Efficiency and Porting

WEEK
1

When writing programs, it's easy to use everything the compiler provides. With a language such as C, it's easy to write code that can be used on different computers including PC compatibles, Macintoshes, mini-computers, and mainframes. Because of C's flexibility, you need to know in advance the direction in which you are headed. Today you learn:

- ☐ What is meant by efficiency and porting.
- ☐ The difference between efficiency and maintainability.
- ☐ What types of applications are most likely to be nonportable.
- ☐ How to ensure portability with the ANSI standard.

What Are Efficiency and Portability?

If you were to take a course in computer programming, two topics would inevitably be mentioned: efficiency and portability. These topics can be especially important when programming in C.

Efficiency

When efficiency is mentioned, it is typically in reference to writing the least amount of code to gain the most functionality. An efficient program is one that only uses system resources when needed. In addition, it is coded to use as little redundant or unnecessary code as possible.

Compilers, Computers, and Efficiency

In days past, if you wanted the most efficient code, you wrote it in assembler. By writing at such a low level, you could weed out any non-essential commands. Today's compilers and computers are much better equipped than those of yesteryear. If you talk to programmers of the early 1980s, you'll hear tales of writing small programs that use minimal resources. Today's computers allow a larger amount of leeway.

Efficiency Versus Maintainability

Although writing efficient code is important, the greatest cost of creating programs is the amount of time spent programming code. It is best to design the code to

minimize the amount of time that must be spent on it. Because writing efficient code can often be time-intensive, the quicker it is written, the less efficient the code may be.

Programming time can be broken down to two phases. The first is the initial time spent developing and coding a program. The amount time for this phase depends on the approach taken. (In Day 12, you'll be shown different ways to approach the initial development of a program.) The second phase is maintaining the program. This maintenance can involve either fixing problems or adding enhancements. Such maintenance can quickly add up to much more time than what was initially spent developing the program.

Coding a program for efficiency can also lead to code that is harder to maintain. When efficiency is the most important factor, coding tricks may be employed. These tricks may make sense to the person who programmed them; however, they seldom are clear to the person who maintains the code. For this reason, the efficiency gained by the coding tricks may be lost in the time and effort spent maintaining them.

Expert Tip: Coding for maintainability is usually more important than coding for the ultimate efficiency.

White Space

Many super-techie programmers try to write compact code. These programmers, along with many others, believe that compact code is more efficient. The next two listings each contain the same code; however, one contains more spacing.

Listing 5.1. Code with spacing.

```
1:   /* Program: list0501.c
2:    * Author:  Bradley L. Jones
3:    * Purpose: This program and list0502.c demonstrate the
4:    *          differences made by white space in a listing.
5:    *=======================================================*/
6:
7:   int main(void)
8:   {
9:     int ctr, ctr2;
10:
11:    printf( "\n\nA program with useless output" );
12:
```

continues

Listing 5.1. continued

```
13:    printf( "\n" );
14:
15:    for( ctr = 0; ctr < 26; ctr++ )
16:    {
17:       printf( "\n" );
18:
19:       for( ctr2 = 0; ctr2 <= ctr; ctr2++ )
20:       {
21:          printf( "%c", ('A' + ctr ));
22:       }
23:    }
24:
25:    return;
26: }
```

Output

A program with useless output

```
A
BB
CCC
DDDD
EEEEE
FFFFFF
GGGGGGG
HHHHHHHH
IIIIIIIII
JJJJJJJJJJ
KKKKKKKKKKK
LLLLLLLLLLLL
MMMMMMMMMMMM
NNNNNNNNNNNNN
OOOOOOOOOOOOOO
PPPPPPPPPPPPPPP
QQQQQQQQQQQQQQQQ
RRRRRRRRRRRRRRRRR
SSSSSSSSSSSSSSSSS
TTTTTTTTTTTTTTTTTTT
UUUUUUUUUUUUUUUUUUUU
VVVVVVVVVVVVVVVVVVVVV
WWWWWWWWWWWWWWWWWWWWWW
XXXXXXXXXXXXXXXXXXXXXXX
YYYYYYYYYYYYYYYYYYYYYYYY
ZZZZZZZZZZZZZZZZZZZZZZZZZ
```

Type

Listing 5.2. Compact code.

```
1:  /* Program: list0502.c
2:   * Author:  Bradley L. Jones
```

```
3:    * Purpose: This program and list0501.c demonstrate the
4:    *          differences made by white space in a listing.
5:    *=========================================================*/
6:   int main(void){
7:   int ctr,ctr2;
8:   printf("\n\nA program with useless output");
9:   printf("\n");
10:  for(ctr=0;ctr<26;ctr++){
11:  printf("\n");
12:  for(ctr2=0;ctr2<=ctr;ctr2++){printf("%c",('A'+ctr));}}
13:  return;}
```

Output

```
A program with useless output

A
BB
CCC
DDDD
EEEEE
FFFFFF
GGGGGGG
HHHHHHHH
IIIIIIIII
JJJJJJJJJJ
KKKKKKKKKKK
LLLLLLLLLLLL
MMMMMMMMMMMM
NNNNNNNNNNNNN
OOOOOOOOOOOOOO
PPPPPPPPPPPPPPP
QQQQQQQQQQQQQQQQ
RRRRRRRRRRRRRRRRR
SSSSSSSSSSSSSSSSS
TTTTTTTTTTTTTTTTTTT
UUUUUUUUUUUUUUUUUUUU
VVVVVVVVVVVVVVVVVVVVV
WWWWWWWWWWWWWWWWWWWWWW
XXXXXXXXXXXXXXXXXXXXXXX
YYYYYYYYYYYYYYYYYYYYYYYY
ZZZZZZZZZZZZZZZZZZZZZZZZZZ
```

Analysis These programs simply print each of the characters of the alphabet. Each consecutive letter is printed one more time than the previous. The letter A is printed once, the letter B twice, up to the letter Z, which is printed 26 times. The output demonstrates that by changing the white space, you don't change the way the program operates.

If you look at the size of the two object files and the two executable files created, you'll find that they are the same. Table 5.1 contains all the associated files and their sizes. The source files are different sizes because of the white space, which is removed by the compiler. However, white space doesn't make the code less efficient. In fact, while white space makes the source file a little bigger, it doesn't detract from a program's efficiency.

Table 5.1. The file sizes of the two listings.

Filename	Size
LIST0501.C	527
LIST0502.C	431
LIST0501.EXE	6536
LIST0502.EXE	6536
LIST0501.OBJ	383
LIST0502.OBJ	383

If you look at Listings 5.1 and 5.2 again, you'll see that Listing 5.1 is much easier to read and understand. The code in Listing 5.1 is also easier—and quicker—for others to maintain.

Review Tip: Use white space to make your programs more readable.

DO	DON'T
DO use white space to make your programs more readable.	
DO consider maintainability over efficiency when coding your programs.	

Portability

One of the major reasons people choose C as their programming language is its portability. C is one of the most portable languages. A program written in a portable

language can be moved from one compiler to another, or from one computer system to another. When moved, the program can be recompiled without any coding modifications. These two areas—hardware portability and compiler portability—are what characterize a portable language. C programs can be written to be portable across both. A C program is truly portable if it can be recompiled on any type of machine with any C compiler.

The ANSI Standard

Portability doesn't happen by accident. It occurs by adhering to a set of standards adhered to by the programmer and your compiler. If you use a compiler that doesn't adhere to the portability standards, you'll be unable to write usable portable code. For this reason, it is wise to choose a compiler that follows the standards for C programming set by the American National Standards Institute (ANSI). The ANSI committee also sets standards for many areas including other programming languages. The ANSI standards are predominantly accepted and used by programmers and compilers.

Standards aren't always good. Too many standards can limit your ability to create effective programs. Because C is a powerful language, it could be detrimental to implement too many standards. The ANSI standards leave a lot of undefined areas to prevent this power limitation. The downside to undefined areas is each compiler can create its own implementations. There are several such areas that will be detailed later today.

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 5.2.

Table 5.2. 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. Examples of compiler-specific keywords are `near` and `huge`. Although several compilers might use the same compiler-specific keywords, there is no guarantee that they 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 `x` is different than a variable named `X`. Listing 5.3 illustrates this.

Type

Listing 5.3. Case sensitivity.

```
1:  /* Program: list0505.c
2:   * Author:  Bradley L. Jones
3:   * Purpose: This program demonstrates case sensitivity
4:   *=========================================================*/
5:
6:  int main(void)
7:  {
8:    int   var1 = 1,
9:          var2 = 2;
10:   char  VAR1 = 'A',
11:         VAR2 = 'B';
12:   float Var1 = 3.3,
13:         Var2 = 4.4;
14:   int   xyz  = 100,
15:         XYZ  = 500;
16:
17:   printf( "\n\nPrint the values of the variables...\n" );
18:
19:   printf( "\nThe integer values:   var1 = %d, var2 = %d",
20:            var1, var2 );
21:   printf( "\nThe character values: VAR1 = %c, VAR2 = %c",
22:            VAR1, VAR2 );
23:   printf( "\nThe float values:     Var1 = %f, Var2 = %f",
24:            Var1, Var2 );
25:   printf( "\nThe other integers:   xyz = %d, XYZ = %d",
26:            xyz, XYZ );
27:
28:   printf( "\n\nDone printing the values!" );
29:
30:   return;
31: }
```

Output

```
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 8 and 9, `var1` and `var2` are defined as integer values. In lines 10 and 11, the same variable names are used with different cases. This time `VAR1` and `VAR2` are in all uppercase. In lines 12 and 13, a third set of declarations is made with the same names, but a 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 later be printed. The printing for these three sets of declarations occurs in lines 19 to 24. As you can see, the values placed in the variables are retained, and each is printed.

Lines 14 and 15 declare two variables of the same type—integers—and the same names. 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 in lines 25 and 26.

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 may 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 only 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 may be able to differentiate between variables with only case differences, but the linker may not. Case sensitivity can affect the workings of both the compiler and the linker.

Most compilers and linkers enable you to 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:

```
list05xx.c:
Error list05xx.c 15: Multiple declaration for 'var1' in function main
*** 1 errors in Compile ***
```

Of course, `var1` would be whatever variable you are using.

Portable Characters

Characters within the computer are represented as numbers. On an IBM PC or compatible, the letter *A* is represented by the number 6, and the letter *a* is represented by the number 97. These numbers come from an ASCII table.

If you're writing portable programs, you cannot assume that the ASCII table is the character translation table being used. A different table may be used on a different computer system. On a mainframe, character 65 may not be *A*.

Warning: You must be careful when using character numerics. Character numerics may not be portable.

There are two general rules about how a character set is to be defined. The first restriction 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 were 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 are not 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 compatability. 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 by either setting a switch within the IDE (Integrated Programming 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, you would enter the following on the command line:

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

If you are compiling with a Microsoft compiler, you would enter:

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

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 may not receive the required prototype warnings.

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, aren't ANSI standard, or they are 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 compiler-specific features, you may eliminate the portability of your program. Later today, you'll see a way around this limitation.

DO	DON'T
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 may not be consistent across compilers. There are only a few rules that are defined within the ANSI standard in regards to the numeric values that can be stored in each variable type. On Day 2, Table 2.1 presented the values that are typically stored in IBM-compatible PCs. These values, however, aren't guaranteed.

The following rules can be observed about variable types:

- A character (`char`) is the smallest data type. A character variable (type `char`) will be 1 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 5.4 is commonly used to print the size of the variables on the machine that the program is compiled on.

Type

Listing 5.4. The size of the data types.

```
1:  /* Program: list0506.c
2:   * Author:  Bradley L. Jones
3:   * Purpose: This program prints the sizes of the variable
4:   *          types of the machine the program is compiled on.
5:   *=========================================================*/
6:
7:  int main(void)
8:  {
9:    printf( "\nVariable Type Sizes" );
10:   printf( "\n=========================" );
11:   printf( "\nchar              %d", sizeof(char) );
12:   printf( "\nshort             %d", sizeof(short) );
13:   printf( "\nint               %d", sizeof(int) );
14:   printf( "\nfloat             %d", sizeof(float) );
15:   printf( "\ndouble            %d", sizeof(double) );
16:
17:   printf( "\n\nunsigned char    %d", sizeof(unsigned char) );
18:   printf( "\nunsigned short   %d", sizeof(unsigned short) );
19:   printf( "\nunsigned int     %d", sizeof(unsigned int) );
20:
21:   return;
22: }
```

Output

```
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 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 may be different. For example, a 32-bit compiler on a 32-bit machine may yield 4 bytes for the size of an integer rather than 2.

Maximum and Minimum Values

If different machines have variable types that are different sizes, how can 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. Table 5.3 shows the different values that can be stored 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 5.3 shows both integral variable and floating decimal values.

Table 5.3. Possible values based on byte size.

Number of Bytes	Unsigned Maximum	Signed Minimum	Signed 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 x 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 seen 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 have to be careful about what number 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, then assigning 100,000 would be okay.

Warning: You aren't guaranteed that the values in Table 5.3 are the same for every compiler. Each compiler may choose a slightly different number. This is especially true with the floating-point numbers which may have different levels of precision. Tables 5.4 and 5.5 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 5.4 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 5.4. 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.
SHRT_MIN	Short variable's minimum value.
SHRT_MAX	Short variable's maximum value.
USHRT_MAX	Unsigned short variable's maximum value.

Constant	Value
INT_MIN	Integer variable's minimum value.
INT_MAX	Integer variable's maximum value.
UINT_MAX	Unsigned integer 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 5.5. 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).

continues

Table 5.5. continued

Constant	Value
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 5.4 and 5.5 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 will ensure that the listing will be portable. Listing 5.5 prints the values stored in the ANSI-defined constants and Listing 5.6 demonstrates the use of some of these constants. The output may be slightly different depending on the compiler used.

Listing 5.5. The values stored in the ANSI-defined constants.

```
1:   /* Program:  list0507.c
2:    * Author:   Bradley L. Jones
3:    * Purpose:  Display of defined constants.
4:    *=========================================================*/
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 ", DBL_MIN );
34:
35:      return(0);
36:  }
```

Output

```
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
```

Analysis This listing is straightforward. The program consists of `printf()` function calls. Each function call prints a different defined constant. You'll notice the conversion character used (that is, `%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 if these values had been defined. This program should make determining the constant values easier.

Type **Listing 5.6. Using the ANSI-defined constants.**

```
1:  /* Program: list0508.c
2:   * Author:  Anon E. Mouse
3:   *
4:   * Purpose: To use maximum and minimum constants.
5:   *
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", ch, ch );
35:
36:     return;
37: }
```

Output

```
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

This listing shows the UCHAR_MAX constant in action. The first new item you should notice is the includes in lines 10 and 11. As stated earlier, these two include files contain the defined constants. If you are questioning the need for FLOAT.H to be included in line 10, then you're doing well. Because none of the decimal point constants are being used, the FLOAT.H header file is not 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 if 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 (and that is an unsigned character), then 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 may find it's easier to switch to the character variable than to continue with the integer. This can help to 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

There are several instances when you'll want to know information about a variable. For instance, you may want to know if 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 5.7, which demonstrates one way of determining if a value stored in a character is a letter of the alphabet.

Type

Listing 5.7. Is the character an alphabetic letter?

```
1:  /* Program: list0509.c
2:   * Author:  Bradley L. Jones
3:   * Purpose: This program may not be portable due to the
4:   *          way it uses character values.
5:   *=========================================================*/
6:
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 an 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!");
37:   return;
38: }
```

Output

```
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) ==> g

g is a letter of the alphabet!
g is an 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 an lowercase letter!

Thank you for playing!
```

Analysis

This program checks to see if a letter is between the uppercase letter A and the uppercase letter Z. In addition, it checks to see if it is between the lowercase a and the lowercase z. If x is between one of these two ranges, then you would think you could assume that the letter is alphabetic. This is a bad assumption! There is not a standard for the order in which characters are stored. If you are 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 5.6 with what it checks for. These functions will return a zero if the given character doesn't meet its check; otherwise it will return a value other than zero.

Table 5.6. The character classification functions.

Function	Purpose
`isalnum()`	Checks to see if the character is alphanumeric.
`isalpha()`	Checks to see if the character is alphabetic.
`iscntrl()`	Checks to see if the character is a control character.

continues

Table 5.6. continued

Function	Purpose
`isdigit()`	Checks to see if the character is a decimal digit.
`isgraph()`	Checks to see if the character is printable (space is an exception).
`islower()`	Checks to see if the character is lowercase.
`isprint()`	Checks to see if the character is printable.
`ispunct()`	Checks to see if the character is a punctuation character.
`isspace()`	Checks to see if the character is a whitespace character.
`isupper()`	Checks to see if the character is uppercase.
`isxdigit()`	Checks to see if 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 if the value of a character variable is equal to `'A'`, but you wouldn't want to check to see if the value of a character is greater than `'A'`.

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

Listing 5.8 is a rewrite of Listing 5.7. Instead of using range checks, the appropriate character classification values are used. Listing 5.8 is a much more portable program.

Listing 5.8. Using character classification functions.

```
1:   /* Program: list0510.c
2:    * Author:  Bradley L. Jones
3:    * Purpose: This program is an alternative approach to
4:    *          the same task accomplished in Listing 5.9.
5:    *          This program has a higher degree of portability!
6:    *=========================================================*/
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 an 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!");
39:    return;
40: }
```

Output

```
Enter a character (Q to quit) ==> z

z is a letter of the alphabet!
z is an 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 Listing 5.9—assuming that 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 as much. Line 23 then checks to see if the character is uppercase with the `isupper()` function. If `x` is an uppercase character, then a message is printed in line 25, otherwise the message in line 29 is printed. If the letter was not an alphabet letter, then a message is printed in line 35. 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 non-equality checks aren't portable. "Not equal to" and "equal to" are both equality checks.

DO	DON'T
DON'T use numeric values when determining maximums for variables. Use the defined constants if you are writing a portable program.	
DON'T assume that the letter A comes before the letter B if you are 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 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, the `if` statement may not be portable. The following is an update function with the `if` statement updated to the portable functions presented in the previous section:

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

This second example is better than the previous listing in terms of portability; however, it still isn't completely portable. This function makes the assumption 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 is not necessarily true on every system. Particularly, it is untrue on non-ASCII character systems.

There are two ANSI standard functions that 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 rewritten would look as follows:

```
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. Byte alignment and the order in which members are stored are two areas of incompatibility that can occur when working with these constructs.

Byte alignment, which was discussed on Day 2, is an important factor in the portability of a structure. A program can't assume that the byte alignment will be the same or that it will be on or off. The members could be aligned on every 2 bytes, 4 bytes, or 8 bytes. You cannot assume to know.

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 are reading or writing structures to a file, the file probably won't be portable. This means you only need to concentrate

on making the program portable. The program would then need 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 will be read.

When you create a structure, you may assume that the members will be stored in the order in which they are listed. In fact, many of the figures of structures that were presented on Day 2 made this assumption. This assumption is not guaranteed. There isn't a standard that states that a certain order must be followed. Because of this, you can't make assumptions about the order of information within a structure. Listing 5.9 shows an incomplete program that makes such an assumption.

Listing 5.9. A program that may not be portable.

```
1:   /* Program: list0511.c
2:    * Author:  Bradley L. Jones
3:    * NOTE:    THIS IS AN INCOMPLETE PROGRAM
4:    * Purpose: Demonstrates potentially non-portable code!
5:    *=========================================================*/
6:
7:   struct date
8:   {
9:     int  year;   /* year in YYYY format */
10:    char month;  /* month in MM format  */
11:    char day;    /* day in DD format    */
12:  }
13:
14:  int main(void)
15:  {
16:    struct date today, birthday;
17:
18:    set_up_values(&today, &birthday);
19:
20:    if ( *(long *)&today == * (long *)&birthday )
21:    {
22:        do_birthday_function();
23:    }
24:  }
```

This program doesn't actually run. This program would accept two dates, one for today's date and one for a birthdate. It then compares the values of the two dates in line 20 to see if they are equal. If they are, the program completes the `do_birthday_function()`.

As you probably guessed, line 20 may not be portable because it takes the value stored in the `today` structure and compares it to the value stored in the `birthday` structure This is a tricky way to compare the two structures. Because the structure members are stored in the order in which they should be compared, it's easier to compare the complete values of the entire structure instead of manipulating each member. This makes the assumption that a date would be stored in year, month, day order as declared in the `date` structure. Two dates—such as 1991, 12, 25 and 1993, 08, 01—could be compared as 19911225 and 19930801.

Because the compiler isn't required to store the `date` structure in `year`, `month`, `day` order, this program wouldn't be guaranteed as portable. A second portability issue also exists. This program assumes that the total size of the structure (2 characters and an integer) is equal to the size of a `long`. This is not guaranteed.

Preprocessor Directives

There are several preprocessor directives that have been defined in the ANSI standards. You use two of these all the time. They are `#include` and `#define`. There are several other preprocessor directives that are in the ANSI standards. The additional preprocessor directives that are available under the ANSI guidelines are listed in Table 5.7.

Table 5.7. ANSI standard preprocessor directives.

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

Later today, you will see an example of using some of the preprocessor directives to create programs with compiler-specific code and still retain portability.

Using Predefined Constants

Every compiler comes with predefined constants. A majority of these are typically compiler specific. This means that there is a good chance that they won't be portable from one compiler to the next. There are, however, several predefined constants that are defined in the ANSI standards. Some of these constants are:

`__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 by 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".

Note: The following listing needs to be compiled with the ANSI compatibility flag on. This is usually set by passing an additional parameter when compiling. For example, with Borland's Turbo C, you would enter the following:

```
TCC -A LIST0512.C
```

The `-A` tells the compiler to compile as an ANSI-compatible source file. If you don't compile with the ANSI compatibility flag, you'll get an error similar to the following:

```
list0512.c:
Error list0512.c 24: Undefined symbol '__STDC__' in function
main
*** 1 errors in Compile ***
```

The best way to understand the predefined constants is to see them in action. Several of the predefined ANSI constants are presented in Listing 5.10.

Listing 5.10. The predefined ANSI constants in action.

```
1:  /* Program: list0510.c
2:   * Author:  Bradley L. Jones
3:   * Purpose: This program demonstrates the values printed
4:   *          by some of the pre-defined identifiers.
5:   * Note:    In order for this to compile, the ANSI standard
6:   *          compiler switch must be set.
7:   *=========================================================*/
8:
9:  #include <string.h>
10:
11: int main(void)
12: {
13:   printf("\n\nCurrently at line %d", __LINE__ );
14:
15:   printf("\n\nThe value of __DATE__ is: ");
16:   printf(__DATE__);
17:
18:   printf("\n\nThe value of __TIME__ is: ");
19:   printf(__TIME__);
20:
21:   printf("\n\nThe value of __LINE__ is: %d", __LINE__ );
22:
23:   printf("\n\nThe value of __STDC__ is 1 if ANSI compatibility
            is on");
24:   (__STDC__ == 1) ? printf("\nANSI on") : printf("\nANSI off");
25:
26:   printf("\n\nThe value of __FILE__ is: ");
27:   printf(__FILE__);
28:
29:   return;
30: }
```

```
Currently at line 13

The value of __DATE__ is: Nov 27 1993

The value of __TIME__ is: 10:14:03

The value of __LINE__ is: 21

The value of __STDC__ is 1 if ANSI compatibility is on
ANSI on

The value of __FILE__ is: list0512.c
```

Listing 5.10 demonstrates the ANSI predefined constants by simply printing their values using `printf()` function calls. Line 13 prints the value of `__LINE__`. As you can see in the output, this is the value of 13. Line 16 prints the value of `__DATE__`. To show that this is a simple string, the `__DATE__` constant is passed as the only parameter to `printf()`. Lines 15 and 16 could be printed as:

```
printf("\n\nThe value of __DATE__ is: %s", __DATE__);
```

Line 19 prints the `__TIME__` constant. Like the date, this is printed as a separate string, but could have been combined with the previous `printf()` statement. Line 24 determines the value of `__STDC__`. This is not a good line because if `__STDC__` is defined, ANSI is on. If it isn't defined, the compiler will give an error and the program won't compile.

Line 27 wraps up the listing by printing the filename it had when it was compiled. If you rename the executable, the value of `__FILE__` will still be the original filename.

Using Non-ANSI Features in Portable Programs

A program can use non-ANSI-defined constants and other commands and still be portable. This is accomplished by ensuring the constants are used only if compiled with a compiler that supports the features used. Most compilers provide defined constants that can be used to identify themselves. By setting up areas of the code that are supportive for each of the compilers, you can create a portable program. Listing 5.11 demonstrates how this can be done.

Type

Listing 5.11. A portable program with compiler specifics.

```
1:  /* Program: list0511.c
2:   * Author:  Bradley L. Jones
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:
9:  #ifdef _WINDOWS
10:
11: #define STRING "DOING A WINDOWS PROGRAM!"
12:
13: #else
14:
15: #define STRING "NOT DOING A WINDOWS PROGRAM"
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", _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", __TURBOC__ );
35:
36:  #endif
37:
38:  #ifdef __BORLANDC__
39:
40:     printf( "\n\nUsing a Borland compiler!" );
41:
42:  #endif
43:
44:     return(0);
45:  }
```

Output

```
NOT DOING A WINDOWS PROGRAM

Using the Turbo C compiler!
   Your compiler version is 300
```


```
NOT DOING A WINDOWS PROGRAM

Using a Borland compiler!
```


```
NOT DOING A WINDOWS PROGRAM

Using a Microsoft compiler!
   Your compiler version is >>
```

Analysis

This listing 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 if 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 of whether `_WINDOWS` has been defined is made. An appropriate message is applied to the constant `STRING`. Line 22 then prints this string, which states whether this listing has been compiled as a Windows program or not.

Line 24 checks to see if `_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 version number of the compiler. 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 similar manners. They check to see if 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 what compiler is being used by checking the defined constants. The object of the program is the same regardless of which compiler is used—print a message stating which compiler is being used. If you are 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 since these can be used in creating portable programs. Appendix E contains the ANSI header files along with a list of their functions.

Summary

Today, you were exposed to a great deal of material. This information centered around efficiency and portability. Efficiency needs to be weighed against maintainability. It's better to write code that can be easily maintained even if it operates a few nano-seconds slower. C is one of the most portable languages—if not the most portable language. 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. There are several areas to consider when writing portable code. These areas include variable case, which character set to use, using portable numerics, ensuring variable sizes, comparing characters, using structures and unions, and using preprocessor directives and preprocessor constants. The day 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 does not define any real standards for programming graphics. With graphics programming being more machine dependent than other programming areas, 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 that you write will only be used by you on the system you are using. 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 Do comments make a program less efficient?

A Comments are stripped out by the compiler. Because of this, they don't hurt a program. If anything, comments add to the maintainability of a program. You should always use comments where they can make the code clearer.

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.

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 firstname,
    FIRSTNAME,
    FirstName,
    Firstname;
```

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? For 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. Rewrite a listing in this chapter and remove all the unneeded spaces. Does the listing still work? Is it smaller than the original listing?
3. Write a function that verifies that a character is a vowel.
4. 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.

5. **ON YOUR OWN:** Understand your compiler. Determine what flags must be set to ignore variable case, allow for byte alignment, and guarantee ANSI compatibility.

6. Is the following code portable?

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

7. Is the following code portable?

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