Origins of this presentation

Several Overload articles on porting:
• Porting: Overload 43, 44, 45 (June - October 2001)

Built on earlier articles:
• Overload 39 (September 2000) : Include files
• Overload 40 (December) : More Include files
• Overload 41 (February) : Source Code Layering
• Overload 42 (April) : Deeper look at Inline functions

Also: Presentation to ACCU-Silicon Valley (August 2001)
( All archived at www.allankelly.net )
Overview

• Why do we port?
• Differences
• Tools
• Shortest route to the top
• Summary and closing remarks

No Q&A at the end
- interrupt me with questions!
Who is Allan Kelly?

• Over 10 years of software development
• Independent contractor in the UK for 5 years
• Worked for Sema Group, BZW (Barclays Capital), Reuters, DIP, Nixdorf and more….
• Two years in Silicon Valley - Tumbleweed Communications and PostX Corp.
• Currently enjoying some leisure……
What is Porting?
• Moving an application from one platform to another

What is a Platform?
• OS
• Database
• API
• Third party library
• Compiler
• Language (computer)
• Language (human)
• Anything else…. 
Ports seldom involves one parameter

• Don’t have the luxury of *All other things being equal*
• Changing OS often means changing compiler e.g. Forte on Sun, GCC on Linux
• Change Database means an API change
• (One of our greatest advantages is keeping these the same e.g. GNU C++)
So, Platform is vague….

• Deliberately leave the definition of platform vague so we can encompass more
• But…..
• OS ports and Database ports are the two big ones
Break: why do we port?

• 3 types of software development houses:
  – Mass producer: 1 to many e.g. Microsoft
  – Specialist producer: 1 to few e.g. Dodge
  – Bespoke: 1 to 1 often inhouse e.g. Goldman Sachs, or bespoke developers e.g. Accenture
Mass software: Why port?

• Porting is not common: often PC and Mac versions are different source code base
• Typically many customers: no individually customer has a great say
• Can avoid porting because they have mass market
• May decide to port to prevent competition filling a void
• Port for public relations (Linux version?)
Bespoke software: Why port?

- Porting is not common
- Organisation may change its preferred platform
- May buy in third party software which has then to be ported
Specialist software: Why port?

- This is where most porting is found
- Typically one company supplying a small number of products to a niche market of a few customers, e.g. much financial software
- Customer decides the platform they decide the product
- Customers dictate the platform
- No port, no sale
- This goes for OS, database, etc.
- These companies are the most likely to have limited resources
Libraries: Why port?

- Special case
- Many library producers actively promote their products as multi-platform, e.g. Rogue Wave
- Many decisions are restricted by marketing requirements; e.g. Rogue Wave use “lowest common denominator” makefiles
- Almost impossible for these companies to support every configuration
- For a library to be widely accepted it must have a number of ports available, e.g. Boost
In an ideal world....
....we know which platforms we are targeting when we start

In the real world.....
.... software is usually developed and deployed before we decide to port

Either:

• port and develop in parallel (can be tricky)
• freeze new development and port (most be quick)

What are your business requirements?

• Branch appears attractive but always means there is a merge to do at the end
How many different ports can there be?
## OS porting: Unices...

<table>
<thead>
<tr>
<th>Solaris (Intel &amp; Sparc)</th>
<th>Interactive Unix</th>
<th>Linux: RedHat, SUSE, Mandrake, Caldera, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Irix</td>
<td>UnixWare</td>
<td>AUX (Apple)</td>
</tr>
<tr>
<td>HP-UX</td>
<td>System V</td>
<td>Apollo</td>
</tr>
<tr>
<td>AIX (IBM &amp; Bull)</td>
<td>BSD – original, Free, Net, Open</td>
<td>Sinix (Siemens Nixdorf)</td>
</tr>
<tr>
<td>Digital Unix / TruUnix</td>
<td>Ultrix</td>
<td>Data General, ICL, Acorn, Olivetti, etc.</td>
</tr>
<tr>
<td>VX-Works</td>
<td>NextStep / Mach</td>
<td></td>
</tr>
</tbody>
</table>
Microsoft & Windows.....

<table>
<thead>
<tr>
<th>Windows</th>
<th>OS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows (1)</td>
<td>NT 3.1 / 3.5</td>
<td>PC-DOS</td>
</tr>
<tr>
<td>Windows 286</td>
<td>NT 4.0</td>
<td>MS-DOS</td>
</tr>
<tr>
<td>Windows 3.x</td>
<td>Win2000</td>
<td>DR-DOS</td>
</tr>
<tr>
<td>Windows 95/98/ME</td>
<td>XP</td>
<td>Free DOS</td>
</tr>
<tr>
<td>PenWindows</td>
<td>OS/2</td>
<td>GEM</td>
</tr>
<tr>
<td></td>
<td>CE 1 / 2 / 3</td>
<td>CP/M 86</td>
</tr>
</tbody>
</table>
And the others….

<table>
<thead>
<tr>
<th>PalmOS</th>
<th>BeOS</th>
<th>VMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacOS</td>
<td>OS-X</td>
<td>MVS</td>
</tr>
<tr>
<td>AmigaOS</td>
<td></td>
<td>OS/360</td>
</tr>
<tr>
<td>Geo interactive</td>
<td></td>
<td>OS/400</td>
</tr>
<tr>
<td>NewOS</td>
<td></td>
<td>Multics</td>
</tr>
<tr>
<td>EPOC</td>
<td>Plan-9</td>
<td></td>
</tr>
<tr>
<td>Symbian</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Databases...

<table>
<thead>
<tr>
<th>Database Type</th>
<th>Database Name</th>
<th>Database Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle 7/8/9</td>
<td>PostGres</td>
<td>xBase</td>
</tr>
<tr>
<td>Sybase SQL Server</td>
<td>MySql</td>
<td>ISAM</td>
</tr>
<tr>
<td>Microsoft SQL Server</td>
<td>Informix (rip)</td>
<td>Versant</td>
</tr>
<tr>
<td></td>
<td>Ingres (rip)</td>
<td>Poet</td>
</tr>
<tr>
<td>Db2</td>
<td>InterBase</td>
<td>Access</td>
</tr>
</tbody>
</table>
Database can be more difficult

Database ports must consider:

• Connectivity : ODBC? Db-lib? Oci?
• SQL : extensions are common
• Database architecture :
  – row locking or table locking?
  – transactions supported?
Old platforms still count!

• Not only do we want to move our products to the new platforms……

• …. we also want to revive old products which may be stuck on an old OS or database
Microsoft <=> Unix

• The most common type of port
• The one we are most likely to encounter
• Focus on this for the rest of the talk
• (But not exclusively)
## Cultural differences

### Microsoft world
- GUI tools
- IDE
- Client
- Crashes a lot
- Suits
- MSDN
- Get the job done
- Role models: Bills Gates & Steve Ballmer

### Unix world
- Command line tools
- Vi
- Server
- Runs forever
- Beards and sandals
- Open Source
- Obscure diversions
- Role models: Bill Joy & Richard Stallman
Culture differences cont.

• Vast generalisations but some element of truth
• And we characterise each other in these terms
• Lead to very different approaches to problems
• Can lead to friction within teams
• Technically the differences are much less, e.g. no platforms is really Posix complaint! (Those in glass houses….)
• Differences can lead to very different use case scenarios
C++ and porting

• C++ is portable
• Although “modern C++” is less portable
• “Traditional C++” is more portable
• ISO standard not implemented so what subset do we use?
• Comes down to compiler….
• But we could lapse back to C
• Or even K&R C

I assume you have an existing C++ product you wish to port
Tools

- Tools play an important part in porting
- The three big ones
  - Source Code control
  - Make
  - Compiler
## Choice of Compiler

<table>
<thead>
<tr>
<th>Native compiler</th>
<th>Common compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>• It is available</td>
<td>• Is it available?</td>
</tr>
<tr>
<td>• Optimised</td>
<td>• Same support on all platforms</td>
</tr>
<tr>
<td>• Lowest common denominator</td>
<td>• Are all elements supported on all platforms?</td>
</tr>
<tr>
<td>• What standards?</td>
<td></td>
</tr>
</tbody>
</table>
If you go for Native compiler....

- How low do you go?
- What libraries are you going to use?
- What tools can you use?
- No exceptions, no namespace, no STL, .....
Some Mozilla coding guidelines for portability

See http://www.mozilla.org/hacking/portable-cpp.html

• Don't use C++ templates.
• Don't use static constructors *(what?)*
• Don't use exceptions.
• Don't use Run-time Type Information.
• Don't use namespace facility.
• Put a new line at end-of-file.
• Always have a default constructor.
• Don't put constructors in header files.
• Don't use return statements that have an inline function in the return expression.
• Don't use mutable.
  … and lots more…..
Common compiler…. 

• Will it support all your platforms?
• Common compiler usually means GNU 
• Tools? Libraries?
• Still have differences in platforms e.g. endian, sockets, etc.
There are two types of code: common and divergent

• Our aim is to have common code on all platforms
• But some code must be different e.g. `WsaStartup();`
  Windows (WinSock) specific call that must be issued before using sockets.
  Has no equivalent in Unix.
How do we mark divergent code?

• Lets cut to the chase…. it comes down to macros!
• We can hide it in different files
• We can hide it in templates
• But it seems impossible to avoid macros
# Macros control

<table>
<thead>
<tr>
<th>One macro does all</th>
<th>One macro per feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>• New platforms require revisiting all <code>#ifdef</code>’s</td>
<td>• Complex make files</td>
</tr>
<tr>
<td>• Simpler to understand, test, debug</td>
<td>• Complex interactions</td>
</tr>
<tr>
<td>• Best for applications with only few platforms</td>
<td>• Easier to add new platforms</td>
</tr>
<tr>
<td></td>
<td>• Harder to maintain</td>
</tr>
<tr>
<td></td>
<td>• Best for libraries</td>
</tr>
</tbody>
</table>
Force the differences down…. 

- Higher levels should deal with application logic 
- Lower levels with platform/compiler logic 
- Keep the program flow obvious 
- Keep the program readable 
  (Both easy to loose with lots of macros in use) 
- Abstract away the differences 
- Application code is just that: application code
Keep the macros at the bottom

• It should be possible to keep all platform dependencies at the bottom
• All platform macros are at the bottom
• Higher level files never include system files
• Because system functions are wrapped
• Even keep .h files free of all system includes
Physical design of application

• Classic three tier model….
  
  Presentation
  Application logic
  Storage, utilities and interfacing

• Model appears again and again
• Usually see more layers today
• The importance of layering…..
Layering....

• A layer is one or more modules
• Layer should be self contained
• Depends on layers below
• Is depended on from above
• Does not depend on layers above or equal
• Package modules as libraries: .dll, .so, .a, .lib
• Exposes a number of header files
Layering looks like

- Presentation
- Application logic
- System level functions- storage, utilities, interfacing
- OS & compiler libraries
Layering is a way of looking at dependencies

• Dependencies must be controlled in any system
• Few dependencies the more solid the code - unlike a brick building
• Especially important when porting
• Good layering is modula code, high cohesion, low coupling
Dependencies

- Shortest route to the top has the fewest dependencies
- What are dependencies?
  - Include files
  - Lower layers
  - Libraries
Shortest route to the top

• Applications are a series of modules
• Deliverables are usually multiple applications (i.e. a set of programs)
• Port the minimum set of modules to get something - anything! - running
• Proof of concept
• Find out what you face
Some programs have more dependencies than others...

- Batch process
- GUI interface
- Maintenance tool
- Common business logic
- System level functions- storage, utilities, etc
- OS & compiler libraries
- Custom GUI widgets
Include files is where dependencies begin

• #include is evil
• #include is essential
• Only include what you need
• Include no more, no less
• Forward declarations can reduce includes
Includes....

• Three types of include
  – System
  – Project
  – Local

• Can further split Project

• Include ordering
  – most specific first
  – least specific first
Inline functions just say no!

• Inlines are for:
  – optimised code
  – libraries
  – templates (which usually libraries and when we have \texttt{export} things will change!)

• Premature optimisation can damage code
• Inlines increase dependencies
• Too many getter/setters indicate poor class design
Technical differences

• Shared libraries: differ on compilers and platforms
  – Initialisation
  – Memory models
  – Linking models

• Endian-ness
  – Not really an issue in porting
  – Is an issue in data transfer: why not use XML?

• Configuration
  – Registry, files, XML
More technical differences

• Resources
• Multi-threading
• Installation
• User access rights
• Scripts
• Process models
• Inter process communication…..
Technical differences: inter-process communication

- IPC looks different at first but actually has more in common
- NT & Unix both have:
  - shared memory
  - pipes
  - TCP
  - semaphores
Using C++ features

• Overloading functions
• Template code generation
• Standard library : increases abstraction
• Exception handling
  – helps force down differences
  – not all errors exist on all platforms so why check for them?
  – let the errors come to you!
  – difficult to retro-fit
Maybe in the future...

```cpp
#include <iostream>
#include <string>
struct Windows {};  
struct Solaris {};  
template<typename Unknown> std::string Foo() {
    return Unknown platform;
}

std::string Foo<Windows>() { return "Microsoft"; }
std::string Foo<Solaris>() { return "Sun"; }

typedef Windows CurrentPlatform;  // need a macro to change this…
int main() {
    std::cout << "Hello world!" << std::endl;
    std::cout << "The OS vendor is " << Foo<CurrentPlatform>() << std::endl;
    return 0;
}
```
... future...

• Since compiler do not need to compile template code which is not used, potentially we can rely on the compiler to remove platform specific code

• But this may force more code than we want into header files and inline functions
Problem with this future…

• We are starting to depend on some of the advanced compiler features which may not be available

• There are pre-processor issues which can not be delt with like this, e.g.

    #include <Winsock.h>

• Will never work on Unix...

• Brings us back to macros...
Closing remarks…. 

• Don’t be daunted by a port
• Look for the commonality not the differences
• Force the differences down
• Take the shortest route to the top
• Get a minimum port up and running fast
Keep doing it!

- Porting is an on going process
- You must now support it on all platforms
- Build early, build often, build always!
- Developers must be cross-platform
- May have platform experts
Alternative approaches

- Separate source code trees: now you have two project!
- Branch and forget: assume you are never going back
- Emulation: OK in the short term or where performance is not an issue
- IBM “Affinity” / Caldera (SCO)
- Outsourcing: make it some elses problem!
What I haven’t mentioned...

• GUI porting
• Wide character issues
• Project structuring (see article 1)
Remember to put these slides on my web site.....

www.allankelly.net