Introduction

The typical definition of portability goes something like this:

Portability is a measure of how easily software can be made to execute successfully on more than one operating system or platform.

This definition, however, is only a starting point. To truly define the term *portability,* one must consider more than the mere fact that the software can be made to execute on another operating system. Consider the following C++ code:

```
#include <iostream>

using namespace std;

int
main( int argc, char *argv[])
{
     cout << "Hello\nWorld";
}
```

Most readers will agree that the preceding code will compile and execute on just about any platform that supports a decent C++ compiler.

However, potential problems lurk, even in code as simple as this. Let's run this program from a shell prompt on a Windows machine (assume the Windows executable that results from building the preceding source has the name foo.exe), and redirect the output to a text file:

```
c:\ > foo > foo.txt
```

We can use `wc(1)` to see how many characters are in the resulting file:

```
c:\ > wc -c foo.txt
12 foo.txt
```

Now let's do the same thing on Mac OS X:

```
% foo > foo.txt
% wc -c foo.txt
11 foo.txt
```

Notice on Windows that `wc` reports that the file contains 12 characters, whereas on Mac OS X, it reports that it contains 11 characters. Surprised? Let's use `od(1)` to figure out what happened. First, let's run `od(1)` on Windows:

```
C:\> od -c foo.txt
0000   H   e   l   l   o  \r  \n   W   o   r   l   d
0014
```

And, then do the same on Mac OS X:

```
% od -c foo.txt
0000   H   e   l   l   o   \n   W   o   r   l   d
0013
```

What has happened here is that on Windows, the `\n` in our string was converted to DOS line endings, which are always `\r\n`. On Mac OS X, the `\n` was left alone.

In most cases, this difference is of no consequence. After all, running the application on either platform gives the same basic output when viewed in a console. If console output were the only criteria, you would be perfectly correct to consider the code as being portable, and leave it at that. But what if the size of the resulting file did matter, somehow? For example, what if the code snippet is part of a system that uses the content of the file to determine a size, which is then used by some process on another system to allocate a buffer? In one case (Windows computes the size, and Mac OS performs the allocation), we'd end up wasting a byte of data, because we would be writing an 11-byte value into a 12-byte buffer. Computing the size on Mac OS X and writing the result on Windows might, however, result in overflow of the buffer by a byte, which could lead to undefined behavior.

So, even the venerable "Hello World" can be the source of portability issues depending on how that code is actually used. Let's look at some of the major factors that can impact your ability to write portable software.

Areas That Can Affect Software Portability

Language

As the title implies, this book covers C++, but because of their close relationship, the C language is also covered, if only indirectly. The C language has been considered a portable one ever since the late 1970s when it was first described in K&R, and it's well known that one of the major reasons the UNIX operating system has found itself on so many different hardware platforms is because a majority of the operating systems have been written in the C language. Standardization efforts (notably ANSI and the more recent C99) have led C to become an even more portable programming language. Programming against the ANSI standard, and avoiding language extensions introduced by compiler vendors, is an important step toward eliminating portability issues in C. You can increase your odds of writing portable C by instructing your compiler (via its command-line flags or settings) to only accept standards-based language constructs, and to reject any and all language extensions provided by the compiler manufacturer. This advice also holds true for C++.

Compilers

Closely tied to the portability of both the C and C++ languages are, of course, the compilers that are used to turn source code into an executable form. I mentioned previously that the compiler can be used to control standards adherence, but there is more to say about the contribution a compiler makes to overall portability. Several compilers are available for the platforms that we care about in this book, the most popular being, by far, Microsoft Visual C++ 6.0 and Visual C++ 7.0 .NET, which are available for Microsoft Windows; and GNU's GCC, an open source compiler that is available on numerous platforms, including Mac OS X, Linux, and via the Cygwin project, Windows, too.

The C and C++ languages, as they are defined, leave the details regarding the implementation of several language features in the hands of compiler vendors. Subsequently, the use of these features can be inherently nonportable. Things to avoid, or be aware of, include the following:

- **Sizes of the short, int, and long built-in types**
  The size of these types is, by definition, compiler dependent. The C standard says that shorts must be at least 16 bits in size. It also says that ints have to be at least as large as shorts. Finally, it says the longs have to be at least as big as ints. However, this means a 32-bit machine can either implement shorts as 16-bit, ints as 32-bit, and longs as 32-bit, or it can implement shorts and ints as 32-bit, and longs as 64-bit. There are obviously other ways to adhere to the standard. Typically, 32-bit machines will support a 32-bit int, and a 64-bit int would be expected on 64-bit systems, because int is typically defined in terms of the native word size. But, even that is not guaranteed. In this book, I introduce the Netscape Portable Runtime Library (NSPR), which provides a solution to this particular problem.
- **Bitwise operators**
  Errors can be introduced by assuming the sizes of the short, int, and long types being manipulated. These sizes are, once again, determined by the compiler. Right-shifting a value can result in either propagation of the sign bit, or zero filling the leftmost bits, and this is also compiler dependent.
- **Signed versus unsigned char types**
  C and C++ do not specify whether the char data type is signed or unsigned; this is left to the discretion of the compiler writer. This can be a problem when mixing char and int types in code—the classic example is reading characters from stdin in C into a unsigned char using the `getchar()` function, which returns an int and typically uses –1 to indicate end of file. If you are in a loop comparing –1 to an unsigned char, your loop will never end, regardless of whether `getchar()` has encountered EOF or not. You can avoid the problem by explicitly declaring the signed attribute of a character variable as follows:

  ```
  signed char foo; // signed, range -128 to 127;
  unsigned char fee; // unsigned, range 0 to 255;
  ```

  Assigning and comparing only like types (assigning only chars to chars, for example), using C++-style casts whenever conversions are necessary, adhering to function prototypes (for instance, don't assign the return value of `getchar()` to anything other than an int), and fixing each and every warning that your compiler generates—all are ways to overcome portability issues such as this.

Binary Data

In addition to endian issues (the order in which bits and bytes are arranged in memory), binary data can also suffer from how the compiler chooses to lay out structs in memory, which is entirely compiler dependent, and impacted by the architecture being targeted. Structs written as binary (to memory or disk) by code generated by one compiler may be a completely different size, or be organized in memory in a completely different way, by another compiler, even on the same platform. The best way to avoid this problem is to avoid writing or interchanging binary data, and use text instead. This might not always be practical, however, and so I discuss strategies for dealing with binary data in Item 20.

Standard Libraries

The standard libraries (and headers) extend the capabilities provided by the core C and C++ languages. Portability is a central motivation for having a standard library. C standard library headers include the familiar `<stdio.h>`, `<ctype.h>`, and `<string.h>`, among several others. Functions and macros declared in these headers, including `strncmp()`, `getchar()`, `printf()`, `malloc()`, `fopen()`, and countless others, have been used by nearly every C program that has ever been written. Because the standard library provides so much value to the average programmer, and because it is supported by every C implementation, using the functions, constants, and macros provided by these libraries and headers increases the chances that C code that you write will port successfully from one compiler (or platform) to another.

In the C++ world, there is the STL, the Standard Template Library, which is formally defined as a part of the standard C++ language. The STL preserves what the C standard library provides (but renames the headers by adding the character `c` as a prefix, and dropping the `.h` suffix (for example, `<cstdio>`, `<cctype>`, and `<cstring>`), and extends it with additional functionality that plays rather nicely with the C++ language in several critical ways. The STL provides not only I/O support, but also a well-designed set of container classes that the standard makes promises about in terms of performance. In general, I recommend that you always use the STL in favor of using similar functionality that might be provided elsewhere (including custom code that you have written). It makes no sense to beat your head against the wall trying to come up with an efficient linked list implementation, when someone before you has gone to the effort to supply you with an optimal implementation in the STL. Using the STL is one more way to increase the chances that your code will port. The same holds true for the rest of the standard library; learn it, and use it whenever possible.

Related to the STL is the open source Boost project (www.boost.org), an effort to fill gaps in what the STL provides, with the premise that some of the work that results will eventually find its way into the STL.

Operating System Interfaces

Operating system interfaces (also referred to as system calls) add to what the core language and standard library provide, enabling applications to perform system-dependent tasks. Many of the functions that one finds in the standard library are implemented in terms of the functions provided in this category. Functionality includes such things as process creation, interprocess communication (IPC), low-level input and output, interfacing to device drivers, and network I/O, to name a few. As you might guess, much of the functionality provided in this category is highly system dependent.

A look at the process creation functions on UNIX and Win32 provides an illustration of just how differently system calls can be implemented. To create a process in Win32, you use the `CreateProcess()` function:

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

For example, to launch the Notepad.exe application, you might write and execute the following code under Windows:

```
#include <windows.h>

STARTUPINFO si;
PROCESS_INFORMATION pi;

ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
```

```
ZeroMemory( &pi, sizeof(pi) );

// Start the child process.
if(!CreateProcess(NULL, "notepad", NULL, NULL,
    FALSE, 0, NULL, NULL, &si,  &pi))
            return E_FAIL;
else
     return S_OK;
```

On UNIX, we would use `fork(2)` and the `exec(3)` family of functions to create and execute a process. The prototype for `fork(2)`, which creates a process, is this:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

The prototype for `execl(3)`, which instructs a process to load and execute a specified program, is this:

```
#include <unistd.h>

extern char **environ;
int execl(const char *path, const char *arg, ...);
```

The following program illustrates how you would use `fork` and `exec` to launch the UNIX `date(1)` program. In the following, the return value from `fork(2)` determines which process was created (the child) and which process called `fork(2)` to begin with (the parent). Here, the child process will execute the `date` program by calling `execl(3)`, while the parent process simply exits:

```
#include <sys/types.h>
#include <unistd.h>

int
main(int argc, char *argv[])
{
    pid_t pid;

    pid = fork();
    if (pid == 0) {
        /* child process */

        execl("/bin/date", "date", NULL);
    }
    return(0);
}
```

It should be obvious from this source code that process creation is anything but portable. However, all is not lost. In Items 16 and 17, I discuss ways in which nonportable native system functionality can be made portable by using standard interfaces and portability libraries such as NSPR. The standard interfaces covered in these items also includes POSIX, the IEEE 1003 standard interface for operating systems, the System V Interface Description (SVID), and finally XPG, The X/Open Portability Guide, which is required of all operating systems that call themselves "UNIX." The GCC compiler family supports all three of these standards on the platforms that this book addresses. If you decide to go with Cygwin and GCC on Microsoft Windows, most of the battle is won. If not, and you choose to use Microsoft or some other vendor to provide your compiler and supporting libraries, other strategies will need to be considered to obtain pure source code compatibility. NSPR (described in Item 17) and Boost can help here. Building your own abstraction layer above the corresponding Win32 interfaces is another way to deal with this problem, and I discuss how abstraction layers can be developed in Item 22.

User Interface

The user interface is perhaps the least portable aspect of modern desktop platforms (at the application level), and we will spend a great deal of time in this book discussing how to overcome this limitation to writing highly portable software.

Each platform provides its own user interface toolkit to support native graphical user interface (GUI) development. On the Windows platform, one finds Win32, Microsoft Foundation Classes (MFC), and the evolving .NET application programming interfaces (APIs) that will eventually replace both. On Mac OS X, one finds the Objective-C based Cocoa framework, and the Carbon API, which was designed to be similar to the legacy Mac OS Toolbox, and to ease the portability of legacy programs from Classic Mac OS 7/8/9 to Mac OS X. And on Linux, you have a wide variety of choices, ranging from Gtk+ (GNOME), Qt (KDE), and in some applications, Xt/Motif (CDE), all of which are based on the X Window System.

None of these toolkits is source code compatible. Nor do any of these toolkits have the same look and feel. Two of the toolkits that were mentioned previously, Qt and Gtk+, are available for Microsoft Windows, but they are not commonly used there, mainly because of the overwhelming dominance that Microsoft-supplied toolkits enjoy on that Windows platform.

A lot can be said for programming to the native GUI toolkits and their APIs. Your application will integrate nicely with the environment when native APIs are used. The users of your application will likely find your user interfaces easy to learn and use, and the ability of your application to interact seamlessly with other applications on the desktop can arguably be expected to be better when you program against a natively supplied toolkit. If portability is not an issue, then using native APIs is usually the way to go. But, because you are reading this book, portability is in all likelihood *the* issue, so we need to find ways to maximize code portability while minimizing the negatives that might be associated with not using native toolkits. We explore ways to solve this problem in Chapters 7, 8, and 9 of this book.

Build System

A build system can vary from a simple script that executes the compiler and linker command line, to a full-blown system based on automake or Imake. The key reason for using a standard and shared build system is so that developers can easily move between machines while in development. For example, if you are a Win32 developer, you can easily move over to OS X and kick off a build with your changes, because the tools you are interfacing with are largely the same. In this book, I describe Imake, focusing on how it supports code portability. Automake and autoconf, popular in the open source community, supply another strategy for dealing with cross-platform Makefile generation, and are well documented elsewhere. Integrated development environments (IDEs) such as those provide with Microsoft Visual Studio .NET, or Apple's Interface Builder and Project Builder, inhibit the development of portable build systems, not to mention portable code. Item 8 discusses the place of IDEs in cross-platform development.

Configuration Management

Imagine you are the only developer on a project, consisting of no more than a couple of dozen source files, all located in a directory somewhere on your hard drive. You edit the files, compile them, and test. Occasionally, you take the source directory, compress it using zip or gzip, and then you save it somewhere in case a catastrophe occurs (such as accidentally deleting the source, or a hardware failure). For the sake of this discussion, let's assume these backups are made weekly.

Now, let's suppose you make a first release of your software, get feedback from others, and spend a few weeks working on bugs and adding several requested features. And then you release a new version to testers, and get the unwelcome news that a feature that was once working is now broken. It is obvious that the feature once worked, and that something you changed since your first release and now is what led to the bug. How do you figure out what you changed in source code that led to the problem? One strategy is to build from your weekly backups, and isolate the bug to the work done on a particular week. Comparing the changes in that build's source code with the source code corresponding to the release that failed will identify what changes were made, and from there you can start making educated guesses as to what went wrong.

However, there is a better solution, one that solves even more problems than the one just outlined. The solution is configuration management. Even if you are the only developer on a project, a configuration management system is highly beneficial; but on projects with multiple developers, configuration management becomes critical, because the problems become magnified, and new problems are introduced, when more than one developer is contributing to a single codebase. Perhaps the most notable of these is how to effectively merge changes made by multiple developers into a single codebase without introducing errors in the process.

There are numerous configuration solutions out there, many which are platform specific, but the best by far are open source, including the Concurrent Version System (CVS) and Subversion. Besides being open source, both are available in command-line and GUI form on almost every platform worth considering. CVS is the standard source control system for the open source community, and it is extremely popular in industry, too. In Item 13, I introduce CVS, giving you all you need to know to use it effectively. A related tool, `patch(1)`, which increases the usefulness of a configuration system, especially when multiple developers are involved in a project, is covered in Item 14.

The Role of Abstraction

Abstraction is a theme central to several of the tips and techniques presented in this book. The importance of abstraction to engineering, if not to the world that we live in, cannot be understated. To *abstract* something means to deal with it *without concern to its concrete details*. Without abstraction, something that was once simple risks becoming overly complicated, its true essence lost in the minutiae. Take, for example, the following PowerPC binary code:

```
0011540 0000 0000 7c3a 0b78 3821 fffc 5421 0034
0011560 3800 0000 9001 0000 9421 ffc0 807a 0000
0011600 389a 0004 3b63 0001 577b 103a 7ca4 da14
```

Do you have any idea what sort of program this code (snippet) corresponds to? (If you do, perhaps you need to get out more often!) Although it is hard to imagine, at one time people actually wrote code at this primitive level, typing in numbers that corresponded directly to machine-level instructions. It's amazing that programming in this way ever resulted in getting computers to do anything useful. (Of course, back in the days of machine language programming, not much was asked of computers, compared to what is asked of them today).

Even today, with all the hardware advances that have occurred, programmers would be rare, and the programs they would write would be fairly lacking in functionality, if they were forced still to program at such a low level of abstraction. Such programming would be error prone; it would be impossible for anyone trying to maintain such code to glean any meaning from a bunch of binary values, such as those illustrated here.

The difficulties that are inherent in working with programs at the binary level were overcome by introducing an abstraction: assembly language.

The following code, written in PowerPC assembler, represents an abstraction over the binary code that was presented previously:

```
_F:
    mflr r0
    stmw r29,-12(r1)
    stw r0,8(r1)
    stwu r1,-80(r1)
    mr r30,r1
    stw r3,104(r30)
    lwz r0,104(r30)
    cmpwi cr7,r0,0
    bne cr7,L8
    li r0,0
    stw r0,56(r30)
    b L10
L8:
    lwz r0,104(r30)
    cmpwi cr7,r0,1
    bne cr7,L11
    li r0,1
    stw r0,56(r30)
    b L10
L11:
    lwz r2,104(r30)
    addi r0,r2,-1
    mr r3,r0
    bl _F
    mr r29,r3
    lwz r2,104(r30)
    addi r0,r2,-2
    mr r3,r0
    bl _F
    mr r0,r3
    add r29,r29,r0
    stw r29,56(r30)
L10:
    lwz r0,56(r30)
    mr r3,r0
    lwz r1,0(r1)
    lwz r0,8(r1)
    mtlr r0
    lmw r29,-12(r1)
    blr
```

However, this level of abstraction is far from ideal. Spend a few minutes looking at the code, and see whether you can determine which well-known mathematical sequence the preceding code is generating. Not so easy, is it? I wrote the code (actually, the GNU C compiler generated the assembly code for me from C code I supplied), and even I have a hard time knowing what this assembly language actually represents, or mapping it back to the C source from which it was generated.

The truth is that although assembly language is a big improvement over machine code, it is not abstract enough to make much of a difference. To most of us, it might as well be in machine code; programming applications at this level is inappropriate, at best. Yet, when introduced, assembly language did lead to the creation of more meaningful applications, and it did make debugging and maintaining code easier to deal with. But, it was only an incremental improvement.

As you no doubt know, assembly language was not the end of the story. Let's take a look at a further abstraction of the above, this one provided courtesy of the C programming language:

```
int F(const int n)
{
    if (!n || n == 1)
        return n;
    return (F(n - 1) + F(n - 2));
}
```

The C programming language provides our first hope of understandable, maintainable code. We can see that the function `F()` is a recursive function that generates an integer result. The result is a Fibonacci number, which is based on the recurrence relation shown in Figure Intro-1.

$$f(x) := \begin{cases} 0 & \text{if } x = 0; \\ 1 & \text{if } x = 1; \\ f(x-1) + f(x-2) & \text{if } x > 1. \end{cases}$$

Figure Intro-1 Fibonacci sequence

Of course, in the end, the code written in C ends up being generated by the compiler to a form less abstract than C itself, and then ultimately, the object code is a series of 1s and 0s. But we don't need to concern ourselves with this level of detail. Programming languages like C make large-scale software development possible, because the abstractions that they provide make the act of writing, debugging, and maintaining programs much easier than it would be done in assembly language.

We can go even further in our quest for abstraction. We can place the preceding code in a library, place the function prototype in a header file, and allow programmers to call `F()` from their applications, without needing to know a thing about how it is implemented. Related techniques might involve using Component Object Model (COM) or Common Object Request Broker Architecture (CORBA), where interactive data language (IDL) is used to define the interface of functions abstractly; but, effectively, the level of abstraction achieved is the same. Once hidden, we can vary the implementation of a function without affecting the code that calls it. (Depending on who is using the function, we can vary its interface, too; but ideally, the interface is a contract between the producer of `F()` and the consumer that does not vary.) We can take advantage of optimizations present on one platform that might not be available on others (for example, hardware support for computing Fibonacci sequences). We can make `F()` available in whatever way a particular platform requires, and no one need be the wiser to what is actually going on.

Let's return for a moment to the C programming language. Although C wasn't the first programming language to provide a suitable level of abstraction, it was perhaps the first language to demonstrate the impact that abstraction could have on code portability on a large scale. Early versions of the UNIX operating system were written completely in assembly language, making it hard (if not impossible) to port from one architecture to another. When faced with the daunting task of porting UNIX to other architectures, the designers of UNIX faced a choice: reimplement the miles of PDP-7 assembly language code, or come up with an abstraction. One choice would have been to come up with an abstract machine and correspondingly abstract assembly language, but by then, procedural languages were coming into existence, and the designers of UNIX made a better choice. They invented the C language explicitly to ease the porting of UNIX to other architectures, and the decision to implement the majority of the UNIX operating system in C, a novel idea back in the 1970s, was essential to the eventual widespread adoption of both C and the UNIX operating system.

Writing code in the C programming language often doesn't make it portable enough, however. We must use, or invent, additional layers of abstraction to make the software we write truly portable. This is especially true for userland application software that is intended to run on Linux, Windows, and Mac OS X. Just like assembly language varies among processor architectures, APIs vary from operating system to operating system, and this is especially true when it comes to the APIs associated with GUIs. To gain portability in our code, we must abstract these APIs. For example, where we might call `open()` on Linux or Mac OS X to open a file, we are forced to call `_open()` under Win32, and so we might abstract this by creating a wrapper function, like this:

```
int PortableOpen(const char *path, int flags, mode_t mode)
```

The function `PortableOpen()` is designed to hide the platform-specific details of opening a file. Programmers use this function, rather than the platform-specific function, allowing them to ignore the details of the platform implementation. In this book, you will see techniques for implementing such abstractions, both in C and C++. Some of these abstractions are simple, like those shown here. Item 17, in particular, introduces NSPR, a C-based API that is an abstraction layer above commonly used native operating system APIs such as `open()`.

The book deals with GUI abstractions in several items. In Chapter 9, I describe XPToolkit and XUL, Netscape's and Mozilla's approach to solving the cross-platform GUI problem, and the design of Trixul, a related GUI toolkit of my own design. Both of these abstract the native GUI APIs found on Mac OS X, Linux, and Windows. They make use of abstractions that are more complex, and considerably more powerful, than the ones used by NSPR, and are based on design patterns such as factory, observer, and model/view. Combined, these abstractions are critical to designing portable software in the fashion that was employed by Netscape, and in the fashion used by Mozilla and Firefox to this day.

The concept of abstraction is mentioned explicitly, or otherwise implied, in numerous items presented in this book. Chapter 8, which covers wxWidgets, and Chapter 9, which covers the design of cross-platform GUI toolkits, both discuss technologies that are based on architectures involving high levels of abstraction.