CHAPTER XV

# Portability

Portability doesn't mean writing programs that can run, unchanged, on every computer ever invented. It just means writing programs so that when things change, the programs don't have to change, much.

Don't be too quick to say, "It won't happen to me." Many MS-DOS programmers didn't worry about portability until MS-Windows came around. Then all of a sudden, their programs had to run on what looked like a different operating system. Mac programmers got to deal with a new processor when the Power PC caught on. Anyone who has maintained a program on various flavors of UNIX probably has more knowledge about portability than would fit into this whole book, let alone one chapter.

Say you're writing anti-lock braking software using Tucker C for the Anti-Lock Braking and Tire Rotation operating system (ALBATR-OS). That might sound like the ultimate in nonportable software. Even so, portability can be important. You might have to port your software from version 7.55c of Tucker C to version 8.0, or from version 3.0 to 3.2a of ALBATR-OS, to pick up some bug fixes. You might have to port it (some of it) to MS-Windows or a UNIX workstation, to be used in a simulation for testing or advertising purposes. And more likely than not, sometime between when the first line of code is written and when the last line is finally debugged, you might have to port it from one programmer to another.

Portability, in its best sense, means doing things in an unsurprising way. The goal isn't to make the job easier for the compiler. The goal is to make the job easier for the poor slobs who have to write (and rewrite!) the code. If you're the "poor slob" who gets someone else's

code, every surprise in the original code will cost you time, leave the potential for making subtle bugs, or both. If you're the original coder, you still want to make the code unsurprising for the next poor slob. You'll want the code to be easy enough to understand that no one will come complaining to you that they don't understand it. Besides, more than likely, the next "poor slob" will be *you*, several months after you've forgotten why you wrote that `for` loop in such a tricky way.

The essence of making your code portable is simple: *If there's a simple, standard way of doing something, do it that way!*

The first step in making your code portable is to use the standard library functions, and to use them with the header files defined in the ANSI/ISO C standard. See Chapter XII, "Standard Library Functions," for details.

The second step is to, whenever possible, write code that you expect will work with all compilers, rather than code that appears to work with your current compiler. If your manual warns about a feature or a function being specific to your compiler, or certain compilers, be wary of using it. Many good books on C programming have advice on what you can depend on working portably. In particular, if you don't know whether something will work, don't immediately write a test program and see whether your compiler accepts it. Just because your compiler does, in this version, doesn't mean that code is very portable. (This is more of a problem for C++ programmers than for C programmers.) Besides, small test programs have a tendency to miss some aspects of the feature or problem they were intended to test.

The third step is to isolate any nonportable code. If you're not sure whether part of your program is portable, add a comment to that effect as soon as you can! If large parts of your program (whole functions or more) depend on where they run or how they're compiled, put the different nonportable implementations in separate .c files. If small parts of your program have portability issues, use `#ifdef`s. For example, filenames in MS-DOS look like `\tools\readme`, but under UNIX, they look like `/tools/readme`. If your code needs to break down such filenames into their separate parts, you need to look for the right separator. With code like

```
#ifdef unix
#define FILE_SEP_CHAR '/'
#endif
#ifdef __MSDOS__
#define FILE_SEP_CHAR '\\'
#endif
```

you can use `FILE_SEP_CHAR` as an argument to `strchr` or `strtok` to find the "path" of directories that lead to the file. That step won't find the drive name of an MS-DOS file, but it's a start.

Finally, one of the best ways to find potential portability problems (and ways to fix them) is to have someone else find them! Seriously, if you can, have someone else look over your code. He or she might know something you don't, or might see something you never thought of. (Some tools, often with the word "lint" in their names, and some compiler options can help find some problems. Don't expect them to find big ones.)

# XV.1: Should C++ additions to a compiler be used in a C program?

## *Answer:*

Not unless your "C program" is really a C++ program.

Some features of C++ were so nifty that they were accepted by the ANSI/ISO C standards committees. They're no longer "C++ additions"; they're part of C. Function prototypes and the `const` keyword were added to C because they were really good ideas.

A few features of C++, such as inline functions and ways of using `const` to replace `#define`, are sometimes called "better C" features. There have been a few partly C++ compilers with a few of these features. Should you use them?

Here's one programmer's opinion: If you want to write C code, write C code. Write code that all C compilers will accept. If you want to take advantage of C++ features, move to C++. You can take baby steps, a few new tricks at a time, or you can go all out and create templated pure abstract base classes with lots of inlines and exceptions and conversion operators. After you've crossed the line, though, you've crossed it; your program is now a C++ program, and you shouldn't expect a C compiler to accept it.

Now let me say this. Work has started on a new C standard, one that will include some C++ features and some brand new features. Over the next few years, some of those new features will be implemented by some compiler vendors. That doesn't guarantee they will be implemented by all compilers, or make it into the next C standard. Keep your ears open. When it sounds as if a new feature has really caught on, not just in the compiler you use but in all the ones you might use, then think about using it yourself. It didn't make sense to wait until 1989 to start using function prototypes. On the other hand, it turns out there was no good time to start using the `noalias` keyword if you wanted your code to be portable.

## Cross Reference:

XV.2. What is the difference between C++ and C?

# XV.2: What is the difference between C++ and C?

## *Answer:*

There are two perspectives to consider: the C programmer's, and the C++ programmer's.

To a C programmer, C++ is a quirky language that's hard to deal with. Most C++ libraries can't be linked into a C program by a C compiler. (There's no support of templates or "virtual tables," which the compiler has to create at link time.) Even if you link your program with a C++ compiler, a lot of C++ functions can't be called at all from C code. C++ programs, unless they're very carefully designed, can be somewhat slower and a lot bigger than similar C programs. C++ compilers have more bugs than C compilers. C++ programs are much harder to port from one compiler to another. Finally, C++ is a big language, hard to learn. The definitive (for 1990) book was more than 400 pages long, and more has been added every year since then.

C, on the other hand, is a nice, simple language. No changes have been made to the language in years. (That won't last forever; see FAQ XV.1.) The compilers are good and getting better. Good C code is trivial to port between good C compilers. Object-oriented design isn't easy to do in C, but it's not that hard. You can (almost) always build your C code with C++ compilers if you want.

To a C++ programmer, C is a good beginning. There are many mistakes you can make in C that you'll never make in C++; the compiler won't let you. Some of the tricks of the C trade can be very dangerous if just slightly misused.

C++, on the other hand, is a great language. With a little discipline and up-front design work, C++ programs can be safe, efficient, and very easy to understand and maintain. There are ways of writing C++ programs so that they will be faster and smaller than the equivalent C programs. Object-oriented design is very easy in C++, but you're not forced to work that way. The compilers are getting better every day, and the standards are firming up. You can (almost) always drop down to the C level if you want to.

What, specifically, is different between C and C++? There are a few C constructs that C++ doesn't allow, such as old-style function definitions. Mostly, C++ is C with new features:

- A new comment convention (see FAQ XV.3).
- A new "Boolean" type with real `true` and `false` values, compatible with existing C and C++ code. (You can throw away that piece of paper taped to your monitor, the one that says, "0 = false, 1 = true." It's still valid, but it's just not as necessary.)
- Inline functions, safer than `#define` macros and more powerful, but just as fast.
- Guaranteed initialization of variables, if you want it. Automatic cleanup of variables when they go away.
- Better, safer, stronger type checking and memory management.
- Encapsulation, so new types can be defined with all their operations. C++ has a `complex` type, with the same operations and syntax as `float` or `double`. It's not built into the compiler; it's implemented in C++, using features every C++ programmer can use.
- Access control, so the only way to use a new type is through the operations it allows.
- Inheritance and templates, two complementary ways of writing code that can be reused more ways than just calling functions.
- Exceptions, a way for a function to report a problem further than just the function that called it.
- A new approach to I/O, safer and more powerful than `printf`, that separates formatting from the kind of file being written to.
- A rich library of data types. You'll never have to write a linked list or a binary tree again. (This time for sure, honest. Really!)

Which is better, C or C++? That depends on who you are, who you're working with, how much time you have to learn, and what tools you need and want and can use. It depends. There are C++ programmers who will never go back to C, and C programmers who have gone back from C++ and love it. There are programmers who are using some C++ features and a C++ compiler, but who have never really understood C++, who are "writing C programs in C++." Hey, there are people writing FORTRAN programs in C (and C++); they never caught on either.

Great programming languages don't make great programs. Great programmers understand the language they're programming in, whatever language it is, and use it to make great programs.

### Cross Reference:

XV.1: Should C++ additions to a compiler be used in a C program?

## XV.3: Is it valid to use // for comments in a C program?

### *Answer:*

No. Some C compilers might be able to support the use of `//`, but that doesn't make it C.

In C, a comment starts with `/*` and ends with `*/`. C-style comments are still valid in C++, but there's another convention as well. Everything after (and including) `//`, up to the end of a line, is considered a comment. For example, in C you could write this:

```
i += 1; /* add one to i */
```

That's valid C++, but so is the following line:

```
i += 1; // add one to i
```

The advantage of the new C++ comments is that you can't forget to close it, as you can with a C-style comment:

```
i += 1; /* add one to i
printf("Don't worry, nothing will be "); /* oops */
printf("lost\n);
```

In this example, there's only one comment. It starts on the first line and ends at the end of the second line. The "don't worry" `printf` is commented out.

Why is this C++ feature more likely than any other to creep into C compilers? Some compilers use a separate program for a preprocessor. If the same preprocessor is used for C and C++ compilers, there might be a way for the C compiler to get the preprocessor to handle the new C++ comments.

C++-style comments are very likely to be adopted into C, eventually. If, one day, you notice that all the C compilers you might use support `//` comments, feel free to use them in your programs. Until then, use C comments for C code.

### Cross Reference:

Introduction to Chapter V: Working with the Preprocessor

V.2: What will the preprocessor do for a program?

XV.1: Should C++ additions to a compiler be used in a C program?

## XV.4: How big is a *char*? A *short*? An *int*? A *long*?

### *Answer:*

One byte, at least two bytes, at least two bytes, and at least four bytes. Other than that, don't count on anything.

A `char` is defined as being one eight-bit byte long. That's easy.

A `short` is at least two bytes long. It might be four bytes, on some machines, with some compilers. It could be even longer.

An `int` is the "natural" size of an integer, as long as that's at least two bytes long and at least as big as a `short`. On a 16-bit machine, an `int` is probably two bytes long. On a 32-bit machine, an `int` is probably four bytes long. When 64-bit machines become common, their `int`s will probably be eight bytes long. The operative word is "probably." For example, the original Motorola 68000 was a hybrid 16/32-bit machine. One 68000 compiler generated either two-byte `int`s or four-byte `int`s, depending on a command-line option.

A `long` is at least as big as an `int` (and thus, at least as big as a `short`). A `long` must be at least four bytes long. Compilers for 32-bit machines might make `short`s, `int`s, and `long`s all be four bytes long—or they might not.

If you need some integral variable to be four bytes long, don't assume that an `int` or a `long` will do. Instead, have a `typedef` to some built-in type (one probably exists), and surround it with `#ifdef`s:

```
#ifdef FOUR_BYTE_LONG
typedef long int4;
#endif
```

You might use such a type if you need to write an integer variable as a stream of bytes, to a file or to a network, to be read by a different machine. (If you do, you should see the next FAQ as well.)

If you need some integral variable to be two bytes long, you might be in trouble! There's no guarantee such a beast exists. You can always squeeze a small value into a two-`char` array; see the next FAQ for details.

## Cross Reference:

X.6: How are 16- and 32-bit numbers stored?

XV.5. What's the difference between big-endian and little-endian machines?

# XV.5: What's the difference between big-endian and little-endian machines?

## *Answer:*

The difference between big-endian and little-endian is in which end of a word has the most significant byte. Looked at another way, it's a difference of whether you like to count from left to right, or right to left. Neither method is better than the other. A portable C program needs to be able to handle both kinds of machines.

Say that your program is running on a machine on which a `short` is two bytes long, and you're storing the value 258 (decimal) in a `short` value at address 0x3000. Because the value is two bytes long, one byte will be stored at 0x3000, and one will be stored at 0x3001. The value 258 (decimal) is 0x0102, so one byte will be 1, and one will be 2. Which byte is which?

That answer varies from machine to machine. On a big-endian machine, the most significant byte is the one with the lower address. (The "most significant byte" or "high-order byte" is the one that will make the biggest change if you add something to it. For example, in the value 0x0102, 0x01 is the most significant byte, and 0x02 is the least significant byte.) On a big-endian machine, the bytes are stored as shown here:

| | | | | | | |
|---|---|---|---|---|---|---|
| *address* | 0x2FFE | 0x2FFF | 0x3000 | 0x3001 | 0x3002 | 0x3003 |
| *value* | 0x01 | 0x02 | | | | |

That makes sense; addresses are like numbers on a ruler, with the smaller addresses on the left and the larger addresses on the right.

On a little-endian machine, however, the bytes are stored as shown here:

| | | | | | | |
|---|---|---|---|---|---|---|
| *address* | 0x3003 | 0x3002 | 0x3001 | 0x3000 | 0x2FFF | 0x2FFE |
| *value* | 0x01 | 0x02 | | | | |

That makes sense, too. The smaller (in the sense of less significant) part is at the lower address.

Bad news: some machines store the bytes one way; some, the other. For example, an IBM compatible handles the bytes differently than a Macintosh.

Why does that difference matter? What happens if you use `fwrite` to store a `short` directly, as two bytes, into a file or over a network, not formatted and readable but compact and binary? If a big-endian machine stores it and a little-endian reads it (or vice versa), what goes in as 0x0102 (258) comes out as 0x0201 (513). Oops.

The solution is, instead of storing `shorts` and `ints` the way they're stored in memory, pick one method of storing (and loading) them, and stick to it. For example, several standards specify "network byte order," which is big-endian (most significant byte in the lower address). For example, if `s` is a short and `a` is an array of two `chars`, then the code

```
a[0] = (s >> 4) & 0xf;
a[1] = s & 0xf;
```

stores the value of `s` in the two bytes of `a`, in network byte order. This will happen if the program is running on a little-endian machine or on a big-endian machine.

You'll notice I haven't mentioned which machines are big-endian and which are little-endian. That's deliberate. If portability is important, you should write code that works either way. If efficiency is important, you usually should *still* write code that works either way. For example, there's a better way to implement the preceding code fragment on big-endian machines. However, a good compiler will generate machine code that takes advantage of that implementation, even for the portable C code it's given.

NOTE

The names "big-endian" and "little-endian" come from *Gulliver's Travels* by Jonathan Swift. On his third voyage, Gulliver meets people who can't agree how to eat hard-boiled eggs: big end first, or little end first.

"Network byte order" applies only to `int`, `short`, and `long` values. `char` values are, by definition, only one byte long, so there's no issue with them. There's no standard way to store `float` or `double` values.

## Cross Reference:

X.5: What is meant by high-order and low-order bytes?

X.6: How are 16- and 32-bit numbers stored?