Writing Easily Portable Code

by M.D.Geib Vista Control Systems, Inc.

Introduction

This paper discusses some of the issues related to producing portable software.

When a software system is ported from one platform to another the following logical steps are normally taken: the source files for the system are moved or made accessible on the new target system; any platform dependent modules are modified to support the new target platform; any required data files are moved to the new target system; and finally the complete system is built and tested on the target. Portable software makes this process as easy and reliable as possible.

Some of the important issues that can affect portability include the differences between the compilers being used on the different platforms, how to interface to required OS facilities, byte ordering, floating-point format, availability of standard APIs and facilities, and possibly the transfer of data between supported platforms.

This paper discusses these issues as they relate to source code and data portability. The paper is the result of experience gained during a project at Vista Control Systems, Inc. to re-engineer Vaccess. Vaccess is a real-time database with an API that transparently supports network access to remotely hosted databases.

Compilers

Compilers and languages in general are important issues with regard to developing portable systems. It is advantageous to choose a language that is available and mature on all the target platforms and in addition one that is covered by an adopted standard definition. There are a number of languages with adopted standards, such as FORTRAN and Pascal. However, languages such as FORTRAN are plagued by the availability of numerous extensions that make each implementation a non-portable super-set of the language. The FORTRAN-90 language holds the promise of being suitable as a portable language. However, FORTRAN-90 does not yet enjoy widespread support. Languages that do not provide a ‘complete’ set of facilities and runtime functions require extensions which are usually not consistent between vendors.

The popularity of C reflects it’s rich set of facilities, useful data types, operators, control structures, and runtime library functions. In addition to the characteristics of the language, C has another advantage over other languages, not specific to portability: There are a very large number of C programmers.

C is not free from many of the problems discussed above. Depending on the compilers involved, there can be a wide range in how consistent the different features of the language are implemented. Many older implementations include constructs and features not supported by, and incompatible with, the newer implementations. ISO compliant C compilers minimize the differences between implementations to a point where they are all but gone. This paper assumes the use of ISO C as the language used to develop portable systems.

What is not defined by ISO

Even if the compiler is ISO compliant, problems can still arise. The ISO standard leaves certain details up to the compiler implementers and these details can affect portability. For example, the ISO standard does not specify the default data type of a variable defined as *char*. Whether the variable is signed or unsigned is left up to the implementer.

In a similar way, some common C programming practices are specified as undefined in the ISO standard. For example, conversion between a function pointer type and a data pointer type is undefined, and the ISO standard does not guarantee that a function pointer can be safely converted to or from a type (*void* *). Code which includes such conversions most likely will not be portable.

Extensions provided by a compiler are also problem areas when it comes to portability. Be aware that some popular compilers may provide extensions to the ISO standard that may not be supported on all the target platforms. If a compiler supports a switch to disable ISO extensions or nonstandard constructs, the compiler can be used to help to find these problems early on in development, or when planning to port an existing system.

The size and range of data types is not completely specified by the ISO C standard. The standard simply states that an *int* can not be smaller than a *short* and that a *long* may not be smaller than an *int*. Similarly a *double* can not be smaller than a *float* and a *long double* may not be smaller than a *double*. The ISO standard does specify the minimum range for each type. For example, the legal values of an *int* fall in the range of -32,767 to 32,767. This requirement means that an ISO conforming compiler can not represent an *int* in only 8 bits. A compiler implementer is free to increase the range of a data type. If a data item must be of a known size, then by using a user-defined data type or new symbol that type can be supported on platforms that may have different sizes and ranges for the built-in C types.

User defined data types

Some platforms and compilers support data types not supported on other platforms and some are not fully specified in the ISO standard. Both of these problems can be handled with user defined data types, or simply new symbols defined for each type a system will need. Along with the user defined data types, it is important to keep in mind how these data types will be used and manipulated. If a data type may not be supported on a target, the chances are good that the normal C operators and runtime functions will not support the data type. In these cases a new function or macro should be defined and used to manipulate the data type. Configure the compiler to do as much argument checking as possible. This helps to reduce the possible incompatible use of user defined data types.

Platform independence

It is best when developing portable systems to isolate all the platform dependent code in separate modules. These modules can then be recoded on each target platform. This method is preferred over that using conditional compiles for a given section of code. Using separate compilation modules for isolating platform dependent code explicitly identifies future porting work. When maintaining the code, it is much less error prone to work on the platform dependent modules when a change is necessary, rather than working on a section of code in a platform independent module that could have side effects on other platforms the code runs on. Isolating platform dependent code also helps keeps the platform independent code much more portable when a new target is selected, since only the platform dependent modules need to be worked on. Platform dependent code includes any code that assumes the representation of data types, makes use of OS provided services, etc.

OS facilities

Any facility not part of the ISO standard for the C language must be isolated in platform dependent modules. Even though the newly adopted POSIX standard has tried to address some of the areas not included in the language specification, its support is not widespread enough to make it dependable. However, the use of POSIX can make the platform dependent modules much easier to port, so its use can still be beneficial. If more than one of the target platforms supports POSIX, porting the platform dependent modules could be trivial. They could be equivalent, thus reducing the work required to support a given platform.

Bi-endian support

When developing a portable system, it is impossible to ignore the issue of byte ordering. Intel and Digital both support little endian byte ordering while Motorola, SUN, and HP support big endian byte ordering. Some new processors can run in either or mixed modes. Any code that assumes the byte ordering of data is dependent on the platform and should be isolated in platform dependent modules.

Byte ordering is an issue when a multi-byte data item is not treated as a single, indivisible entity. For example, any code that accesses the individual bytes within a multi-byte integer, or manipulates the bits within an integer, is certainly byte order dependent. The use of C unions can also lead to byte order dependencies. Unions are used in a nonportable fashion any time a union component is referenced when the last assignment to the union was not through the same component.

Any use of C bit fields is likely nonportable. The ISO standard allows compiler implementers to impose constraints on the maximum size of a bit field and to specify certain addressing boundaries that bit fields cannot cross. Typically bit fields are used in platform dependent code to force a data structure to match a fixed hardware representation.

In general, any code that makes assumptions about the representation of the supported C data types is not platform independent and should be isolated.

File Specification Syntax

The syntax for specifying file names varies from platform to platform. All code that explicitly manipulates file specifications must be isolated in platform dependent modules.

Command Line Interface

Although the command line interface an application presents to the user can in most cases be made portable, the platform the application is running can be an issue. Users on a UNIX system expect a certain look and feel to the command lines presented to them, while OpenVMS users expect an entirely different presentation. Again, the code to interface with and parse the command line can be isolated in platform dependent modules. Another approach is to develop a library of routines for interfacing to the command line. This library could support all the different platform styles. This approach has the advantage that, for users who move between disparate looking platforms, the application can accept any style of command line. The user can enter command lines in a style that they are most comfortable with.

Standards

Standard facilities or interfaces can be used for minimizing the work required when porting the platform dependent modules of a system. However, since the support of these standards is not consistent or wide spread, they cannot be used in platform independent modules unless the availability of the standard on the target platforms is researched and validated. If a standard becomes mature enough and supported on all target platforms, it is very easy to convert a platform dependent module into a platform independent one.

Tools for improving portability

The portability of source code can be improved by using some common automated checking tools. Although tools cannot guarantee that source will be portable, they can help reduce the number of problems that require a programmer’s attention.

UNIX Lint

As described in the "man page" entry for lint, lint is a program checker that attempts to detect features of C programs that are likely to be bugs, non-portable, or wasteful. Lint can be very useful in finding problems related to the inconsistent use of data. It flags such potential problems as assigning a *long* value to a non-*long* value, comparisons with unsigned values, questionable use of pointers, and statements with an unknown order of evaluation. In general the type checking of lint is much stricter than compilers. Lint can also be used to find name clashes when external names are truncated to six characters and non-external names are truncated to eight characters. The ISO standard only guarantees external names are unique within the first six characters.

Multiple compiler use during development

Since portable source will almost always be compiled with different compilers, it is a big advantage to compile new source with as many different compilers as possible during development. Multiple compilers can quickly weed out the subtle differences between compiler implementations. Using two or more compilers during development produces source that is very likely to compile on additional platforms in the future.

Code generating tools

Another class of tool that can be applied to the production of portable systems are tools that generate code. If the code generated by a tool is verified to be portable, such a tool could be a great asset in producing a portable system. Such tools may include code generators for graphical interfaces, graphical tools for generating software controller algorithms, etc.

Portable data

For systems that make use of data files, another aspect of portability is the support for these data files. For simple files, like an X resource file, it is appropriate to use text files that are fairly portable.

Many systems read and write binary files that must be portable. For example, a utility on one platform may be used to generate a file that other related utilities read, and these other utilities may be running on various platforms. In this case, the utilities must be able to read the file independent of what platform produced it.

File headers

One method for making data files portable is by prefixing them with a standard file header which includes information indicating how the file was written. The header information should include enough information to discover the byte ordering of data in the file, the format of the floating point data, the character collating sequence, and possibly the format of any time data included in the file. Once this file header is developed, any utility can be developed to support reading and writing of portable binary. This method has the advantage that for files written and read on similar systems, no type conversion is required, resulting in better performance.

Using portable data types

Another approach to producing portable data files is to adopt a standard data format for all required data types. Since a single format is used for all data applications simply convert to that type for writing and convert from that type for reading. For systems that have native data types that match the standard format chosen, no conversion is every required. This prevents any performance gain when the platform where the data is written is similar to the platform where the data is read, if both have native types that are different from the chosen standard file format.

Data communication

As was the case for portable data files, many systems must pass data between different machines while they are running. This requires that all the machines must be able to read and write data that other machines write and read. Like the data file, to handle this problem for data communication a number of approaches are available.

RPC

A number of Remote Procedure Call packages are available that handle many of the problems associated with passing data between different machines. These packages handle the conversion of data between the host machine and the client machine.

Typically, a tool is provided to define all the arguments passed in the RPCs. A utility then reads the data definitions and produces code to handle the conversion of the data to and from the network representation.

Raw/Low level data

Using RPC may not always be appropriate, or available. When lower level data communication is required between machines, code must be written to support the conversion of data to and from different platform types.

One approach is to prefix the data with information similar to the binary file header so that the receiver of the data can convert the data to the native types if required. As with the file header approach, this technique has the advantage that for two similar platforms, no conversion is necessary.

Another approach is to convert the data to a standard type for transmission to the other machine. All parties involved must then convert to and from the standard type in order to communicate with the other machines. This is the approach normally taken by the RPC packages. In fact, many of the platforms that support RPC packages provide access to the routines used by the RPC for data conversion so that users can use the same routines for converting data to and from a standard network representation for their own use.

Portability of standard (popular) libraries

Many systems make use of libraries of routines that have become defacto standards or at least very popular. Even though these libraries are available on many platforms, there is no guarantee that the implementation on each platform is consistent. And in some cases the use of such a library provides so many advantages over writing equivalent routines that a system should make use of it even if it is not available on all the target platforms. In this case, isolating the use of the library in a platform dependent module allows the library to be used on platforms where it is available, and provides a good location in which to implement the equivalent functionality on platforms that do not.

Mixed language support

If a software system provides an API, support for mixed languages may be required even if the system is implemented in C. Normally, API support for mixed languages is provided by supplying different language bindings that support the standard argument passing methods for each language. A portable system may have to isolate the binding for a specific language in a platform dependent module since the inter-language calling conventions may not be uniform on different computing platforms.

One Experience

Vista has experienced great success in producing portable systems. Many of the techniques discussed in this paper have been applied in the re-engineering of our core product, Vaccess. With the initial release of the re-engineered version of Vaccess expected early next year a number of successful ports have already been completed. The initial development of the new version was done on Digital UNIX and VxWorks. The first port of the product was to OpenVMS and was accomplished in less than eight man hours. Subsequent ports to UNIXware, Solaris x86, and Solaris SPARC have been completed in four man hours each.

As a comparison, early in 1990 the then current version of Vaccess was ported from OpenVMS to VAXELN. This port benefited from many advantages in that the two platforms supported many similar OS facilities, and they both ran on the VAX processor. There was no need for supporting different data types, byte ordering, etc. With all of these advantages this port took more than four man months of effort to complete initially. This is about 100 times the effort required to port the new product to platforms with much different characteristics. In addition, maintenance of Vaccess became more and more difficult because of the required support for both OpenVMS and VAXELN.

CONCLUSIONS

The development of portable systems can be made straightforward by taking the time to become aware of the issues involved in portability. Although implementing a portable system may incur an additional overhead initially, the benefits justify this expense.

Experiences at Vista have demonstrated the direct benefit of producing portable systems with the completion of numerous ports with minimal effort. Additional benefits will be realized with the reduction in maintenance effort for the system during its lifetime.