

## Porting part 3 : Ways forward

Porting is a big subject. Witness the fact that I sat down to write a couple of thousand words on the subject and here we are in a third article and still I'm thinking of things I should cover!

Even if you never need to do a port an awareness of the issues is important. Firstly, someone else may need to port your code so a little bit of thought up front may payoff in the long run. Secondly, porting is like travelling, it widens your horizons, why do other platforms do things different? can you live without a message pump? It imposes extra constraints on software which usually improve the quality because they force us to think more like engineers and less like hackers.

In wrapping up here I'd like say a few words about database ports, as I said in the first article this can be as big a challenge as an OS port. After that I want to mention a few more issues and solutions you should think about in a porting project.

### ***Database ports***

Porting between database can raise as many issues as a port between OS's. The issues fall into three main areas:

- ? Database architecture : before you even start to look at your database connectivity and SQL it is worth auditing your database usage. Do you rely on row-level locking? Will table level locking cause problems? How extensively do you use stored procedures, triggers and constraints? These are all areas where databases can differ at a fundamental level. For example, Sybase and Microsoft implement the concept of multiple "databases", within any database installation (i.e. server install with allocated disk space, etc.) you may have several different databases with different schemes; Oracle provides a similar concept with "tablespaces" but these differ significantly in implementation and the syntax used to access them.
- ? SQL and extensions : although there has been an ANSI standard SQL since 1986 it is heavily extended by vendors. If your SQL is to be truly portable you must stick to just the common sub-set of facilities provided by your databases. In particular this can have problems in areas such as stored procedures, triggers and string handling. Even where two databases provide the same services you may find that the syntax is different. One particularly painful area is that of types: beyond the basic types of integer, string, float, etc. all modern databases provide a rich variety of types but these will certainly differ from database to database.
- ? Programmatic communication to the database : C++ has no native way of talking SQL but at some point our code must communicate with the database. Vendors supply APIs for this, e.g. OCI for Oracle, db-lib for Sybase and Microsoft SQL Servers, although Sybase have moved to Open Client-library (ct-lib) and Microsoft prefer you to use ODBC. Needless to say these are incompatible, there are even minor differences between db-lib from Microsoft and Sybase.  
If your system uses the API directly the most obvious answer to these difference is adding another layer of abstraction. I described in Overload 41 how an application may be "jacked up" and a custom layer inserted to provide portability. You may also consider third party products such as RogueWave although these imply cost issues.

Many applications use *embedded-SQL* for database connectivity. Embedded-SQL allows you to write SQL statements in a C/C++ source file and have variables *bind* to database values. A vendor supplied pre-processor then translates your source code file into one which access the database using the vendor's API layer. This in effect gives you your extra layer of abstraction and because there is an ANSI standard for embedded SQL porting should, in theory, be easier.

However, embedded SQL is not suitable for every application, nor is it particularly nice to look at: programs can look confusing as the C++ and SQL are intermingled in the source. Further, you must avoid any vendor specific enhancements over and above the standard if you to are ensure portability.

Of course many of these issues may be addressed by using ODBC – Open Database Connectivity. This provides a call level API and imposes some standards to handling SQL and architectural issues.

However, ODBC brings its own issues, not all the SQL and architectural issues are resolved and it is largely a Windows solution, while most database have Windows ODBC drivers available other drivers can be more difficult, and costly to get.

Database connectivity is an area where it is worth doing your homework. Some sites worth checking out are:

- ? ODBC for Linux and FreeBSD: <http://www.unixodbc.org/>
- ? ODBC for Unix: <http://www.openlinksw.com>
- ? RogueWave software: <http://www.roguewave.com>

I'll also add a small disclaimer here: its a couple of years since I looked at this in great detail so things may of changed. I don't believe anything I've written is wrong, but things change.

Finally, in database intensive applications the performance and optimisation of a database can make or break and system. Tuning an Oracle database is very different to tuning an Informix database which is itself very different from tuning mySQL. There are two ways to tackle database performance: tuning the SQL statements and tuning the database tables, e.g. configuring table layouts, cache sizes, disc access and so on.

If you are lucky, you will be able to achieve satisfactory performance from just server tuning. More likely, you will want to do some SQL tuning. Both of these routes are highly vendor specific and in the case of SQL tuning you may be forced to implement vendor specific code modifications to maximise your performance.

### ***Compile on all platforms and keep doing so***

A change on one platform should always be compiled on all platforms, even if the code appears innocuous you should at least compile it on your other platforms. Function calls may differ slightly, an extra underscore in the a function name, or parameters may differ – an int on one platform may be a short on another, or maybe you've used a new variable which just happens to be the name of a macro hidden in some system file on another platform.

When you first port you probably want to test each set of changes immediately on the other platforms to deal with any problems before you check in. In this case it is often best to share a directory between

the two environments, however, this can seriously lengthen compile times, not to mention debug times if the debugger must load a large symbol table across Samba or NFS.

Alternatively, you could choose to check in the files you have changed and immediately check them out and compile them on another platform. While this reduces compile time you can find yourself stuck in an seemingly infinite loop of: platform A: check out, test, change, checkin; platform B: checkout, test, change, checkin, repeat on A. And your change log fills up with “Not yet compiled on NT”, “Fixed to build on Unix”.....

Once your porting effort starts to reach something vaguely stable it is worth setting up a nightly batch build on all your target machines. Apart from the usual advantages this confers on a project it can help ensure that your changes are building in all configurations on all platforms. The number of configurations quickly rises when you start porting: an application ported to Solaris and NT for Oracle and Sybase gives you four build combinations, if you build debug and release you have eight. Port to AIX and you have 16 combinations to check! Only an automated build system can ensure you are not introducing problems.

As well as saving time and effort automating build also helps developers who are not so familiar with a given platform, or who may be tempted to skip building on platform X.

### ***Quick list of other things***

When you have a source code control system that understands your platforms, neutral compilers and makefiles you still face many porting issues. Here is a quick list to get you started:

- ? Case sensitive filenames: Microsoft doesn't care if you name a file Foo.cpp, foo.cpp FoO.cPp or anything else, but Unix seems these as distinct files. The simplest thing is to mandate that all filenames are lowercase, if this doesn't appear than take care.
- ? Algorithm for finding include files differs: as I mentioned in my articles on include files (Overload 40, December 2000) the search path for your include files differences from compiler to compiler. Decide on a directory structure, keep it simple and keep it common to all platforms.
- ? Text files with over 79 characters per line: I've never been one to shy away from a description function name just because its a little long, certainly, in graphical IDEs there is an increasing tendency to regard the 80 characters per line limit as a thing of the past. But, for developers using standard Unix tools such as telnet, vi and Emacs this is still a consideration.
- ? Shared libraries aka DLLs: not only does DLL library handling differ between Windows and Unix but it differs between Windows versions (entry at LibMain or DllMain?), Unix platforms differ in their handling of shared objects, and difference exists from compiler to compiler, Sun C++ lets you specify start-up functions with #pragma init, but GNU C++ uses \_\_attribute\_\_((constructor)). Even at build time there are differences, Windows (Visual C++) links against stub functions and resolves all externals at link time, while Solaris (GNU C++) has a full blown run-time linker and only attempts to link functions at run time, so you may have an unresolved external and not know about it until your code executes.

- ? Debugging can be a very different experience: few debuggers measure up to the graphical ones available on Windows. The differences with shared libraries be particularly difficult as symbol information may not be available until part of the run is complete.
- ? Multi-threaded: bog standard mutli-threading on different platforms is best handled using some kind of abstraction layer, your own or bought in. More advanced multi-threading is best avoided altogether. Again, Windows differs to Unix, Windows itself ranges from none on 16-bit to advanced on Windows 2000 but features may be missing on '95 and family. Unix has some standardisation with Pthreads, but Pthreads is not as rich an environment as, say, Solaris threads. Given the potential for very complex problems with threading the best advice is to Keep It Simple. This should keep you portable.
- ? Installation : Windows users expect a graphic install system. Unix users are accustomed to text only scripts for installation. Before you decide to force this on Windows users recognise that the Windows batch language is a pale shadow of the rich Unix script languages. As for installations to Epos or Palm things are different again. This is one area where you will probably have to live with the difference. When making your first impression adhering to platforms conventions is important so as not to scare people off.
- ? Backslash/forward slash: Microsoft OSs use back slash, “\” to separate elements of a path name, while Unix uses forward-slash “/”. Other OS have been know to use other things but fortunately these are becoming less common. NT will happily accept “/” in system calls but Unix is not so forgiving and regards “\” as a valid filename character. This may seem a small and trivial things but I recently wasted half a day because one piece of code had been changed and correctly created someFile in someDirectory (someDirectory/someFile) but another piece was trying to open the **file** someDirectory\someFile.
- ? Character size: a character is one byte but how big is a wide character wchar\_t? On NT a Unicode wide character is 16 bits (using the UTF-16 encoding) but on Solaris uses the UTF-32 encoding so each wide character is 32 bits. Not only this, but NT will allow system calls with wide or narrow characters but Solaris demands narrow characters for all system calls. To make matters worse, Microsoft has not implemented Unicode on the home OSs like '95 and '98.
- ? Endianess: although this is normally at the top of people’s porting issues list the reality is that unless you are twiddling the bits on your machine, or passing data between different machines endianess is not a major issue. High level C++ code which is communicating with like systems may even avoid the issue altogether.
- ? The CR-LF problem: another seemingly trivial problem: Microsoft OSs terminate lines with a carriage return, line feed character pair (10, 13) while Unix uses only a line feed (10), and Macs, I believe, just use carriage return (13). At runtime this can cause problems reading and writing datafiles. At development time it can play havoc with source code files, Unix *vi* and *more* will show carriage return character (^M) while Emacs will hide it but retain it. Visual C++ will happily accept files which use just a line feed, but will insert a CR-LF pair on the lines you edit. Things get worse if your compiler chokes on extra (or missing) characters. Fortunately most compilers in my experience are forgiving, but GNU C++ has a problem if you use the macro

continuation character “\” which is followed by a CR-LF pair – it decides the next line is blank and chokes on the following line which is the macro continuation!

So far I’ve made Visual C++ sound good here: the editor and compiler both understand the problem. You could almost live in a LF terminated world. But, the built-in make system doesn’t. If your .dsw and .dsp lack the CR-LF pair the IDE will consider them corrupted.

- ? Signals and events : Within the Unix world signals differ a little with each platform but fall into two main groups: BSD derived signals, and System V derived signals. NT implements a subset of Unix signals. In addition NT also supports a concept of events which have some similarities but are much more flexible and easier to program, and consequently, present difficulties when NT code is ported.

## **Solutions**

Those of you who have read the two previous articles may be wondering by now when I’m going to stop talking about the problems encountered in a port and start presenting some hard solutions. So far I’ve tried to present strategies for approaching the issues and side-stepping them. Actual solutions are more difficult because these frequently depend on your objectives and business model. However, here are some solutions I’ve found useful in the past:

- ? Use language features to express difference : suppose your configuration details may be stored in the Windows registry or an Unix configuration file. The mechanics of this difference can be forced down to a low level, at the application level code can deal with an AbstractConfiguration class which is implemented by either a RegistryConfiguration class or a FileConfiguration class. Inheritance and a class factory can be combined to hide this detail.
- ? Overloaded functions can help greatly with porting. Where a data type is different on two platforms you can provide two similar functions to process it differentiated by the type. In many cases an overload need only perform some data conversion and call the original function. For example, consider the case of opening a file, on NT this will take a wchar\_t string while on Solaris it must take a narrow character:

```
void ProcessFile(const std::string<wchar_t>& filename) {
    #ifdef _WIN32
        FILE *f = fopen(filename.c_str(), _T("w"));
    #else
        FILE *f = fopen(Narrow(filename).c_str(), "w");
    #endif
    .....
}
```

If we introduce the function ApiString which always returns a string of the type required by the OS, and provide several overloads for say, char\*, wchar\_t\*, std::basic\_string < char, ...>, std::basic\_string<wchar\_t, ...>, etc. we can re-write the code as:

```

void ProcessFile(const std::string<wchar_t>& filename) {
    FILE *f = fopen(ApiString(filename).c_str(),
                    ApiString("w").c_str());
    .....
}

```

Here, the platform difference is forced down, making the application code easier to read and the developer is relieved from needing to handle platforms differently. In addition, if we add a Linux platform we only have one place where the code needs to be changed.

At best ApiString may be an inline function which simply returns the supplied argument allowing the compiler to optimise it away. At worst, a conversion takes place which is needed anyway. Also, by returning a std::basic\_string memory allocation issues are avoided.

- ? Stay consistent with yourself even if you break consistency with the OS: thinking about the configuration examples again, why should different platforms be different? Only because there is a convention in Windows of storing configuration in the registry. Before the registry we used configuration files, and anyway, most people curse the registry. So why not make the Windows version use a configuration file like the Unix version? This will also save effort on the part of your technical writing team.
- ? Make a virtue out of commonality: so I haven't convinced you to give up the registry for the sake of commonality? What is, the configuration file is in XML? The Windows registry doesn't support XML data so it makes sense to have this in a file. Not only will this provide common code on all platforms and make your tech writers happy but you can add another buzz word to your product information sheet.
- ? Keep your interfaces the same: this is an essential tactic in forcing the differences in your system downwards. If the BeOs and Solaris versions both provide function Foo there is no point in making the signatures different. This will only force the client code, higher up the application, to use an #ifdef to decide which to use. Even if you need to pass an unused parameter on some platforms you have pushed the difference down.
- ? Use common interfaces : frequently we have a choice of API function calls to make, always choose the more available one, usually the more standard one, even if you need to write a little extra code. You will need to write the extra code for those platforms that don't support the less standard one so why complicate matters with separate code paths?
- ? Widen the pre-condition, narrow the post-condition: this comes from the field of formal methods so stay with me for a moment. Even if you don't write pre and post conditions for your functions they have them. Before any function will operate correctly there is an implicit pre-condition even if it is not specified. Likewise there is an implicit post-condition.

For any function, F, it is always possible to widen the pre-condition of a function. This is because any function calling F will comply with the current pre-condition. Likewise, a similar argument can be made for narrowing the post-condition, in the new post-condition all values remain valid return

values for the old post-condition. You can always enhance your code so it deals with the existing set of values and then some more.

This comes in to play when porting because it is possible to take an existing function and change it so it will operate on a new platform provided you only widen the pre-condition and/or narrow the post condition.

## **Resources**

In truth, vendors are usually keen to see you port your application to their platform. Some will even loan you a machine, or give you access to a porting lab. Even if they are not this genours most provide some resources for firms looking to port their software.

Sun have some useful information on porting NT application to Solaris at

<http://soldc.sun.com/ntmigration/migissues/>. Moving from Solaris, RedHat have a Solaris to Linux porting guide: [http://www.redhat.com/devnet/whitepapers/solaris\\_port/book1.html](http://www.redhat.com/devnet/whitepapers/solaris_port/book1.html). ( A quick search of the Microsoft web site reveals several papers on interoperability but nothing with porting in the title. )

The Mozilla project has an interesting set of guidelines to portable C++ coding:

<http://www.mozilla.org/hacking/portable-cpp.html>. Unfortunately, these rule out some of the modern coding idioms: don't use exceptions, don't use namespaces, don't use RTTI, get the idea? Even if you don't use these rules they give you an idea of what kind of problems exist.

If you are porting between Unix and NT a good Unix like shell on NT can be a great help. While I've normally used MKS Toolkit (<http://www.mkssoftware.com/>) I'm currently using the Cygwin (<http://www.cygwin.com/>) development tools. Using Cygwin for the Bash shell and tools is fine, but before you use any of their porting layer check the licensing terms.

IBM have another approach to porting. They are extending AIX so the API is Linux compatible. This is an interesting approach, we developers can develop on cheap Linux boxes, but IBM can still sell AIX to the big corporation for the data centre. More details at of "Affinity" at

[http://www.ibm.com/servers/esdd/articles/aix\\_linux.html](http://www.ibm.com/servers/esdd/articles/aix_linux.html).

Actually, if your think about it, both Linux and AIX are Unices and should be compatible from the word go so we have come full circle!

## **The End.....**

Hopefully these articles give those of you who have never had the pleasure of porting code now have some idea of what is entailed, and for porting veterans I hope I've given you some new ideas.

Once you have ported code you view custom solutions, how ever small, in a different light. I actually think that aiming at multiple platforms improves code quality: it forces you to address issues which you may otherwise brush under the carpet, and it forces you to produce code which is understandable by different tools and programmers with different backgrounds.

(c) Allan Kelly, 2001