In This Chapter:

- *Modularity*
- *Word Size*
- *Byte-Order Problem*
- *Alignment Problem*
- *NULL-Pointer Problem*
- *Filename Problems*
- *File Types*
- *Summary*
- *Answers to Chapter Questions*

26

Portability Problems

Wherein I spake of most disastrous changes,
Of moving accidents by flood and field,
Of hair-breadth 'scapes i' the
imminent deadly breadth...

—Shakespeare on program porting
Othello, Act 1, Scene 3

You've just completed work on your great masterpiece, a ray-tracing program that renders complex three-dimensional shaded graphics on a Linux cluster supercomputer using 30TB of memory and 5PB of disk space. What do you do when someone comes in and asks you to port this program to an IBM PC with 64MB of memory and 100GB of disk space? Killing him is out. Not only is it illegal, but it also is considered unprofessional. Your only choice is to whimper and start the port. It is during this process that you will find that your nice, working program exhibits all sorts of strange and mysterious problems.

C++ programs are supposed to be portable. However, C++ contains many machine-dependent features. Also, because of the vast difference between UNIX and MS-DOS/Windows, system dependencies can frequently be found in many programs. This chapter discusses some of the problems associated with writing truly portable programs as well as some of the traps you might encounter.

Modularity

One of the tricks to writing portable programs is to put all the nonportable code into a separate module. For example, screen handling differs greatly in MS-DOS/Windows and UNIX. To design a portable program, you'd have to write machine-specific screen-update modules.

For example, the HP-98752A terminal has a set of function keys labeled F1–F8. The PC also has a function-key set. The problem is that these keys don't send out the same set of codes. The HP sends "<esc>p<return>" for F1 and the PC sends "<null>". In this case, you would want to write a `get_code` routine that gets a character (or function key string) from the keyboard and translates function keys. Because the translation is different for both machines, a machine-dependent module is needed for each one. For the HP machine, you would put together the program with *main.cc* and *hp-tty.cc*, while for the PC you would use *main.cc* and *pc-tty.cc*.

Word Size

A `long int` is 32 or 64 bits, a `short int` is 16 bits,* and a normal `int` can be 16, 32 or 64 bits depending on the machine. This can lead to unexpected problems. For example, the following code works on a 32-bit UNIX system and Win32 systems, but fails when ported to MS-DOS:

```
int zip;

zip = 92126;
std::cout << "Zip code " << zip << '\n';
```

The problem is that on MS-DOS, `zip` is only 16 bits — too small for 92126. To fix the problem, we declare `zip` as a 32-bit integer:

```
long int zip;

zip = 92126;
std::cout << Zip code " << zip << '\n';
```

Now `zip` is 32 bits and can hold 92126.

Byte-Order Problem

A `short int` consists of two bytes. Consider the number 0x1234. The two bytes have the value 0x12 and 0x34. Which value is stored in the first byte? The answer is machine dependent.

This can cause considerable trouble when you try to write portable binary files. Sun SPARC machines use one type of byte order (ABCD), while Intel machines use another (BADC).

One solution to the problem of portable binary files is to avoid them. Put an option in your program to read and write ASCII files. ASCII offers the twin advantages of being far more portable as well as human readable.

---

* The C++ standard does not specify the actual size of `long int` or `short int`. However, on every machine I know of, a `long int` is 32 bits and a `short int` is 16 bits.

The disadvantage is that text files are larger. Some files may be too big for ASCII. In that case, the magic number at the beginning of a file may be useful. Suppose the magic number is 0x11223344 (a bad magic number, but a good example). When the program reads the magic number, it can check against the correct number as well as the byte-swapped version (0x22114433). The program can automatically fix the file problem:

```
const int MAGIC      = 0x11223344; // File identification number
const int SWAP_MAGIC = 0x22114433; // Magic number byte swapped

std::ifstream in_file;  // File containing binary data
long int magic;    // Magic number from file

in_file.open("data");

in_file.read((char *)&magic, sizeof(magic));

switch (magic) {
    case MAGIC:
        // No problem
        break;
    case SWAP_MAGIC:
        std::cout <<"Converting file, please wait\n";
        convert_file(in_file);
        break;
    default:
        std::cerr << "Error: Bad magic number " << magic << '\n’;
        exit (8);
}
```

Alignment Problem

Some computers limit the addresses that can be used for integers and other types of data. For example, the 68000 series requires that all integers start on a two-byte boundary. If you attempt to access an integer using an odd address, you generate an error. Some processors have no alignment rules, while some are even more restrictive, requiring integers to be aligned on a four-byte boundary.

Alignment restrictions are not limited to integers. Floating point numbers and pointers also must be aligned correctly.

C++ hides the alignment restrictions from you. For example, if you declare the following structure on a 68000:

```
struct funny {
    char    flag;   // Type of data following
    long int value; // Value of the parameter
};
```

C++ allocates storage for this structure as shown on the left in Figure 26-1.

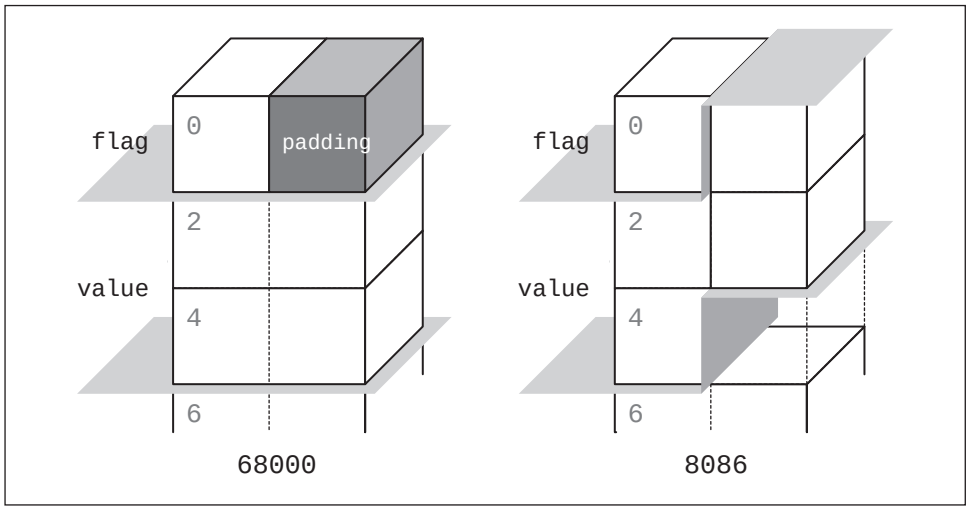

Figure 26-1. Structure on 68000 and 8086 architectures

On an old 8-bit oriented processor such as an 8086* with no alignment restrictions, this is allocated as shown on the right in Figure 26-1.

The problem is that the size of the structure changes from machine to machine. On a SPARC, the structure size is six bytes and on the 8086, it is five. So if you write a binary file containing 100 records on a SPARC, it will be 600 bytes long, while on an 8086 it will be only 500 bytes long. Obviously the file is not written the same way on both machines.

One way around this problem is to use ASCII files. As we have said before, there are many problems with binary files. Another solution is to explicitly declare a pad byte:

```
struct new_funny {
        char    flag;   // Type of data following
        char    pad;    // Not used
        long int value; // Value of the parameter
};
```

The pad character makes the field value align correctly on a SPARC computer while making the structure the correct size on an 8086-class machine.

Using pad characters is difficult and error-prone. For example, although `new_funny` is portable between machines with one- and two-byte alignment for 32-bit integers, it is not portable to any machine with a four-byte integer alignment.

* I know that the 8086 uses 16 bit registers, but if you look at the design, you can see that it's an 8-bit oriented processor.

NULL-Pointer Problem

Many programs and utilities were written using UNIX on VAX computers. On this computer, the first byte of any program is 0. Many programs written on this computer contain a bug: They use the null pointer as a string.

Example:

```
#define NULL 0

char *string;

string = NULL;
std::cout << "String is '" << string "'\n";
```

This is actually an illegal use of `string`. Null pointers should never be dereferenced. On the VAX, this error causes no problems. Because byte zero of the program is zero, `string` points to a null string. This is due to luck, not design.

On a VAX, this will print:

```
String is ''
```

On an old Celerity, the first byte of the program is a “Q.” When this program is run on a Celerity, it will print:

```
String is 'Q'
```

On other computers, this type of code can generate unexpected results. Many of the utilities ported from a VAX to a Celerity exhibited the “Q” bug.

Filename Problems

UNIX specifies files as `/root/sub/file` while MS-DOS/Windows uses `\root\sub\file`. When porting from UNIX to MS-DOS/Windows, file names must be changed. For example:

```
#ifndef __MSDOS__
#include <sys/stat.h> /* UNIX version of the file */
#else __MSDOS__
#include <sys\stat.h> /* DOS version of the file */
#endif __MSDOS__
```

Question 26-1: *Why does Example 25-1 work on UNIX, but when we run it in MS-DOS/Windows we get the message:*

```
oot
    ew      able:   file not found
```

Example 26-1. i

```
#ifndef __MSDOS__
#define NAME "/root/new/table"
#else /* __MSDOS__ */
#define NAME "\root\new\table"
#endif /* __MSDOS__ */

in_file.open(NAME);
if (in_file.bad()) {
    std::cout << NAME << ": file not found\n";
    exit(8);
}
```

File Types

The C language was invented on UNIX. In fact the `'\n'` character used to separate lines in C (and C++) is the character UNIX uses to separate lines. The Microsoft Windows operating system uses a different convention for end of line. So when you read and write text files on Microsoft Windows, the operating system must translate between the file end of line convention (<carriage-return> <line-feed>) and the C++ convention (`'\n'` or <line-feed>).

The O_BINARY flag is used to tell Microsoft Windows to turn off this feature and to read the file as is.

So to open a binary file on MS-DOS/Windows you *must* the statement:

```
file_descriptor = open("file", O_RDONLY|O_BINARY);
```

and to open a text file you *must* use a statement like:

```
file_descriptor = open("file", O_RDONLY);
```

The `O_BINARY` flag is required to distinguish the two file types.

On UNIX there is no need for an end-of-line fixup. So it doesn't matter if the file is text or binary, the results are the same. In fact, it is possible to open a binary file with the statement:

```
file_descriptor = open("file", O_RDONLY);
```

This is incorrect, because a binary file should be opened with `O_BINARY`. But this will work just fine on UNIX. The code will compile and execute correctly.

But when the program is ported to Microsoft Windows, things will go wrong. Because you told the system that this is a text file (no `O_BINARY`) it will perform the end of line fixup and modify the data coming in. This means that the binary data that you wanted to read unmodified is getting modified.

legacy code is particularly vulnerable to this problem. You see the early C programming systems didn't have a `O_BINARY` flag. The older programs just never used it. Unfortunately some of the old C programs got translated to C++ and became programs that someone wanted to port to operating systems such as Microsoft Windows.

The result is code with a lot of `O_BINARY` parameters missing. If you are doing a porting job, you should be aware of this problem.

Summary

It is possible to write portable programs in C++. Because C++ runs on many different types of machines that use different operating systems, it is not easy. However, if you keep portability in mind when creating the code, you can minimize the problems.

Porting four-letter words

Portability problems are not limited to programming. When *Practical C Programming* was translated to Japanese, the translator had a problem with one exercise: "Write a program that removes four-letter words from a file and replaces them with more acceptable equivalents."

The problem was, in Japanese, everything is a one-letter word. When the translator came to a phrase he couldn't directly translate, he did his best and also put in the English. But for the four-letter words, he decided to include the English as well as some additional help (Stars added. I don't use words like that.):

【練習問題 6】 テキストファイルの中で使われている言葉遣いを，きれいにするプログラムを書きなさい。このプログラムは，ファイル中に four-letter words(四文字語，卑猥な言葉, f***, c***, c***, c***, dumb, ***ho など) を見つけたら，それをもっとおだやかな言葉に置き換えるものです。

Answers to Chapter Questions

Answer 26-1: The problem is that C++ uses the backslash (\) as an escape character. The character `\r` is <return>, `\n` is <new line>, and `\t` is <tab>. What we really have for a name is:

```
<return>oot<new line>ew<tab>able
```

The name should be specified as:

```
#define NAME "\\root\\new\\table"
```

NOTE

The #include uses a filename, not a C++ string. While you must use double backslashes (\\) in a C++ string, you use single backslashes in an #include. The following two lines are both correct:

```
const string name("\\root\\new\\table");
#include "\root\new\defs.h"
```