Appendix D Portability Issues

The ANSI Standard

Portability doesn't happen by accident. It occurs when you adhere to a set of standards followed by other programmers and other compilers. For this reason, it's wise to choose a compiler that follows the standards for C programming set by the American National Standards Institute (ANSI). The ANSI committee sets standards for many areas, including other programming languages. Almost all C compilers provide the option of adhering to the ANSI standard.

The ANSI Keywords

The C language contains relatively few keywords. A *keyword* is a word that is reserved for a program command. The ANSI C keywords are listed in Table D.1.

Table D.1. The ANSI C keywords.

asm	auto	break
case	char	const
continue	default	do
double	else	enum
extern	float	for
goto	if	int
long	register	return
short	signed	sizeof
static	struct	switch
typedef	union	unsigned
void	volatile	while

Most compilers provide other keywords as well. Examples of compiler-specific keywords are near and huge. Although several compilers might use the same compiler-specific keywords, there is no guarantee that those keywords will be portable to every ANSI-standard compiler.

Case Sensitivity

Case sensitivity is an important issue in programming languages. Unlike some languages that ignore case, C is case-sensitive. This means that a variable named a is different from a variable named A. Listing D.1 illustrates the difference.

Listing D.1. Case sensitivity.

```
1:    /*=======================================================*
2:     * Program: listD01.c                                    *
3:     * Book:    Teach Yourself C in 21 Days                  *
4:     * Purpose: This program demonstrates case sensitivity   *
5:     *=======================================================*/
6:    #include <stdio.h>
7:    int main(void)
8:    {
9:      int   var1 = 1,
10:           var2 = 2;
11:     char  VAR1 = `A',
12:           VAR2 = `B';
13:     float Var1 = 3.3,
```

```
14:          Var2 = 4.4;
15:    int   xyz  = 100,
16:          XYZ  = 500;
17:
18:    printf( "\n\nPrint the values of the variables...\n" );
19:
20:    printf( "\nThe integer values:   var1 = %d, var2 = %d",
21:             var1, var2 );
22:    printf( "\nThe character values: VAR1 = %c, VAR2 = %c",
23:             VAR1, VAR2 );
24:    printf( "\nThe float values:     Var1 = %f, Var2 = %f",
25:             Var1, Var2 );
26:    printf( "\nThe other integers:   xyz = %d, XYZ = %d",
27:             xyz, XYZ );
28:
29:    printf( "\n\nDone printing the values!\n" );
30:
31:    return 0;
32:  }
Print the values of the variables...
The integer values:     var1 = 1, var2 = 2
The character values:   VAR1 = A, VAR2 = B
The float values:       Var1 = 3.300000, Var2 = 4.400000
The other integers:     xyz = 100, XYZ = 500
Done printing the values!
```

ANALYSIS: This program uses several variables with the same names. In lines 9 and 10, var1 and var2 are defined as integer values. In lines 11 and 12, the same variable names are used with different cases. This time, VAR1 and VAR2 are all-uppercase. In lines 13 and 14, a third set of declarations is made with the same names, but with another different case. This time, Var1 and Var2 are declared as float values. In each of these three sets of declarations, values are placed in the variables so that they can be printed later. The printing for these three sets of declarations occurs in lines 20 through 25. As you can see, the values placed in the variables are retained, and each value is printed.

Lines 15 and 16 declare two variables of the same type--integers--and the same name. The only difference between these two variables is that one is uppercase and the other is not. Each of these variables has its own value, which is printed on lines 26 and 27.

Although it's possible to use only case to differentiate variables, this isn't a practice to enter into lightly. Not all computer systems that have C compilers available are case-sensitive. Because of this, code might not be portable if only case is used to differentiate variables. For portable code, you should always ensure that variables are differentiated by something other than the case of the variable name.

Case sensitivity can cause problems in more than just the compiler. It can also cause problems with the linker. The compiler might be able to differentiate between variables with only case differences, but the linker might not.

Most compilers and linkers let you set a flag to cause case to be ignored. You should check your compiler to determine the flag that needs to be set. When you recompile a listing with variables differentiated by case only, you should get an error similar to the following. Of course, var1 would be whatever variable you're using:

```
listD01.c:

Error listD01.c 16: Multiple declaration for `var1' in function main
*** 1 errors in Compile ***
```

Portable Characters

Characters within the computer are represented as numbers. On an IBM PC or compatible, the letter A is represented by the number 65, and the letter a is represented by the number 97. These numbers come from an ASCII table (which can be found in Appendix A).

If you're writing portable programs, you can't assume that the ASCII table is the character translation table being used. A different table might be used on a different computer system. In other words, on a mainframe, character 65 might not be A.

WARNING: You must be careful when using character numerics. Character numerics might not be portable.

Two general rules apply to how a character set is to be defined. The first rule restricting the character set is that the size of a character's value can't be larger than the size of the char type. In an 8-bit system, 255 is the maximum value that can be stored in a single char variable. Because of this, you wouldn't have a character with a value greater than 255. If you're working on a machine with a 16-bit character, 65,535 is the maximum value for a character.

The second rule restricting the character set is that each character must be represented by a positive number. The portable characters within the ASCII character set are those from 1 to 127. The values from 128 to 255 aren't guaranteed to be portable. These extended characters can't be guaranteed because a signed character has only 127 positive values.

Guaranteeing ANSI Compatibility

The predefined constant __STDC__ is used to help guarantee ANSI compatibility. When the listing is compiled with ANSI compatibility set on, this constant is defined--generally as 1. It is undefined when ANSI compatibility isn't on.

Virtually every compiler gives you the option to compile with ANSI enforced. This is usually done either by setting a switch within the IDE (Integrated Development Environment) or by passing an additional parameter on the command line when compiling. By setting the ANSI on, you help ensure that the program will be portable to other compilers and platforms.

To compile a program using Borland's Turbo C command line, you would enter the following on the command line:

```
TCC -A program.c
```

If you're compiling with a Microsoft compiler, you would enter

```
CL /Ze program.c
```

NOTE: Most compilers with Integrated Development Environments (IDEs) provide an ANSI option. By selecting the ANSI option, you are virtually guaranteed ANSI compatibility.

The compiler then provides additional error checking to ensure that ANSI rules are met. In some cases, there are errors and warnings that are no longer checked. An example is prototype checking. Most compilers display warnings if a function isn't prototyped before it is used; however, the ANSI standards don't require this. Because ANSI doesn't require the prototypes, you might not receive the required prototype warnings.

Avoiding the ANSI Standard

There are several reasons why you wouldn't want to compile your program with ANSI compatibility on. The most common reason involves taking advantage of your compiler's added features. Many features, such as special screen-handling functions, either aren't covered within the ANSI standard or might be compiler-specific. If you decide to use these compiler-specific features, you won't want the ANSI flag set. In addition, if you use these features, you might eliminate the portability of your program. Later in this chapter, you'll see a way around this limitation.

DO use more than just case to differentiate variable names.

DON'T assume numeric values for characters.

Using Portable Numeric Variables

The numeric values that can be stored in a specific variable type might not be consistent across compilers. Only a few rules are defined within the ANSI standard regarding the numeric values that can be stored in each variable type. On Day 3, "Storing Data: Variables and Constants," Table 3.2 presented the values typically stored in IBM-compatible PCs. However, these values aren't guaranteed.
The following rules apply to variable types:

- A character (char) is the smallest data type. A character variable (type char) will be one byte.
- A short variable (type short) will be smaller than or equal to an integer variable (type int).
- An integer variable (type int) will be smaller than or equal to the size of a long variable (type long).
- An unsigned integer variable (type unsigned) is equal to the size of a signed integer variable (type int).
- A float variable (type float) will be less than or equal to the size of a double variable (type double).

Listing D.2 presents a commonly used way to print the size of the variables based on the machine on which the program is compiled.

Listing D.2. Printing the size of the data types.

```
1:   /*==========================================================*
2:    * Program: listD02.c                                       *
3:    * Book:    Teach Yourself C in 21 Days                     *
4:    * Purpose: This program prints the sizes of the variable   *
5:    *          types of the machine the program is compiled on *
6:    *==========================================================*/
7:   #include <stdio.h>
8:   int main(void)
9:   {
10:    printf( "\nVariable Type Sizes" );
11:    printf( "\n=========================" );
12:    printf( "\nchar             %d", sizeof(char) );
13:    printf( "\nshort            %d", sizeof(short) );
14:    printf( "\nint              %d", sizeof(int) );
15:    printf( "\nfloat            %d", sizeof(float) );
16:    printf( "\ndouble           %d", sizeof(double) );
17:
18:    printf( "\n\nunsigned char    %d", sizeof(unsigned char) );
19:    printf( "\nunsigned short   %d", sizeof(unsigned short) );
20:    printf( "\nunsigned int     %d\n", sizeof(unsigned int) );
21:
22:    return 0;
23:  }
Variable Type Sizes
=========================
char             1
short            2
int              2
float            4
double           8
unsigned char    1
unsigned short   2
unsigned int     2
```

ANALYSIS: As you can see, the sizeof() operator is used to print the size in bytes of each variable type. The output shown is based on the program's being compiled on a 16-bit IBM-compatible PC with a 16-bit compiler. If compiled on a different machine or with a different compiler, the sizes might be different. For example, a 32-bit compiler on a 32-bit machine might yield four bytes rather than two for the size of an integer.

Maximum and Minimum Values

If different machines have variable types that are different sizes, how do you know what values can be stored? It depends on the number of bytes that make up the data type and whether the variable is signed or unsigned. On Day 3, Table 3.2 shows the different values you can store based on the number of bytes. The maximum and minimum values that can be stored for integral types, such as integers, are based on the bits. For floating values, such as floats and doubles, larger values can be stored at the cost of precision. Table D.2 shows both integral-variable and floating-decimal values.

Table D.2. Possible values based on byte size.

Number	Unsigned	Signed	Signed
of Bytes	Maximum	Minimum	Maximum
Integral Types			
1	255	-128	127
2	65,535	-32,768	32,767
4	4,294,967,295	-2,147,483,648	2,147,438,647
8		1.844674 * E19	

Floating Decimal Sizes

4*	3.4 E-38	3.4 E38
8**	1.7 E-308	1.7 E308
10***	3.4 E-4932	1.1 E4932

*Precision taken to 7 digits.
**Precision taken to 15 digits.
***Precision taken to 19 digits.

Knowing the maximum value based on the number of bytes and variable type is good; however, as you saw earlier, you don't always know the number of bytes in a portable program. In addition, you can't be completely sure of the level of precision used in floating-point numbers. Because of this, you must be careful about what numbers you assign to variables. For example, assigning the value of 3,000 to an integer variable is a safe assignment, but what about assigning 100,000? If it's an unsigned integer on a 16-bit machine, you'll get unusual results because the maximum value is 65,535. If a 4-byte integer is being used, assigning 100,000 would be okay.

WARNING: You aren't guaranteed that the values in Table D.2 are the same for every compiler. Each compiler might choose a slightly different number. This is especially true with the floating-point numbers, which can have different levels of precision. Tables D.3 and D.4 provide a compatible way of using these numbers.

ANSI has standardized a set of defined constants that are to be included in the header files LIMITS.H and FLOAT.H. These constants define the number of bits within a variable type. In addition, they define the minimum and maximum values. Table D.3 lists the values defined in LIMITS.H. These values apply to the integral data types. The values in FLOAT.H contain the values for the floating-point types.

Table D.3. The ANSI-defined constants within LIMITS.H.

Constant	Value
CHAR_BIT	Character variable's number of bits.
CHAR_MIN	Character variable's minimum value (signed).
CHAR_MAX	Character variable's maximum value (signed).

SCHAR_MIN	Signed character variable's minimum value.
SCHAR_MAX	Signed character variable's maximum value.
UCHAR_MAX	Unsigned character's maximum value.
INT_MIN	Integer variable's minimum value.
INT_MAX	Integer variable's maximum value.
UINT_MAX	Unsigned integer variable's maximum value.
SHRT_MIN	Short variable's minimum value.
SHRT_MAX	Short variable's maximum value.
USHRT_MAX	Unsigned short variable's maximum value.
LONG_MIN	Long variable's minimum value.
LONG_MAX	Long variable's maximum value.
ULONG_MAX	Unsigned long variable's maximum value.

Table D.4. The ANSI-defined constants within FLOAT.H.

Constant	**Value**
FLT_DIG	Precision digits in a variable of type float.
DBL_DIG	Precision digits in a variable of type double.
LDBL_DIG	Precision digits in a variable of type long double.
FLT_MAX	Float variable's maximum value.
FLT_MAX_10_EXP	Float variable's exponent maximum value (base 10).
FLT_MAX_EXP	Float variable's exponent maximum value (base 2).
FLT_MIN	Float variable's minimum value.
FLT_MIN_10_EXP	Float variable's exponent minimum value (base 10).
FLT_MIN_EXP	Float variable's exponent minimum value (base 2).
DBL_MAX	Double variable's maximum value.
DBL_MAX_10_EXP	Double variable's exponent maximum value (base 10).
DBL_MAX_EXP	Double variable's exponent maximum value (base 2).
DBL_MIN	Double variable's minimum value.
DBL_MIN_10_EXP	Double variable's exponent minimum value (base 10).
DBL_MIN_EXP	Double variable's exponent minimum value (base 2).
LDBL_MAX	Long double variable's maximum value.
LDBL_MAX_10_DBL	Long double variable's exponent maximum value (base 10).
LDBL_MAX_EXP	Long double variable's exponent maximum value (base 2).
LDBL_MIN	Long double variable's minimum value.
LDBL_MIN_10_EXP	Long double variable's exponent minimum value (base 10).
LDBL_MIN_EXP	Long double variable's exponent minimum value (base 2).

The values in Tables D.3 and D.4 can be used when storing numbers. Ensuring that a number is above or equal to the minimum constant and less than or equal to the maximum constant ensures that the listing will be portable. Listing D.3 prints the values

stored in the ANSI-defined constants, and Listing D.4 demonstrates the use of some of these constants. The output may be slightly different depending on the compiler used.

Listing D.3. Printing the values stored in the ANSI-defined constants.

```
1:   /*=======================================================*
2:    * Program:  listD03.c                                  *
3:    * Book:     Teach Yourself C in 21 Days                *
4:    * Purpose:  Display of defined constants.              *
5:    *=======================================================*/
6:   #include <stdio.h>
7:   #include <float.h>
8:   #include <limits.h>
9:
10:  int main( void )
11:  {
12:      printf( "\n CHAR_BIT        %d ", CHAR_BIT );
13:      printf( "\n CHAR_MIN        %d ", CHAR_MIN );
14:      printf( "\n CHAR_MAX        %d ", CHAR_MAX );
15:      printf( "\n SCHAR_MIN       %d ", SCHAR_MIN );
16:      printf( "\n SCHAR_MAX       %d ", SCHAR_MAX );
17:      printf( "\n UCHAR_MAX       %d ", UCHAR_MAX );
18:      printf( "\n SHRT_MIN        %d ", SHRT_MIN );
19:      printf( "\n SHRT_MAX        %d ", SHRT_MAX );
20:      printf( "\n USHRT_MAX       %d ", USHRT_MAX );
21:      printf( "\n INT_MIN         %d ", INT_MIN );
22:      printf( "\n INT_MAX         %d ", INT_MAX );
23:      printf( "\n UINT_MAX        %ld ", UINT_MAX );
24:      printf( "\n LONG_MIN        %ld ", LONG_MIN );
25:      printf( "\n LONG_MAX        %ld ", LONG_MAX );
26:      printf( "\n ULONG_MAX       %e ", ULONG_MAX );
27:      printf( "\n FLT_DIG         %d ", FLT_DIG );
28:      printf( "\n DBL_DIG         %d ", DBL_DIG );
29:      printf( "\n LDBL_DIG        %d ", LDBL_DIG );
30:      printf( "\n FLT_MAX         %e ", FLT_MAX );
31:      printf( "\n FLT_MIN         %e ", FLT_MIN );
32:      printf( "\n DBL_MAX         %e ", DBL_MAX );
33:      printf( "\n DBL_MIN         %e \n", DBL_MIN );
34:
35:      return(0);
36:  }
```

```
CHAR_BIT        8
CHAR_MIN        -128
CHAR_MAX        127
SCHAR_MIN       -128
SCHAR_MAX       127
UCHAR_MAX       255
SHRT_MIN        -32768
SHRT_MAX        32767
USHRT_MAX       -1
INT_MIN         -32768
INT_MAX         32767
```

```
UINT_MAX         65535
LONG_MIN         -2147483648
LONG_MAX         2147483647
ULONG_MAX        3.937208e-302
FLT_DIG          6
DBL_DIG          15
LDBL_DIG         19
FLT_MAX          3.402823e+38
FLT_MIN          1.175494e-38
DBL_MAX          1.797693e+308
DBL_MIN          2.225074e-308
```

NOTE: Output values will vary from compiler to compiler; therefore, your output may differ from the output shown here.

ANALYSIS: [endd] Listing D.3 is straightforward. The program consists of printf() function calls. Each function call prints a different defined constant. You'll notice the conversion character used (in this case, %d) depends on the type of value being printed. This listing provides a synopsis of what values your compiler used. You could also have looked in the FLOAT.H and LIMITS.H header files to see whether these values had been defined. This program should make determining the constant values easier.

Listing D.4. Using the ANSI-defined constants.

```
1:   /*===========================================================*
2:    * Program: listD04.c                                        *
3:    * Book:    Teach Yourself C in 21 Days                      *
4:    *                                                           *
5:    * Purpose: To use maximum and minimum constants.            *
6:    * Note:    Not all valid characters are displayable to the *
7:    *          screen!                                          *
8:    *===========================================================*/
9:
10:  #include <float.h>
11:  #include <limits.h>
12:  #include <stdio.h>
13:
14:  int main( void )
15:  {
16:      unsigned char ch;
17:      int  i;
18:
19:      printf( "Enter a numeric value.");
20:      printf( "\nThis value will be translated to a character.");
21:      printf( "\n\n==> " );
22:
23:      scanf("%d", &i);
24:
25:      while( i < 0 || i > UCHAR_MAX )
26:      {
27:         printf("\n\nNot a valid value for a character.");
28:         printf("\nEnter a value from 0 to %d ==> ", UCHAR_MAX);
29:
30:         scanf("%d", &i);
31:      }
```

```
32:      ch = (char) i;
33:
34:      printf("\n\n%d is character %c\n", ch, ch );
35:
36:      return(0);
37:  }
Enter a numeric value.
This value will be translated to a character.
==> 5000
Not a valid value for a character.
Enter a value from 0 to 255 ==> 69
69 is character E
```

ANALYSIS: [Listing D.4 shows the UCHAR_MAX constant in action. The first new items you should notice are the includes in lines 10 and 11. As stated earlier, these two include files contain the defined constants. If you're questioning the need for FLOAT.H to be included in line 10, you're doing well. Because none of the decimal point constants are being used, the FLOAT.H header file isn't needed. Line 11, however, is needed. This is the header file that contains the definition of UCHAR_MAX that is used later in the listing.

Lines 16 and 17 declare the variables that will be used by the listing. An unsigned character, ch, is used along with an integer variable, i. When the variables are declared, several print statements are issued to prompt the user for a number. Notice that this number is entered into an integer. Because an integer is usually capable of holding a larger number, it is used for the input. If a character variable were used, a number that was too large would wrap to a number that fits a character variable. This can easily be seen by changing the i in line 23 to ch.

Line 25 uses the defined constant to see whether the entered number is greater than the maximum for an unsigned character. We are comparing to the maximum for an unsigned character rather than an integer because the program's purpose is to print a character, not an integer. If the entered value isn't valid for a character--or, more specifically, for an unsigned character--the user is told the proper values that can be entered (line 28) and is asked to enter a valid value.

Line 32 casts the integer to a character value. In a more complex program, you might find that switching to the character variable is easier than continuing with the integer. This can help prevent reallocating a value that isn't valid for a character into the integer variable. For this program, the line that prints the resulting character, line 34, could just as easily have used i rather than ch.

Classifying Numbers

In several instances, you'll want to know information about a variable. For instance, you might want to know whether the information is numeric, a control character, an uppercase character, or any of nearly a dozen different classifications. There are two different ways to check some of these classifications. Consider Listing D.5, which demonstrates one way of determining whether a value stored in a character is a letter of the alphabet.

Listing D.5. Is the character a letter of the alphabet?

```
1:   /*=======================================================*
2:    * Program: listD05.c                                    *
3:    * Purpose: This program may not be portable due to the  *
4:    *          way it uses character values.                *
5:    *=======================================================*/
6:   #include <stdio.h>
7:   int main(void)
8:   {
9:     unsigned char x = 0;
10:    char trash[256];              /* used to remove extra keys */
11:    while( x != `Q' && x != `q' )
12:    {
13:       printf( "\n\nEnter a character (Q to quit) ==> " );
14:
15:       x = getchar();
16:
17:       if( x >= `A' && x <= `Z')
18:       {
19:          printf( "\n\n%c is a letter of the alphabet!", x );
```

```
20:          printf("\n%c is an uppercase letter!", x );
21:       }
22:       else
23:       {
24:          if( x >= `a' && x <= `z')
25:          {
26:             printf( "\n\n%c is a letter of the alphabet!", x );
27:             printf("\n%c is a lowercase letter!", x );
28:          }
29:          else
30:          {
31:             printf( "\n\n%c is not a letter of the alphabet!", x );
32:          }
33:       }
34:       gets(trash); /* eliminates enter key */
35:    }
36:    printf("\n\nThank you for playing!\n");
37:    return;
38:  }
Enter a character (Q to quit) ==> A
A is a letter of the alphabet!
A is an uppercase letter!
Enter a character (Q to quit) ==> f
f is a letter of the alphabet!
f is a lowercase letter!
Enter a character (Q to quit) ==> 1
1 is not a letter of the alphabet!
Enter a character (Q to quit) ==> *
* is not a letter of the alphabet!
Enter a character (Q to quit) ==> q
q is a letter of the alphabet!
q is a lowercase letter!
Thank you for playing!
```

ANALYSIS: This program checks to see whether a letter is between the uppercase letter A and the uppercase letter Z. In addition, it checks to see whether it is between the lowercase a and the lowercase z. If the letter is between one of these two ranges, you would think you could assume that the letter is alphabetic. This is a bad assumption! There is no standard for the order in which characters are stored. If you're using the ASCII character set, you can get away with using the character ranges; however, your program isn't guaranteed portability. To guarantee portability, you should use a character-classification function.

There are several character-classification functions. Each is listed in Table D.5 with what it checks for. These functions return 0 if the given character doesn't meet the check; otherwise, they return a value other than 0.

Table D.5. Character-classification functions.

Function	Description
isalnum()	Checks to see whether the character is alphanumeric.
isalpha()	Checks to see whether the character is alphabetic.
iscntrl()	Checks to see whether the character is a control character.
isdigit()	Checks to see whether the character is a decimal digit.
isgraph()	Checks to see whether the character is printable (space is an exception).
islower()	Checks to see whether the character is lowercase.
isprint()	Checks to see whether the character is printable.
ispunct()	Checks to see whether the character is a punctuation character.
isspace()	Checks to see whether the character is a whitespace character.

isupper()	Checks to see whether the character is uppercase.
isxdigit()	Checks to see whether the character is a hexadecimal digit.

With the exception of an equality check, you should never compare the values of two different characters. For example, you could check to see whether the value of a character variable is equal to `A', but you wouldn't want to check to see whether the value of a character is greater than `A'.

```
if( X > `A' )      /* NOT PORTABLE!! */
...
if( X == `A' )     /* PORTABLE */
...
```

Listing D.6 is a rewrite of Listing D.5. Instead of using range checks, the appropriate character classification values are used. Listing D.6 is a much more portable program.

Listing D.6. Using character-classification functions.

```
1:   /*==========================================================*
2:    * Program: listD06.c                                       *
3:    * Book:    Teach Yourself C in 21 Days                     *
4:    * Purpose: This program is an alternative approach to     *
5:    *          the same task accomplished in Listing D.5.      *
6:    *          This program has a higher degree of portability!  *
7:    *==========================================================*/
8:   #include <ctype.h>
9:
10:  int main(void)
11:  {
12:    unsigned char x = 0;
13:    char trash[256];                /* use to flush extra keys */
14:    while( x != `Q' && x != `q' )
15:    {
16:       printf( "\n\nEnter a character (Q to quit) ==> " );
17:
18:       x = getchar();
19:
20:       if( isalpha(x) )
21:       {
22:          printf( "\n\n%c is a letter of the alphabet!", x );
23:          if( isupper(x) )
24:          {
25:             printf("\n%c is an uppercase letter!", x );
26:          }
27:          else
28:          {
29:             printf("\n%c is a lowercase letter!", x );
30:          }
31:       }
32:       else
33:       {
34:          printf( "\n\n%c is not a letter of the alphabet!", x );
35:       }
36:       gets(trash);   /* get extra keys */
37:    }
38:    printf("\n\nThank you for playing!\n");
39:    return(0);
40:  }
Enter a character (Q to quit) ==> z
z is a letter of the alphabet!
z is a lowercase letter!
Enter a character (Q to quit) ==> T
T is a letter of the alphabet!
```

```
T is an uppercase letter!
Enter a character (Q to quit) ==> #
# is not a letter of the alphabet!
Enter a character (Q to quit) ==> 7
7 is not a letter of the alphabet!
Enter a character (Q to quit) ==> Q
Q is a letter of the alphabet!
Q is an uppercase letter!
Thank you for playing!
```

ANALYSIS: The outcome should look virtually identical to that for Listing D.5--assuming you ran the program with the same values. This time, instead of using range checks, the character-classification functions were used. Notice that line 8 includes the CTYPE.H header file. When this is included, the classification functions are ready to go. Line 20 uses the isalpha() function to ensure that the character entered is a letter of the alphabet. If it is, a message is printed in line 22 stating that fact. Line 23 then checks to see whether the character is uppercase with the isupper() function. If the character is uppercase, a message is printed in line 25; otherwise, the message in line 29 is printed. If the character isn't a letter of the alphabet, a message is printed in line 34. Because the while loop starts in line 14, the program continues until Q or q is pressed. You might think line 14 detracts from the portability of this program, but that is incorrect. Remember that equality checks for characters are portable, and inequality checks aren't portable. "Not equal to" and "equal to" are both equality checks.

DON'T use numeric values when determining maximums for variables. Use the defined constants if you're writing a portable program.

DO use the character classification functions when possible.
DO remember that "!=" is considered an equality check.

Converting a Character's Case: A Portability Example

A common practice in programming is to convert the case of a character. Many people write a function similar to the following:

```
char conv_to_upper( char x )
{
if( x >= `a' && x <= `z' )
{
x -= 32;
}
return( x )
}
```

As you saw earlier, this if statement might not be portable. The following is an update function with the if statement updated to the portable functions presented in the preceding section:

```
char conv_to_upper( char x )
{
if( isalpha( x ) && islower( x ) )
{
x -= 32;
}
return( x )
}
```

This example is better than the previous listing in terms of portability; however, it still isn't completely portable. This function assumes that the uppercase letters are a numeric value that is 32 less than the lowercase letters. This is true if the ASCII character set is used. In the ASCII character set, `A' + 32 equals `a'; however, this isn't necessarily true on every system. This is especially untrue on non-ASCII character systems.

Two ANSI standard functions take care of switching the case of a character. The toupper() function converts a lowercase character to uppercase; the lowercase() function converts an uppercase character to lowercase. The previous function would look like this when rewritten:

```
toupper();
```

As you can see, this is a function that already exists. In addition, this function is defined by ANSI standards, so it should be portable.

Portable Structures and Unions

When using structures and unions, care must also be exercised if portability is a concern. Word alignment and the order in which members are stored are two areas of incompatibility that can occur when working with these constructs.

Word Alignment

Word alignment is an important factor in the portability of a structure. Word alignment is the aligning of data on a word boundary. A *word* is a set number of bytes. A word usually is equivalent to the size of the processor on the computer being used. For example, an IBM 16-bit PC generally has a two-byte word. Two bytes equals 16 bits.

An example will make this easy to understand. Consider the following structure. Using two-byte integers and one-byte characters, determine how many bytes of storage are needed to store the structure.

```
struct struct_tag {
int    x;    /* ints will be 2 bytes */
char   a;    /* chars are 1 byte */
int    y;
char   b;
int    z;
} sample = { 100, `A', 200, `B', 300);
```

Adding up the integers and the characters, you might come up with eight bytes for the amount of storage. This answer could be right. It also could be wrong! If word alignment is on, this structure will take 10 bytes of storage. Figures D.1 and D.2 illustrate how this structure would be stored in memory.

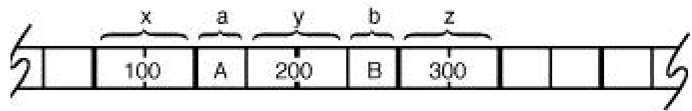

Figure D-1. Word alignment is off.

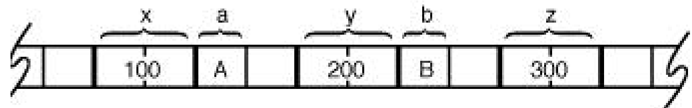

Figure D-2. Word alignment is on.

A program can't assume that the word alignment will be the same or that it will be on or off. The members could be aligned on every two bytes, four bytes, or eight bytes. You can't assume that you know.

Reading and Writing Structures

When reading or writing structures, you must be cautious. It's best to never use a literal constant for the size of a structure or union. If you're reading or writing structures to a file, the file probably won't be portable. This means you need to concentrate only on making the program portable. The program then needs to read and write the data files specific to the machine compiled on. The following is an example of a read statement that would be portable:

```
fread( &the_struct, sizeof( the_struct ), 1, filepointer );
```

As you can see, the sizeof command is used instead of a literal. Regardless of whether byte alignment is on or off, the correct number of bytes should be read.

Structure Order

When you create a structure, you might assume that the members will be stored in the order in which they are listed. There isn't a standard stating that a certain order must be followed. Because of this, you can't make assumptions about the order of information within a structure.

Preprocessor Directives

On Day 21, "Advanced Compiler Use," you learned about several preprocessor directives you can use. Several preprocessor directives have been defined in the ANSI standards. You use two of these all the time: #include and

#define. Several other preprocessor directives are in the ANSI standards. The additional preprocessor directives available under the ANSI guidelines are listed in Table D.6.

Table D.6. ANSI standard preprocessor directives.

#define	#if
#elif	#ifdef
#else	#ifndef
#endif	#include
#error	#pragma

Using Predefined Constants

Every compiler comes with predefined constants. A majority of these are typically compiler-specific. This means that they probably won't be portable from one compiler to the next. However, several predefined constants are defined in the ANSI standards. The following are some of these constants:
Constant Description
__DATE__ This is replaced by the date at the time the program is compiled. The date is in the form of a literal string (text enclosed in double quotes). The format is "Mmm DD, YYYY". For example, January 1, 1998 would be "Jan 1, 1998".
__FILE__ This is replaced with the name of the source file at the time of compilation. This will be in the form of a literal string.
__LINE__ This will be replaced with the number of the line on which __LINE__ appears in the source code. This will be a numeric decimal value.
__STDC__ This literal will be defined as 1 if the source file is compiled with the ANSI standard. If the source file wasn't compiled with the ANSI flag set, this value will be undefined.
__TIME__ This is replaced with the time that the program is compiled. This time is in the form of a literal string (text enclosed in double quotes). The format is "HH:MM:SS". An example would be "12:15:03".

Using Non-ANSI Features in Portable Programs

A program can use constants and other commands that aren't ANSI-defined and still be portable. You accomplish this by ensuring that the constants are used only if compiled with a compiler that supports the features used. Most compilers provide defined constants that you can use to identify them. By setting up areas of the code that are supportive for each of the compilers, you can create a portable program. Listing D.7 demonstrates how this can be done.

Listing D.7. A portable program with compiler specifics.

```
1:   /*=======================================================*
2:    * Program: listD07.c                                    *
3:    * Purpose: This program demonstrates using defined      *
4:    *          constants for creating a portable program.   *
5:    * Note:    This program gets different results with     *
6:    *          different compilers.                         *
7:    *=======================================================*/
8:   #include <stdio.h>
9:   #ifdef _WINDOWS
10:
11:  #define STRING "DOING A WINDOWS PROGRAM!\n"
12:
13:  #else
14:
15:  #define STRING "NOT DOING A WINDOWS PROGRAM\n"
16:
17:  #endif
18:
19:  int main(void)
```

```
20:  {
21:     printf( "\n\n") ;
22:     printf( STRING );
23:
24:  #ifdef _MSC_VER
25:
26:     printf( "\n\nUsing a Microsoft compiler!" );
27:     printf( "\n   Your Compiler version is %s\n", _MSC_VER );
28:
29:  #endif
30:
31:  #ifdef __TURBOC__
32:
33:     printf( "\n\nUsing the Turbo C compiler!" );
34:     printf( "\n   Your compiler version is %x\n", __TURBOC__ );
35:
36:  #endif
37:
38:  #ifdef __BORLANDC__
39:
40:     printf( "\n\nUsing a Borland compiler!\n" );
41:
42:  #endif
43:
44:     return(0);
45:  }
```

Here's the output you'll see when you run the program using a Turbo C for DOS 3.0 compiler:

```
NOT DOING A WINDOWS PROGRAM
Using the Turbo C compiler!
   Your compiler version is 300
```

Here's the output you'll see when you run the program using a Borland C++ compiler under DOS:

```
NOT DOING A WINDOWS PROGRAM
Using a Borland compiler!
```

Here's the output you'll see when you run the program using a Microsoft compiler under DOS:

```
NOT DOING A WINDOWS PROGRAM
Using a Microsoft compiler!
   Your compiler version is >>
```

ANALYSIS: This program takes advantage of defined constants to determine information about the compiler being used. In line 9, the #ifdef preprocessor directive is used. This directive checks to see whether the following constant has been defined. If the constant has been defined, the statements following the #ifdef are executed until an #endif preprocessor directive is reached. In the case of line 9, a determination is made as to whether _WINDOWS has been defined. An appropriate message is applied to the constant STRING. Line 22 then prints this string, which states whether or not this listing has been compiled as a Windows program.

Line 24 checks to see whether _MSC_VER has been defined. _MSC_VER is a constant that contains the version number of a Microsoft compiler. If a compiler other than a Microsoft compiler is used, this constant won't be defined. If a Microsoft compiler is used, this will be defined with the compiler's version number. Line 27 will print this compiler version number after line 26 prints a message stating that a Microsoft compiler was used.

Lines 31 through 36 and lines 38 through 42 operate in a similar manner. They check to see whether Borland's Turbo C or Borland's professional compiler were used. The appropriate message is printed based on these constants.

As you can see, this program determines which compiler is being used by checking the defined constants. The object of the program--to print a message stating which compiler is being used--is the same regardless of which compiler is used. If you're aware of the systems that you will be porting, you can put compiler-specific commands into the code. If you do use compiler-specific commands, you should ensure that the appropriate code is provided for each compiler.

ANSI Standard Header Files

Several header files that can be included are set by the ANSI standards. It's good to know which header files are ANSI standard, because these can be used in creating portable programs. Appendix E, "Common C Functions," contains the ANSI header files along with a list of their functions.

Summary

This appendix exposed you to a great deal of material. This information centered on portability. C is one of the most portable languages--if not *the* most portable. Portability doesn't happen by accident. ANSI standards have been created to ensure that C programs can be ported from one compiler to another and from one computer system to another. You should consider several areas when writing portable code. These areas include variable case, choosing which character set to use, using portable numerics, ensuring variable sizes, comparing characters, using structures and unions, and using preprocessor directives and preprocessor constants. This appendix ended with a discussion of how to incorporate compiler specifics into a portable program.

Q&A

Q How do you write portable graphics programs?
A ANSI doesn't define any real standards for programming graphics. Graphics pro-gramming is more machine-dependent than other programming areas, so it can be somewhat difficult to write portable graphics programs.
Q Should you always worry about portability?
A No, it's not always necessary to consider portability. Some programs you write will be used only by you on your system. In addition, some programs won't be ported to a different computer system. Because of this, some nonportable functions, such as system(), can be used that wouldn't be used in portable programs.
Q Are comments portable if they are done with // instead of /* and */?
A No. Forward-slash comments come from C++. Many C programmers now use these comments. Although they will most likely be a standard in the future, you might find that some current C compilers don't support them.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.
Answers are not provided for the following quiz questions and exercises.

Quiz

1. Which is more important: efficiency or maintainability?
2. What is the numeric value of the letter a?
3. What is guaranteed to be the largest unsigned character value on your system?
4. What does ANSI stand for?
5. Are the following variable names valid in the same C program?
int lastname,
LASTNAME,
LastName,
Lastname;
6. What does isalpha() do?
7. What does isdigit() do?
8. Why would you want to use functions such as isalpha() and isdigit()?
9. Can structures be written to disk without worrying about portability?
10. Can __TIME__ be used in a printf() statement to print the current time in a program? Here's an example:

```
printf( "The Current Time is:  %s", __TIME__ );
```

Exercises

1. BUG BUSTER: What, if anything, is wrong with the following function?

```
void Print_error( char *msg )
{
    static int ctr = 0,
               CTR = 0;
    printf("\n" );
    for( ctr = 0; ctr < 60; ctr++ )
    {
```

```
        printf("*");
    }
    printf( "\nError %d, %s - %d: %s.\n", CTR,
           __FILE__, __LINE__, msg );
    for( ctr = 0; ctr < 60; ctr++ )
    {
        printf("*");
    }
}
```

2. Write a function that verifies that a character is a vowel.
3. Write a function that returns 0 if it receives a character that isn't a letter of the alphabet, 1 if it is an uppercase letter, and 2 if it is a lowercase letter. Keep the function as portable as possible.
4. For your compiler, determine what flags must be set to ignore variable case, allow for byte alignment, and guarantee ANSI compatibility.
5. Is the following code portable?

```
void list_a_file( char *file_name )
{
    system("TYPE " file_name );
}
```

6. Is the following code portable?

```
int to_upper( int x )
{
    if( x >= `a' && x <= `z' )
    {
         toupper( x );
    }
    return( x );
}
```