Where to begin: Porting

What is porting?

Sooner or later we all contemplate porting software, for many this will not come to pass, it only takes a few moments consideration to know that porting your MFC application to VxWorks is not an option. However, until Java, Linux and Oracle succeed destroying to opposition it will remain a fact of life.

Usually when we talk about porting we mean taking a chunk of source code and making it work on another operating system, or another flavour of the same OS, e.g. making a UNIX app run on NT, or a Solaris app run on AIX. However, porting really covers the process of moving any source code base which relies on a particular API and making it run on another API. Commonly the API in question is an OS but others ports are frequently performed, for example, taking a Sybase application and making it run on Oracle is database porting. Here are a few more you may, or may not, of come across:

? Application server porting: iPlant, WebLogic, IIS you may need to change to another.

? Compilers: fed up of Microsoft's template handling? Move to Borland.

? Language porting: I once came across a company that between version 3 and 4 moved it's source code base from Pascal to C++ with the aid of the GNU p2c tool.

? Word size: the days of moving 16 bit Windows applications to 32 bits may seem long ago but we are about to enter the 64 bit age so here we go again!

? Localisation is a form of porting, particularly if it means your application must move from narrow characters to wide characters.

This list could go on and on, but you get the idea. I'd like to spend the rest of this article discussing a few strategies for porting. A future article will discuss porting issues such as tools and cultural problems.

Terminology

I tend to talk of porting between *platforms*, in most porting cases *platform* is a OS but it could equally be an database or a compiler. Further, most ports must deal with a number of issues; if you are porting your application from Solaris 2.8 to Windows 2000 you are likely also changing compiler. So when I talk of porting from platform-A to platform-B I'm talking about the bigger picture, platform A may be Windows NT with Microsoft SQLserver and Visual C++, while B is HP-UX with Informix and GNU C++.

Finally, most of my remarks are made in the C or C++ context here. Java programmers can kick back and feel smug if they wish, or they can think about their own porting issues: different JVMs, database issues outside of JDBC, different threading models, and so on.

Strategies

When porting there are technical and logistical issues. The technical issues concern API calls, error handling, bit tweaking and such; the logistical issues concern source code control, branching, backwards compatibility and such. In an ideal world we would be asked to port one parameter at a time, on an unchanging code base, with no need to support previous releases, no need to support multiple platforms going forward and unlimited time! Given this, we would simply label our source code "Last Windows version" and start work on UNIX.

In the real world the port is normally destined to run on multiple platforms: so today's NT program will tomorrow compile and run on NT and UNIX. This means we must come up with a cross-platform code base that we can maintain on both platforms.

In addition we often face the need to continue releasing on the current platform while the port takes place. For these reasons this means a good source code control system must be in place before any work begins. When porting don't leave home without source code control, if your porting across platforms make sure your source control understands different OS's.

Logistical tacts

There are as many logistical strategies for porting as their are platforms to port to. I'd like to outline several I've come across and the general style. The strategy that works for you may be different, indeed, most of these strategies can be mixed-and-matched.

Before discussing strategies there are two tenets I think are essential to a porting effort:

? You must be able to get back where you started from: if things go badly you should be able to throw away your effort, mark it down to experience and roll back.

? Aim for one source code base: at the end of the porting exercise you should to have one source code tree. If you have one source code tree for Oracle and a different one for Sybase you really have two applications to look after, and two places to make future changes

Given these two tenets there are several strategies available:

? Freeze and port: if you can escape doing any releases during the porting period then you have the option to freeze new development and port the source code base. This is the best option if it is available as it also allows you to concentrate your entire resources on the porting operation.

? Branch and port: an obvious variant on freeze and port, you branch your source code base in two and while one team ports the code another continues with new developments and maintenance. The problem here is that at some point in the future there needs to be a merge to return the two branches to one tree. Depending on the amount of work to be undertaken by each team this may be daunting, or it may be a short afternoon exercise.

? Creeping port: hopefully you have a well layered application which has several different modules, some libraries, some DLLs, some executables. Starting from the bottom it may be possible to "freeze and port" one module at a time. In such cases it is best to map out your layers and attack the shortest path from bottom to top which will give you a working, testable binary. Even the simplest executable can serve as proof of concept.

? Proto-type port: before beginning the port for real it may be worth attempting a *quick and dirty* port to discover some of the issues involved. This is a kind of branch and port where you don't intend to do the merge, indeed, you probably give up on source code control so you can move faster. This is worth attempting if you have never ported before and need to discover what you are up against.

One technique that does not work is "port and clean." This says: "we are going to port this code, and in the process we're going to clean it up and refactor it." The problem here is two fold; firstly the developers have mixed objectives, secondly, it makes it extremely difficult to reconcile the original code with the new code when problems occur. Don't take this advice to the extreme because often the best way to port a section of code is to refactor first, but making this an objective on it's own leads to problems.

If your worried about your source code quality then it is worth investing some time prior to the port in refactoring the code. This revised version can then be tested before the port begins. Hopefully the port will proceed more easily then. This can be called "clean, then port."

This brings us to...

? Back porting: This is an extremely powerful porting technique because it allows developers to focus on one objective at a time, forces cultural differences out into the open and can produce a ported version quickly – albeit at the expense of the original version.

  With a Back Port strategy you first port the code to a the new platform, e.g. from Solaris to NT. In the process the ported version becomes the main development branch and the original version is orphaned. Developers make a best attempt at supporting the original platform but don't lose time over it. Once the new version is up and running, even potentially delivered, developers then switch their attention to making it run on the original platform, this should not be a large job as it used to run on that platform.

It is worth noting that many of these techniques rely on "waves of porting." In each wave another step to a fully ported product is delivered. These can be fitted in with a release schedule so that over several releases the product is gradually ported.

Technical tacts

As with the logistical strategies for porting technical strategies are not exclusive. In any ported source code base you are likely to use most of the following ideas.

? Fat binary or several binaries: sometimes it is possible to support all your ports with one executable, e.g. on executable could contain the Sybase port and the Informix port, users could decide at run time which database is used, this would be a fat binary since all users would have code they do not need. Alternatively, you may produce different specialised binaries. This is often the case as it is not possible to bundle radically different platforms such as Linux and NT.

Abstract and port : write an abstraction layer which replaces dependencies on API-X with a dependency on your own *Abstract API*. You then write a concrete implementation of the abstract API which interfaces to API-X, the concrete representation is created through a factory. Next you replace your calls to API-X with calls through the abstract API, with this framework in place you can write alternative representations for API-Y and API-Z. This is almost the classic use of the factory pattern. I metaphorically think of this as lifting the application up, slipping in an abstraction layer and lowering the app onto this.

When porting with this method you can minimise changes to the existing source code by modelling your abstract API on API-X, however, this may increase the effort needed when you write the implementation to support API-Y.

Traditionally one would use a abstract base class for the abstract API and virtual functions to provide run-time polymorphism. An alternative technique would be to use the compile-time polymorphism provided by templates and instanciate specialised template members for the correct platform.

? Macros : in C and C++ macros are the most used porting tool there is. This does not mean we can forgive their sins, but we are forced to accept them to a large extent. There are two basic techniques with macros.

? Give each platform it’s own macro (typically a vendor defined one such as _MSC_VER) and use it to switch on and off the lines of code for each platform.

? Or, define macros for each feature and enable and disable code based on that macro. Then, for each platform decide which macros to define, much GNU code does this.
The first technique is easier to follow while the second if more flexible if a bit more obscure. As a general rule, if you are porting a library, or to a large number of platforms, then feature macros are probably the best way to go as the extra work will pay off as commonalties between different platforms become apparent. However, for ports of specific applications to a few platforms the simplicity of a platform macro is hard to beat.

? Force down the differences : code littered if #ifdef WIN32, or #elsif SUNOS becomes untidy and more difficult to read and hence maintain. It is therefore a good idea to force differences down to low level libraries and allow application code to stick to plain language and library functions. Typically, the lower level the function the shorter it is likely to be and hence, the amount of code and confusion should be minimised.

? Macros are not the only fruit: there are several alternatives all of which help force the differences down:

? Typedef : types differ from platform to platform, how big is an int? 16 bit in DOS, 32 bit on NT, 64 bit on some machines but Digital UNIX (now Compaq’s Tru-UNIX) decide to leave it at 32 bits for “compatibility reasons.” Typedef’ing a type can help isolate you from these kind of problems, particuarly when used with the next technique.

? Overloaded functions : in C++ it is possible to provide two versions of a function and allow the compiler to decide which one to use, e.g. foo(wchar_t*) and foo(char*). However, this technique only works where there both functions can be compiled.

? Different files : where two functions differ greatly it is possible to write two different implementations in different files and have the makefile include the correct file for the current platform. While this is a powerful technique my experience with it has been unhappy, I actually find having to look at two different files for the same thing confusing; additionally, when applied as the principal technique it leads to files which are nearly identical but with small differences.

? Third party libraries : sometimes it seems the world is full of vendors who will sell you a cross platform library. However, many of these turn out to be snake oil, you replace a dependency on a OS API with a dependency on a libraries API. And remeber to always keep a close watch on run time license costs for any library you use. Libraries which are cross platform are not usually a problem but those which set out to provide a common API frequently fail because of performance issues, or because they confuse developers. When starting a project afresh such a library may well work well, but fitting one to an existing application can be extremely difficult. As an exception, some libraries such as MKS NuT-Cracker are designed for exactly this role.

  It is also worth looking at the compatibility matrix: you can only port to the platforms a vendor supports and they can limit your choice of other libraries and tools. This can be crippling if, for example, library X does not work with the vendor's new compiler, the vendor may no longer sell the old compiler so getting licenses for new developers is impossible. Of course the opposite can also happen, tools available on one platform may well find faults that are common to all platforms.

  Open source libraries can avoid some of these problems but may lack support and force you to take on some of the maintenance yourself.

Pause for breath

I hope that gives you some idea of the problems faced when you come to port code. Next time I'll try and give you some idea of the issues involved with compilers, source code control systems, culture issues and the OS's and databases themselves.