross-compilers; build under system A, run on system B.

Problems with porting (1):

It will probably identify bugs in your code (some “strict” compilers are stricter than others):

```
WRITE(*,*),’hello world’
```

compiles as expected with f77 on Tru64 UNIX, g77 on Linux & MacOS, but not with xlf on MacOS.

```
WRITE(*,*)’hello world’;
```

compiles as expected with f77 on Tru64 UNIX, g77 on Linux & MacOS, xlf on MacOS, but not on Irix with f77 (because it's actually f90).

Problems with porting (2):

May identify bugs in compilers - e.g. file “break.f” contains the following:

```
   CHARACTER*2 TEST(2)
      CHARACTER*1 JUNK
      JUNK = 'O'
      WRITE(TEST,FMT=1000)JUNK
 1000 FORMAT(A)
      END
```

```
[g4-15:~/programs] harry% gfortran -c break.f
break.f: In function 'MAIN__':
break.f:3: internal compiler error: Bus error
Please submit a full bug report,
with preprocessed source if appropriate.
See <URL:http://gcc.gnu.org/bugs.html> for instructions.
```

This can lead to a new career in compiler development!

Problems with porting (3):

If you can't distribute static binaries, your customer's computers are probably missing vital libraries, shared objects, and other bits and pieces necessary to run your code which you

(a) didn't think were necessary
(b) assumed were ubiquitous
(c) hadn't even thought about

They might be there, but are from an older/newer version of the operating system and are therefore incompatible.

Problems with porting (4):

You may feel that months of work may be wasted when a manufacturer makes a major change to its hardware or operating system, e.g.

Apple - OS X made all previous Apple software obsolete

Apple - change of processor from PowerPC to Intel makes xlc/xlf compiler specific work obsolete.

Problems with porting (5):

Porting to (and from) MS-Windows is probably the most difficult step because of its unique environment; all other popular platforms share a similar operating system and have available a similar windowing system.

For example, *identical* code for Mosflm* (150,000 lines of Fortran, 100,000 lines of C, + 80,000 lines of graphics library) builds using the same commands on all UNIX boxes, but without tools like Cygwin requires a major restructuring for MS-Windows.

* a popular data integration program

new graphical interfaces

Command-line interfaces - old-fashioned but generally straightforward; however, MS-Windows users won't like it!

Graphical interfaces - need to distinguish between

- windowing environment - most platform dependent
  - MS-Windows - tied to Windows PCs
  - Aqua - tied to Macintosh
  - X11 - can be used on anything it's been ported to
- interface utilities -
  - Tk (Tcl/Tk, Tkinter, ...)
  - wxWidgets (*was* wxWindows), Qt, Clearwin...
  - browser applets

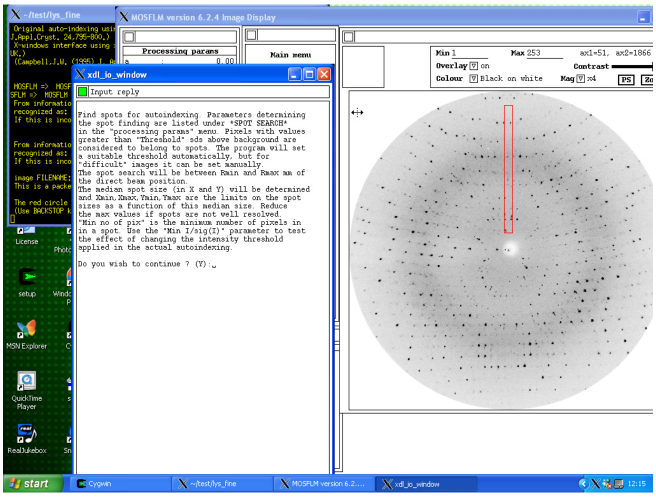
For portability we need the utility to have been ported to the environment (or the environment to have been ported to the platform)

The current Mosflm approach to a GUI -

1. uses a custom-built (obsolete) X11 widget set - `xdl_view`
2. compile to create a monolithic program

Pros:
- distribution of the executable is easy (no need for extra libraries for the GUI)
- familiar widget set (for the programmer, anyway)

Cons:
- development of both Mosflm and the GUI are intimately linked
- "hard" to port to OS's without an X-server
- `xdl_view` can behave subtly differently on different OS's - relied on some benign features of compilers



The new Mosflm approach to a GUI;

1. core crystallographic functions (controlled by command line instructions)
2. A Tcl/Tk GUI which reads XML and writes native Mosflm commands
3. the two parts communicate *via* TCP/IP sockets

Pros:
- Mosflm can be run on any platform independently of the GUI
- development of both Mosflm and the GUI can proceed semi-independently

Portable interface utilities 1 - Tcl/Tk

mature but losing popularity; development in some important modules is almost non-existent (*e.g.* incrTcl).

Installation of Tcl/Tk based programs may require additional packages to be installed (*e.g.* TkImg), but some are not available for all platforms (*e.g.* BLT is not available for Aqua).

Tkinter is Python's *de facto* standard GUI package. It is a thin object-oriented layer on top of Tcl/Tk.

can interface with compiled code via 4 basic routes -

1. embed compiled code into Tcl/Tk script
2. run external programs directly from the script
3. read from & write to a file
4. read from & write to a TCP/IP socket

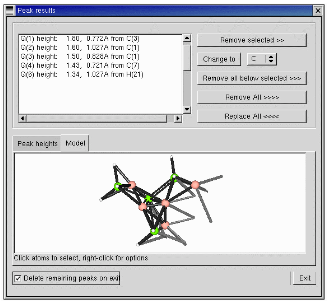
Portable interface utilities 2 - wxWidgets

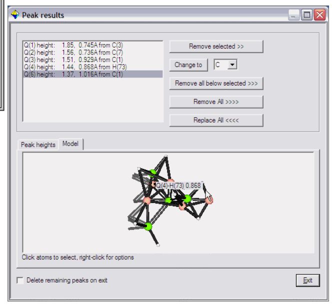
Set of C++ class libraries
Main differences from Tcl/Tk:

- encourages an object oriented approach
- the GUI can be compiled so that the programming details can be hidden from the user; also this can give a performance advantage, especially for interactive graphics.

Gives a native look and feel on different platforms, usable with any common C++

wxPython is “a blending of wxWidgets with Python” - used in PHENIX





A popular small molecule program running on Windows and Linux

Porting functionality

Could be viewed as a new project - coding from scratch - probably best to use new languages (C++, Python, Java) rather than established languages.

What you use depends on the application to some extent, e.g.
web applications - Java
heavyweight applications - C++
prototyping - Python

If you have a free choice at this point, development can be faster and portable features can be designed in.

Why not just use FORTRAN?

- fast executables with little effort
- many applications/libraries available
- well-established amongst senior scientists
- F90, 95, 200x have modern programming constructs

Why *shouldn’t* you use FORTRAN?

- can quickly lead to spaghetti code
- possible lack of long-term maintainability
- little support from “real” programmers (it isn’t taught in CS departments!)
- modern languages make development faster
- good OOP imposes modularity

Don’t re-invent the wheel unnecessarily - there is a whole host of functionality which has already been coded (software libraries for underlying crystallographic operations) - but there may be licensing issues.

Code can be cribbed from books (e.g. “*Numerical Recipes”*) but be aware of copyright issues.

Other developers may be willing to "give" you their code for inclusion (e.g. both autoindexing routines in Mosflm)

Finally -

Design in portability -

- don’t use platform specific features if possible
- use standard programming contructs
- gcc/g77/g++ is available for virtually every platform - essentially the same build can be used (some compiler flags and libraries may differ)