Portability in C -- A Case Study

Matt Bishop

Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

Introduction

There are a large number of computers and operating systems in use today, and more and more programs are being moved from one computer or operating system to another. In fact, when you complete a program, your job is often only half done; since you almost certainly have access to other computers, you will want to make the program portable enough to run on all of those machines. Taking some care while writing the program can greatly reduce, or even eliminate, the difficulty of such ports.

In this article we will examine several C programming constructs that produce more portable programs. We will do this in three stages, looking first at dependencies upon the compiler, then at dependencies upon the operating system, and finally at dependencies upon the computer architecture. Each of these considerations affects different parts of the program.

The vehicle we shall use is a program called *trnum*. This program is used to number equations, tables, and any other text you like; the numbers are given to *trnum* as symbols and are converted to numbers by that program. It was written before UNIX System V was available, and was ported to that operating system with literally no changes. So, the goal of programs which can be ported without changing anything is not an impossible dream.

Compiler Dependencies

Compilers which implement C as defined in ''The C Programming Language -- Reference Manual'' (see **The C Programming Language** by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978; the **UNIX Programmer's Manual Volume 2** contains a short addendum) are pretty much the same; only a few features may cause problems. Four are as a result of an extension to the language in 1978 (and covered in the addendum to the reference manual); the remaining one arises from efficiency considerations, and is explicitly left undefined in the reference manual. We will deal first with the

extensions.

Trnum allows the user to specify the format of the counter, so it will be printed as capital or small Arabic letters, Roman numerals, or numbers. These are represented internally by the symbols *F1* (for numbers), *FA* (for capital letters), *Fa* (for small letters), *FI* (for capital Roman numerals), and *Fi* (for small Roman numerals). From a programmer's point of view, these may be used to represent formats in two ways.

Using the extension, one way is to create an *enumerated type* by saying

```
enum counter_format { F1, FA, Fa, FI, Fi };
```

and declaring the format field associated with each counter to be of type *enum counter_format*. But some compilers will not accept this as a legal C statement, so *trnum* does not use this C language feature. Instead, it defines *F1*, *FA*, *Fa*, *FI*, and *Fi* as macros and declares the format field to be an integer. This works with all C compilers, and in this case the application is simple enough so that the loss of the added type checking is far outweighed by portability considerations.

The three other areas in which the language has been extended are closely related. Structure assignment, passing structures as function parameters, and functions returning structures are now allowed. Depending on the computer, these may or may not increase the speed of the program (some computers can do ‘‘block moves’’ which are faster than the corresponding non-block moves). It is best to avoid passing structures as arguments or writing functions that return structures; instead, pass pointers to structures to and from functions. As for assignment, it is best to code that as a macro which may be replaced with a function call should the program be moved to a compiler which does not allow structure assignment. For example, the macro

```
/*
 * note a and b are pointers,
 * so if we have to convert this
 * to a function call, we can
 * just pass the pointers
 */
#define STRUCTURE_COPY(a,b)       (*(a) = *(b))
```

works quite nicely.

The other area in which compilers are very different is the order in which expressions are evaluated; the best known result of this is that functions producing side effects should *never* be used in expressions involving any variables or functions affected by the side effects. The order in which expressions are evaluated is left undefined by the C reference manual very deliberately, to allow the compiler writer to choose the most efficient method for the target machine. We shall look at two cases where this affects expressions, and how to work around this ambiguity.

The first example is the order in which function arguments are evaluated. In *trnum*, there is a function called *getnext*() which is most often called with the first character of a string as its first argument, and a pointer to the remainder of the string as its second argument. Hence, if *p* is the pointer to the beginning of the string, it is most tempting to write this call as:

```
getnext(*p++, p);
```

Now, suppose *p* points to the string ".y>". If the compiler evaluates function arguments beginning at the left, *getnext*() will be called as

```
getnext('.', "y>");
```

as we want. But, if the compiler evaluates function arguments beginning at the right (as do VAX and PDP-11 compilers), the call becomes

```
getnext('.', ".y>");
```

which is not what we expected. The only way to guarantee the order of evaluation is to use a temporary variable:

```
base = &p[1];
getnext(*p++, base);
```

This will work on all compilers.

As a second example of the dangers of assuming an order of evaluation, consider the expression

```
a[i] = b[i--];
```

which might be used as part of a *while* loop for copying strings (there are many such loops in *trnum*). If *i* is 1, *b[0]* is 'x', and *b[1]* is 'y', then this statement will assign 'y' to *a[1]* if the expression is evaluated from left to tight, but it will assign 'y' to *a[0]* if the expression is evaluated from right to left. Again, to get around this problem, move the decrement of *i* outside the assignment statement, and say

```
a[i] = b[i];
i--;
```

if the first interpretation is what you want, or

```
i--;
a[i] = b[i];
```

if the second interpretation is what you want.

Operating System Dependencies

Unlike changing C compilers, it is not possible to write C code that will not change at some level when moved to another operating system. The trick is to keep those changes confined to library functions, header files, or both. With properly-written programs, it should only be necessary to change the library functions or header files appropriately, and recompile and relink the program. In most cases, only recompilation and relinking will be necessary, since the majority of applications programs access the kernel using libraries provided with the operating system (such as the *standard input-output library*, also called *stdio*.)

The program *trnum* falls into this class of programs because its interaction with the operating system is done through the standard input-output library package. So long as this (relatively) standard interface exists, it can be compiled and linked without change, and the package will handle the details of interfacing the program's input and output with the operating system. Let us look at how this package hides operating system dependent details from *trnum*.

The *stdio* package associates with each file a collection of information such as how the file was opened (for reading, writing, or appending), the current offset of the file pointer into the file, and the file index number. This structure is hidden by defining a structure and macro similar to the following in the header file <*stdio.h*>:

```
struct _finfo {
        int _fileindex;         /* index into file table */
        unsigned int _mode;     /* open for read, write */
};
#define    FILE       struct _finfo
```

and now for each different operating system, the header file can be changed as required.

As another example, each operating system has its own system call for opening files. These may, or may not, be compatible with the system calls of other operating systems; for example, on V7 UNIX, the second argument to the *open* system call is **2** for appending to the file, but on 4.2 BSD UNIX, the second argument must be **8** to append to the file. Fortunately, the *stdio* package provides a uniform interface for opening a file; saying

```
fp = fopen(filename, "a");
```

will open the named file for appending, regardless of what the operating system call requires. Again, the details of interacting with the operating system are segregated from the rest of the program.

A far less common, but more dramatic, example of this occurs when a program that reads the contents of directory files is ported from V7 UNIX to 4.2 BSD UNIX. The directory formats are vastly different; for instance, V7 UNIX uses a fixed-length record for each file name in the directory, whereas 4.2 BSD UNIX uses variable-length records. Hence, the directory should be accessed only through a set of functions separate from the rest of the program. So, for example, use routines called *open_directory*, *read_directory*, and *close_directory* to open, look through, and close the directory file rather than putting the directory scanning code with the rest of the program code. Then, when the program is moved from a V7 UNIX system to a 4.2 BSD UNIX system, only the three directory accessing routines need be rewritten.

Another problem arises when you use library functions that the system provides. Sometimes these differ from system to system. The least dangerous of this is when the function returns one datatype for an operating system and another datatype for another operating system. In *trnum*, the function *sprintf* did this (returning an integer in SYSTEM V UNIX and a character pointer in 4.2 BSD UNIX.) There is no obvious way to determine what these functions are other than going to a manual, but you should avoid relying on return values of functions for which return values are not central to the purpose of the function. In *trnum*, for example, *sprintf* was used to format a string in core, and so

the return value was completely ignored; hence, when it was moved to SYSTEM V UNIX, there was never a problem.

Some library functions have no counterparts on other systems. Examples are the *search* functions provided by SYSTEM V UNIX. In this case, all that can be done is to write equivalent functions. A similar but more subtle form of this occurs when two different functions have the same name on different systems. Fortunately, neither of these occurred in *trnum*; the only thing that can be done about them is to check the manual of the operating system the program is being ported to.

Aside from localizing operating system dependencies in functions, there are two other considerations programmers must keep in mind -- defining and naming external variables. We will look at naming them first.

As stated in the previous section, most compilers impose a maximum on the number of significant characters in identifiers. However, the linker also uses these names to link the program modules together. On some systems, the maximum number of significant characters of identifiers the linker recognizes is different than the number the compiler recognizes. Furthermore, some linkers ignore the case of characters in identifiers. Hence, all external symbols should be unique within the first six characters, exclusive of case. (This seems to be the least common denominator of the restrictions.) As an example, the *trnum* program uses the function *issalnum*() to determine if a string is composed of alphanumeric characters; the name *isalnum_str*() (for ''is alphanumeric string'') would be more readable, but has the same first six characters as the function *isalnum*(); hence, this could produce a conflict.

The other problem with external variables arises when the keyword *extern* is omitted in a declaration. With many versions of the UNIX operating system, if an external variable is defined in several files, the linker just uses one definition and ignores the others. Many other UNIX systems will object that the variable is multiply defined, and fail to link the modules. Thus, in *trnum*, all variables that are referenced by routines in more than one file were defined in the file containing the main routine and simply declared as *extern* in all the others. For example, the variable *topt*, which is 1 when a table of numbers is to be generated and 0 when it is not, is declared as

```
int topt;          /* generate table of numbers? */
```

in the main file and as

```
extern int topt;/* generate table of numbers? */
```

in the other files. This prevents the linker from finding multiply defined variables.

Machine Dependencies

Several factors combine to create problems when moving a program from one machine to another. All of these spring from differences in the computer architectures and the C compiler's customizing object code to be as efficient as possible for that machine.

As an example, consider the representation of various datatypes. Assuming a specific number of bits for characters, short integers, integers, and long integers is a recipe for disaster, because this number may not be the same for all machines. Unfortunately, this type of assumption is very common, particularly when using bits as flags. For example, in *trnum*, the structure of the hash table used to store counters contains a word for miscellaneous flags; if the low bit is set, that entry is unused. It is very tempting to write the statement clearing the low bit as

```
flag &= 0177776;
```

on a PDP-11 (which has 16 bit words), but this will fail miserably on a VAX, which has 32 bit words (and hence will clear the high-order 16 bits as well.) However, writing

```
flag &= ~01;
```

will clear the bit on any machine, because the compiler will automatically change "~01" into a word with the low bit clear and all other bits set. The moral is to let the compiler worry about datatype lengths.

Another obnoxious problem arises when you assume two datatypes can be used interchangeably. Recall the function *issalnum*(); it takes a pointer to a string as its argument. Call this parameter *s*. Many programmers would tend to declare this function as

```
int issalnum(s)
{
    ...
```

and on many machines there would be no problem, since a pointer to a character is the same length as an integer. But on some machines, this would fail (with a core dump, if you are lucky) because the two are not interchangeable. The declaration should be (and in *trnum* is) written:

```
int issalnum(s)
char *s;
{
    ...
```

Related to this is the problem of type coercion. Consider that on some machines (such as a PDP-11), a character pointer may refer to any memory location but an integer pointer must point to an even address. Hence on a PDP-11, it is *not* safe to coerce a character pointer to an integer pointer and access the data as an integer. (On a PDP-11, in fact, this causes an error.) If such coercion must be done, make it into a subroutine so that it can be changed easily for different computers. This problem is particularly acute with storage allocators; the approach used in *trnum* is to define a new type with

```
typedef char *ALIGN;
```

so, when it is moved from a VAX to a PDP-11, the header file need only be changed to say

```
typedef int *ALIGN;
```

(in fact, this is done by a conditional compilation rather than changing the file.)

This brings up another point -- byte ordering. Some computers, such as the VAXen and PDP-11's, number bytes from right to left; others, such as the MC68000, number bytes from left to right. As a result, when bytes are read from integers, the order in which the bytes are read is machine-dependent. The same is true of bits being accessed. If the order is important, do the reading (or accessing) in a separate routine that can be modified should the program be moved to a machine that reads bytes in the opposite order.

Characters are also a source of frustration. A program cannot assume ASCII ordering, or even a contiguous alphabet (in EBCDIC, there are nonalphabetic characters interspersed within the alphabet.) Hence, constructs such as the following, to capitalize the character in *c*:

```
capital_c = c – 'a' + 'A';
```

should be avoided since they are non-portable. Instead, use the functions (or macros) defined in *<ctype.h>*; the above would be written

```
capital_c = toupper(c);
```

which will work on all computers, since *toupper* is defined appropriately on each computer.

Even when using ASCII characters, there are certain common assumptions which are quite dangerous. For instance, it is widely accepted that characters are all nonnegative, and the end of file marker is negative. Hence, the test

```
if ((c = getchar()) < 0){
    ...
```

is true only at the end of file. On a VAX, this is correct. But on a PDP-11, characters are really integers in the range −128 to 127 inclusive. In this case, the character will be sign-extended before the test is made; so, the above test will be true when the character has the high bit set, regardless of whether or not the end of file is reached. To be safe, always assume characters are sign-extended when converted to integers, and when using the standard input-output package, use the end of file marker explicitly, as in

```
if ((c = getchar()) == EOF){
    ...
```

which will work on all machines.

Related to this is the multicharacter constant. As characters are really short integers, many compilers allow several characters to be stored in one integer. However, because the order in which the characters are actually put into the integer is machine dependent (upon the way the computer numbers its bytes), this is an extremely non-portable construct. Use character strings instead to ensure portability.

Conclusion

There is one tool that is quite useful in checking for non-portable constructs: the program *lint*. This program will check for, and report, constructs which might cause problems when the program is ported to another machine or operating system. It also

reports expressions the values of which depend on the order of evaluation. (It may, or may not, report use of the extensions to C; that depends on whether or not the compiler recognizes them.) When run with the option **−p**, the standard input-output library function calls are also checked for calling errors; when **−c** is given type casts are very carefully checked; and when the flag **−a** is given, assignments that would cause values to be truncated (such as moving a *long* into an *integer*) are also reported. (Be aware the use of these flags varies from system to system; the above applies to 4.2 BSD UNIX only.) *Lint* will be examined more closely in a future issue of **The C Journal** magazine.

This paper has examined some of the ways programmers can make their programs more portable. As programs are moved from one machine to another, these considerations become vital to writing useful programs.

∞