Porting part 2 : Addressing the differences

You don't want to do that!

Of course, we would all like porting to be a non-issue. In an ideal world we build our products for multiple platforms. In the real world we can't see the future: we don't know which database vendor will become dominate in our field, or which OS will become strategically important to the company, nor do we know that the company will be taken over in a year and that as part of the IBM empire we must support OS/2! Even when we have the opportunity to design for many platforms the issues we face are the same as porting an existing project.

When we design we make trade-offs. If we didn't we would spend too much time catering for things that never come to pass and we would never ship product. Perhaps the biggest trade-offs are in terms of our platform. Unfortunately even here events can over take us, this makes porting a necessity.

One of the effects of this is that porting often involves older platforms and/or competing technologies. As such it can have us trying to remember stuff we long ago purged from our cache, or providing us with a nice way to update our skills.

Last time I looked at some technical and logistical strategies for porting, this time I'd like to look at some issues which you typically encounter en-mass when you port between OSs.

Cultural differences

Different platforms inspire different cultures, which in turn leads to different ways of solving problems, the most obvious example is the difference between Unix hackers who first think of solutions using scripts and command line tools, while Windows people think of point and click solutions and you may find the Sybase way of doing things is not the Oracle way, which is not the Informix way; while an embedded developer has executable size uppermost in mind, many server developers don't even know the size of their application.

None of the culture difference come close to that between Unix and Windows thinking. Windows developers have meaningful call names such as GetTempFilename, nice looking GUI tools, API's measured in thousands of functions and worship Bill Gates, in comparison Unix developers have calls with slightly cryptic names like mktemp, use a windowing system to run several command lines at once, may use micro-kernels with half a dozen functions and trust in Richard Stallman.

These differences, and perceived differences manifest themselves in many ways, from choosing a function name to a willingness to use GNU software; views on what is an acceptable mean-time between failures, or an acceptable user interface. The perceived differences are as important as actual differences, if any of the stereo-types I've just sighted have left you saying "hang on there" you've just proved my point: you, a Windows developer, may be far more concerned with MTBF than your Unix programming colleague who

sits next to you but many people believe Windows programmer care less about programs which crash than Unix developers.
The media don't help this situation, the casting of Good v. Bad, Microsoft v. Sun, C++ v. Java can lead to real tensions as individuals or groups see themselves in this grander battle.
When it comes to choosing tools these culture difference can cause significant problems. Windows developers may refuse to adopt a source code control system that does not integrate with Visual Studio, while Unix developers demand a command line tool.
Ideally you want all developers to be productive on all platforms. Few people will be equally experienced and equally happy on all platforms but you can lessen the differences. Allow developers to be moved freely between the platforms will help them grow as developers and understand the system better.
Hopefully, both groups will learn to get the best from both worlds, there should be no demarcation lines based on OS, but this means helping and educating people – or facilitating and empowering if buzz words are your organisations thing!
There are countless ways in which the two cultures clash and the whole area is probably worth a PhD in sociology. Both cultures should be equally valued. In a traditional Unix environment the addition of a couple of Windows developers to port the product may leave the Unix developers feeling betrayed, particularly if a Windows port is undertaken instead of a Linux port, equally, the Windows developers may feel under-siege from more established developers who show animosity to Microsoft products.
Equally valuing both cultures means accommodating both and trying to get the best from both.
I would go as far to say that culture difference between Microsoft centric people and Unix centric people are greater than the technical differences between the respective OSs.

Tool chain

Tools play a big part in allowing developers move easily between different platforms. The compiler, make system and the source code control system are covered below, and while these are the three most important tools there are others tools. Where possible it is best to choose tools which are common to all platforms but this is not always possible, or can be prohibitively expensive.
Many small tools are which are standard on some platforms are simply not available on others or buried inside and IDE, e.g. grep. While we can always re-write our scripts and automation tools to avoid these tools it makes life harder. More importantly, there is a human factor, developers must know two sets of tools and context switch between them. Far better then to invest in portable toolsets such as MKS Toolkit and portable GNU tools (such as Cygwin) which can provide similar services on different platforms.
Hopefully this should common tools will replace several learning curves with one and simplify maintenance, although in practice this means write once, test many.
Providing similar tools on all platforms can help overcome cultural problems too: a command line debugger can feel as alien to a Windows developer as the Cmd shell feels to a Unix developer but install a graphical debugger like GVD on Unix and a Korn shell on NT and much of the alien landscape looks familiar.

Sometimes platform neutral tools can be best. Here many Open Source or GNU tools are useful. Want a scripting language? Python and Perl these are readily available on many platforms and are free of the historic baggage that attaches to Bash and Cmd.

Editors

Editors tend to break the commonality rule because editors are highly individual. Developers directly and indirectly invest a lot of time in getting to know their editors. Those brought up on IDEs will find their first encounter with vi traumatic, however, the reverse can also be true. While IDEs may be highly platform specific the most popular editors like vi and Emacs are available on most platforms and it's probably best to allow developers to use which ever they are happiest with, at least initially.

Compiler

Porting from compiler to compiler can be a bigger job than porting from OS to OS. Unfortunately, porting from OS to OS frequently involves changing compiler too – sometimes this means different version of the same compiler. When you port between OS you have a choice:

? Use the native compiler for each OS : sometimes your only option is the native compiler simply because no one else has the time or experience to port their compile - if you want to support Interactive Unix you choices are very limited. Sometimes the native compiler can be the best options, you can expect better code optimised and vendor support.

? Use some common compiler on all OSs: this is a good option if you can do it. Normally, the common compiler is GNU C++, also know as gcc and g++[1]. While you can expect the language subset supported by the compiler to be reasonably consistent there may still be differences, for example, in 1997 I has problems with GNU exception handling which was not fully supported on all the platforms I was using, I don't know the current level of support, hopefully 3.0 has resolved this issue.
If nothing else, even where the language subset differs you can at least expect common command line options. When using native compilers your makefiles must accommodate vast differences in flags from platform to platform.

? Use native on some platforms (typically Visual C++ on Windows) and common elsewhere (typically GNU on Unices) : surprisingly this is a common configuration for porting environments. In part it reflects the culture differences, Microsoft compilers are relatively cheaper than their native Unix counter parts and have a big native compiler advantage.

---

[1] The GNU compiler suite provides two C/C++ compilers: gcc and g++. gcc will run as C or C++ depending on the filename, i.e. foobar.c is C while foobar.cpp is C++. However, it is better to use g++ for C++ programs because, although it calls gcc for the actual compile, g++ will link perform additional actions for C++ such as linking in the C++ standard library. For more information see: http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_3.html#IDX37

Many server products have graphic client tools written in Visual C++ with MFC which are never ported. By keeping the server in VC++ there is commonality with the client tools even if all the Unix server ports compile with GNU.

Although I've talked about GNU C++ as the portable compiler of choice it is not the only compiler available on many platforms, others do exist, CVu 13.3 (June 2001) carries a piece on Comeau C++ which can claims to combine the power of a portable C++ compiler with the features of a native compiler. However, GNU C++ is be far the most used for this kind of work. In theory, if it does not exist for your platform you could port it. In reality, you project deadline probably won't allow you to take time out to port a compiler too, but, you may be able to sub-contract this work if, for example, you are attempting to port to a new platform where no suitable compiler exists.

Finally on this subject a word on linkers. Once upon a time linkers where separate tools and would deserve a section on their own. Increasingly they are part of the compiler so you have no need to decide on compatible linkers.

Make and makefiles

Make systems may do more to isolate Windows developers from the rest of the developer community than any under single item covered here. (Why does this remind me of the infamous British newspaper headline: "Fog in channel, Europe isolated" ?)

Until version Visual Studio 5.0 Microsoft supported make: OK, they used their own "nmake" and some syntax differed to standard make, and version 4.0 automation of makefile generation made them unusable on any other platform. In version 5.0 Microsoft did the honourable thing and ceased to support make as their default build mechanism. Instead they produce .DSP files for projects, and group these with .DSW files. The syntax in these files is similar to make but different.

You can still build a project on Windows with good-old make, or nmake, but, and this is a big but to Windows developers: you loose all sorts of nice things that Microsoft have added to make working in the IDE nicer and integrate with source code control system.

Whether you, and your developers, can break from Microsoft's make system depends on your organisation its culture, and people's willingness to change. Many companies choose not to fight these battles and accept instead two make systems, one on Windows and one on the other platforms.

(I'd like to ignore the VC++ option to "export to makefile" but I'm sure some people are thinking about it. I'd like to ignore it because it generates a nmake makefile which may not work with other make programs, and, the file is oriented towards the Windows project. Finally, it is really a once only export, you cannot change it and reimport it, nor can you use it as the basis for Unix makefiles.)

One the other platforms the situation is slightly easier. Most vendors supply their own version of make with the OS, the syntax and capabilities of this differ. Makefiles are also notoriously difficult to debug and coding ifdef's in the makefiles complicates them by orders of magnitude.

Luckily the best make program around is also the most portable, namely GNU make, also called gmake. The GNU developers have made enhancements to the make syntax to simplify the files.
The design, implementation and debugging of makefiles is worth of an article in its own right. By using a common make system you get two benefits: first you do not need to overhaul or create a new make system for each platform, secondly, it enables you to very quickly get builds running on a new platform so you can quickly assess the forthcoming work.

Source code control

The subject of source code control system arises regularly on the Accu-General mailing list so before deciding on anything have a look through the archives and see what peoples experiences are.
When it comes to porting it goes without saying that you want a system that will understand all your platforms and will enable you to check out and checkin from where-ever you are. At the simplest level almost every system can do this even if it means mounting a drive on a machine which runs the system – a not uncommon approach where Visual Source Safe if used with Unix projects.
However, there are several issues which should be considered in a cross platform environment:

? Usability: as mentioned above, there is almost always a work around but is it really usable? Is it error prone? Can it square the circle of being consistent with itself and the platforms it runs on.
? Merging, branching and multiple checkouts: porting projects tend to need more merge intensive than regular maintenance. Likewise, if porting is proceeding in parallel with development branching and multiple checkouts may be more common occurrences.
? Does the system understand CR-LF? The CR-LF problem deserves a section in its own right. A source control system which can perform this conversation automatically can save a lot of trouble.
? Are client tools available (and affordable) on all platforms being used? If not you may need two machines, one for development and one for checkin-checkout. Again, this increases the risk of mistakes and accidents. The affordability of tools may be more important than availability – the Windows version of VSS seems to come free with a packet of cornflakes, but the Unix versions costs money and seldom seems to get bought.

OS ports

I would hazard a guess that most OS porting activity at the moment involves at least one of Solaris, Linux or Windows NT/2000 either as the originating platform, or the target. However, there are many more OS ports which may be required – I'm sure many MacIntosh developers are busy porting to OS-X as we speak. Even if we confine ourselves to Unix derivatives I can think of 20 different versions, OK so maybe only half a dozen are currently receiving significant resources but that just mean there are 14 less-known Unix which people may want to move software from.
(Actually, the number of Microsoft systems is larger than you may think: MS-DOS, several 16 bit flavours (3.1, Workgroups), several hybrids ('95, '98, ME), the NT family (3.5, 4.0, 2000, XP), not forgetting CE

(I'm told 3.0 is quite different to 2.0) and then there several close cousins: Pharlap, OS/2 (yes, it still lives, never underestimate the legevity of IBM products) and even Sun's WABI system.)
Sometimes porting between Unices can be a significant issue in it's own right. Even within the Unix world there can be significant differences between BSD based Unices and System V based Unix, in recent years these differences have lessened as System V is now the basis for most Unices.
Another problem with the lesser known Unices is the lack of tool and library support. In one case I know of the company insisted that Interactive-Unix was support, at the time (1995) this lacked even a ANSI-C compiler so all code was reduced to K&R style C. Fortunately Sun are pensioning off Interactive, and while several other lessor Unices are also disappearing (one hopes Caldera will finally kill/merge SCO and Unixware with their Linux) it seems we live in an age with more operating systems than ever – the latest I have heard of is NewOS (www.newos.org).
It is important to look for commonality in different OS rather than the differences, for example, remember that NT was designed to support Posix and to this end it actually contains a great number of standard Unix calls – although these are often renamed with a proceeding _underscore. This can be a great advantage as it allows exactly the same code to be compiled on both platforms. It is always worth checking the Windows document when faced with a Unix call that needs porting, the same call may exist is a slightly different form, with an preceding underscore, or slightly different arguments, it may be possible to get the functionality you require a hide it behind a thin wrapper. Unfortunately, the same is not true in reverse – the good news for Unix developers looking to create a software with a fancy GUI is that Apple's OS-X is based on BSD Unix, this may provide an attractive platform.
Luckily the standard C library is approximately the standard Unix library. While this may not provide all the functionality we require it is a starting point. In other areas like TCP/IP communication there are standard calls we can choose to use on multiple platforms. It becomes a case of discipline: avoid the propriety function calls.
As I said there are many more OSs than I can name here. These become can be very different and present more substantial problems than replacing a few function calls. Symbian's EPOC imposes very different porting restrictions because of it's active-object pattern and when porting to smaller, embedded OSs such as EPOC and PalmOS we may face problems which necessitate a major redesign.

Next time....

Proof reading this article I notice that I spend a lot of time talking about Unix and NT differences. This wasn't my intention, but reflects two facts: firstly, this is where most of my porting experience lies, secondly, that this is the big divide in modern OSs. Companies seem to decide Microsoft or Unix and only then, if the answer is Unix, decide which flavour. This applies equally to software houses and purchasing companies.
In the next article in this short series I'll look at database porting, some techniques for dealing with platform differences and present a grab bag of issues you need to think about.