Chapter 4: Assignment and Logical Compares

Throughout this chapter, references are given to various ranges of variables. This refers to the range of values that can be stored in any given variable. Your compiler may use a different range for some of the variables since the ANSI standard does not define specific limits for all data types. Consult the documentation for your compiler for the exact range for each of the variable types.

INTEGER ASSIGNMENT STATEMENTS

Load the file named intasign.c and display it for an example of assignment statements. Three variables are defined for use in the program and the remainder of the program is merely a series of illustrations of various kinds of assignment statements. All three variables are defined on one line and have unknown values stored in them initially.

The first two lines of the assignment statements, lines 8 and 9, assign numerical values to the variables named **a** and **b**, and the next five lines illustrate the five basic arithmetic functions and how to use them. The fifth is the modulo operator and gives the remainder if the two variables were divided. It can only be applied to **int** or **char** type variables. The **char** type variable will be defined in the description of the next example program. Lines 15 and 16 illustrate how to combine some of the variables in relatively complex math expressions. All of the above examples should require no comment except to say that none of the equations are meant to be particularly useful except as illustrations.

The expressions in lines 17 and 18 are perfectly acceptable as given, but we will see later in this chapter that there is another way to write these for more compact code.

VERY STRANGE LOOKING CODE

This leaves us with the last two lines which may appear to you as being very strange. The C compiler scans the assignment statement from right to left, (which may seem a bit odd since we do not read that way), resulting in a very useful construct, namely the one given here. The compiler finds the value 20, assigns it to **c**, then continues to the left finding that the latest result of a calculation should be assigned to **b**. Thinking that the latest calculation resulted in a 20, it assigns it to **b** also, and continues the leftward scan assigning the value 20 to **a** also. This is a very useful construct when you are initializing a group of variables. The statement in line 21 illustrates that it is possible to actually do some calculations to arrive at the value which will be assigned to all three variables. The values of **a**, **b**, and **c**, prior to the beginning of the statement in line 21 are used to calculate a value, which is then assigned to each of the variables.

As an aid to understanding, line 22 is given which contains parentheses to group the terms together in a meaningful way. Lines 20 and 22 are identical statements since they lead to the same result.

The program has no output, so compiling and executing this program will be very uninteresting. Since you have already learned how to display some integer results using the **printf()** function, it would be to your advantage to add some output statements to this program to see if the various statements do what you think they should do. You can add your own assignment statements also to gain experience with them.

DEFINITIONS FIRST THEN EXECUTABLE STATEMENTS

This would be a good time for a preliminary definition of a rule to be followed in C. The variable definitions are always given before any executable statements in any program block. This is why the variables are defined first in this program and in every C program. If you try to define a new variable after executing some statements, your compiler will issue an error. A program block is any unit of one or more statements surrounded by braces. More will be said about blocks later.

ADDITIONAL DATA TYPES

Loading and editing mortypes.c will illustrate how some additional data types can be used. Once again we have defined a few integer type variables which you should be fairly familiar with by now, but we have added two new types, the **char**, and the **float**.

The **char** type of data is nearly the same as the integer except that it can only be assigned numerical values between -128 and 127 on most implementations of C, since it is stored in only one byte of memory. The **char** type of data is usually used for ASCII data, more commonly known as text. The text you are reading was originally written on a computer with a word processor that stored the words in the computer one character per byte. In contrast, the integer data type is stored in two bytes of computer memory on nearly all microcomputers.

Keep in mind that, even though the **char** type variable was designed to hold a representation of an ASCII character, it can be used very effectively to store a very small value if desired. Much more will be discussed on this topic in chapter 7 when we discuss strings.

DATA TYPE MIXING

It would be profitable at this time to discuss the way C handles the two types **char** and **int**. Most operations in C that are designed to operate with integer type variables will work equally well with character type variables because they are a form of an integer variable. Those operations, when called on to use a **char** type variable, will actually promote the **char** data into integer data before using it. For this reason, it is possible to mix **char** and **int** type variables in nearly any way you desire. The compiler will not get confused, but you might. It is good not to rely on this too much, but to carefully use only the proper types of data where they should be used.

The second new data type is the **float** type of data, commonly called floating point data. This is a data type which usually has a very large range, a large number of significant digits, and a large number of computer words are required to store it. The **float** data type has a decimal point associated with it and has an allowable range of from 3.4E-38 to 3.4E+38 when using most C compilers on microcomputers, and is composed of about 7 significant digits. Four bytes of memory are required to store a single float type variable.

HOW TO USE THE NEW DATA TYPES

The first three lines of the program assign values to all nine of the defined variables so we can manipulate some of the data between the different types.

Since, as mentioned above, a **char** data type is in reality an integer data type, no special considerations need be taken to promote a **char** to an **int**, and a **char** type data field can be assigned to an **int** variable. Most C compilers simply truncate the most significant bits and use the 8 least significant bits. An **int** type variable will translate correctly to a **char** type variable if the value is within the range of -128 to 127.

Line 16 illustrates the simplicity of translating an **int** into a **float**. Simply assign it the new value and the system will do the proper conversion. When converting from **float** to **int** however, there is an added complication. Since there may be a fractional part of the floating point number, the system must decide what to do with it. By definition, it will truncate it and throw away the fractional part.

This program produces no output, and we haven't covered a way to print out **char** and **float** type variables, so you can't really get in to this program and play with the results. The next program will cover these topics for you.

Be sure to compile and run this program after you are sure you understand it completely. Note that, once again, the compiler may issue warnings about type conversions when compiling this program. They can be ignored because of the small values we are using to illustrate the various type conversions.

LOTS OF VARIABLE TYPES

Load the file lottypes.c and display it on your screen. This file contains nearly every standard simple data type available in the programming language C. There are other types, but they are the compound types (ie - arrays and structures) that we will cover in due time.

Observe the file. First we define a simple **int**, followed by a **long int** which has a range of -2147483648 to 2147483647 with most C compilers, and requires four bytes of storage. Next we have a **short int** which has a range that is identical to that for the **int** variable, namely -32768 to 32767 and requires two bytes of storage. The **unsigned** is next and is defined as the same size as the **int** but with no sign. The **unsigned** then will cover a range of 0 to 65535. It should be pointed out that when the **long**, **short**, or **unsigned** is desired, the **int** is optional and is left out by most experienced programmers. We have already covered the **char** and the **float**, which leaves only the **double**.

The **double** is a floating point number but covers a greater range than the **float** and has more significant digits for more precise calculations. It also requires more memory to store a value than the simple float. The **double** in most C compilers covers a range of 1.7E-308 to 1.7E+308, contains 15 significant digits, and uses 8 bytes of memory to store a single value.

Note that other compounding of types can be done such as **long unsigned int**, **unsigned char**, etc. Check your documentation for a complete list of variable types.

Another diversion is in order at this point. Your compiler probably has no provision for floating point math, only double floating point math. It will promote a **float** to a **double** before doing calculations and therefore only one math library will be needed. Of course, this is transparent to you, so you don't need to worry about it. Because of this, you may think that it would be best to simply define every floating point variable as **double**, since they are promoted before use in any calculations, but that may not be a good idea. A **float** variable requires 4 bytes of storage and a **double** requires 8 bytes of storage, so if you have a large volume of floating point data to store, the **double** will obviously require much more memory. If you don't need the additional range or significant digits, you should use the **float** type rather than the **double**.

After defining the data types in the program under consideration, a numerical value is assigned to each of the defined variables in order to demonstrate the means of outputting each to the monitor.

SOME LATE ADDITIONS

As any programming language evolves, additional constructs are added to fill some previously

overlooked need. Two new keywords have been added to C with the release of the ANSI-C standard. They are not illustrated in example programs, but they will be discussed here. The two new keywords are **const** and **volatile** and are used to tell the compiler that variables of these types will need special consideration. A constant is declared with the **const** keyword and declares a value that will never be changed by either the program or the system itself. If you inadvertently try to modify an entity defined as a **const**, the compiler will generate an error. This is an indication to you that something is wrong. Declaring an entity as **const** allows the optimizer to do a better job which could make your program run a little faster. Since constants can never have a value assigned to them in the executable part of the program, they must always be initialized. If **volatile** is used, it declares a value that may be changed by the program but it may also be changed by some outside influence such as a clock update pulse incrementing the stored value. This prevents the optimizer from getting too ambitious and optimizing away something that it thinks will never be changed.

Examples of use in declaring constants of these two types are given as;

```
const int index1 = 2;
const index2 = 6;
const float big_value = 126.4;
volatile const int index3 = 12;
volatile int index4;
```

THE CONVERSION CHARACTERS

Following is a list of some of the conversion characters and the way they are used in the **printf()** statement. A complete list of all of the conversion characters should be included with the documentation for your compiler.

```
d  decimal notation
i  decimal notation (new ANSI standard extension)
o  octal notation
x  hexadecimal notation
u  unsigned notation
c  character notation
s  string notation
f  floating point notation
```

Each of these is used following a percent sign to indicate the type of output conversion desired. The following fields may be added between those two characters.

```
-   left justification in its field
(n)  a number specifying minimum field width
.   to separate n from m
(m)  significant fractional digits for a float
l   to indicate a long
```

These are all used in the examples which are included in the program named lottypes.c, with the exception of the string notation which will be covered later in this tutorial. Lines 31 through 33 illustrate how to set the field width to a desired width, and lines 37 and 38 illustrate how to set the field width under program control. This is not part of the original definition of C, but it is included in the ANSI standard and has become part of the C language. The field width for the float type output in lines 41 through 45 should be self explanatory. Compile and run this program to see what effect the various fields have on the output.

You now have the ability to display any of the data fields in the previous programs and it would be to your advantage to go back and see if you can display some of the fields anyway you desire.

COMBINING THE VARIOUS TYPES

Examine the file named combine.c for examples of combining variables of the various types in a program. Many times it is necessary to multiply an **int** type variable times a **float** type variable and C allows this by giving a strict set of rules it will follow in order to do such combinations.

Five variables of three different types are declared in lines 4 through 6, and three of them are initialized so we have some data to work with. Line 8 gives an example of adding an **int** variable to a **float** variable and assigning the result to a **char** type variable. The cast is used to control the type of addition and is indicated by putting the desired type in front of the variable as shown. This forces each of the two variables to the character type prior to doing the addition. In some cases, when the cast is used, the actual bit patterns must be modified internally in order to do the type coercion. Lines 9 through 11 perform the same operation by using different kinds of type casting to achieve the final result.

Lines 13 through 15 illustrate the use of the cast to multiply two **float** variables. In two of the cases the intermediate results are cast to the **int** type, with the result being cast back to the **float** type. The observant student will notice that these three lines will not necessarily produce the same result.

Be sure to compile and execute this program.

LOGICAL COMPARES

Load and view the file named compares.c for many examples of compare statements in C. We begin by defining and initializing nine variables to use in the following compare statements. This method of variable initialization is new to you and can be used to initialize variables when they are defined.

The first group of compare statements represents the simplest kinds of compares because they simply compare two variables. Either variable could be replaced with a constant and still be a valid compare, but two variables is the general case. The first compare checks to see if the value of **x** is equal to the value of **y** and it uses the double equal sign for the comparison. A single equal sign could be used here but it would have a different meaning as we will see shortly. The second comparison checks to see if the current value of **x** is greater than the current value of **z**.

The third compare introduces the not operator, the exclamation, which can be used to invert the result of any logical compare. The fourth checks for the value of **b** less than or equal to the value of **c**, and the last checks for the value of **r** not equal to the value of **s**. As we learned in the last chapter, if the result of the compare is true, the statement following the **if** clause will be executed and the results are given in the comments.

Note that "less than" and "greater than or equal to" are also available, but are not illustrated here.

It would be well to mention the different format used for the **if** statement in this example program. A carriage return is not required as a statement separator and by putting the conditional clause on the same line as the **if**, it adds to the readability of the overall program in this case.

MORE COMPARES

The compares in the second group are a bit more involved. Starting with the first compare, we find a rather strange looking set of conditions in the parentheses. To understand this we must understand just what a true or false is in the C language. A false is defined as a value of zero, and true is defined as any non-zero value. Any integer or character type of variable can be used for the result of a true/false test, or the result can be an implied integer or character.

Look at the first compare of the second group of compare statements. The conditional expression `"r != s"` will evaluate as a true since the value of **r** was set to 0.0 in line 13, so the result of the compare will be a non-zero value. With most C compilers, it would always be set to a 1, but you could get in trouble if you wrote a program that depended on it being 1 in all cases because the compiler writer is permitted to use any non-zero value. Good programming practice would be to not use the resulting 1 in any calculations. Even though the two variables that are compared are **float** variables, the logical result will be of type **int**. There is no explicit variable to which it will be assigned so the result of the compare is an implied **int**. Finally, the resulting number, probably 1 in this case, is assigned to the integer variable **x**. If double equal signs were used, the phantom value, namely 1, would be compared to the value of **x**, but since the single equal sign is used, the value 1 is simply assigned to the variable named **x**, as though the statement were not in parentheses. Finally, since the result of the assignment in the parentheses was non-zero, the entire expression is evaluated as true, and **z** is assigned the value of 1000. Thus we accomplished two things in this statement, we assigned **x** a new value, probably 1, and we assigned **z** the value of 1000. We covered a lot in this statement so you may wish to review it before going on. The important things to remember are the values that define true and false, and the fact that several things can be assigned in a conditional statement. The value assigned to the variable **x** was probably a 1, but remember that the only requirement is that it is nonzero.

The example in line 20 should help clear up some of the above in your mind. In this example, **x** is assigned the value of **y**, and since the result is 11, the condition is non-zero, which is true, and the variable **z** is assigned 222.

The third example of the second group in line 21, compares the value of **x** to zero. If the result is true, meaning that if **x** is not zero, then **z** is assigned the value of 333, which it will be. The last example in this group illustrates the same concept, since the result will be true if **x** is non-zero. The compare to zero in line 21 is not actually needed and the result of the compare is true. The third and fourth examples of this group are therefore identical.

ADDITIONAL COMPARE CONCEPTS

The third group of compares will introduce some additional concepts, namely the logical "and" and the logical "or" operators. We assign the value of 77 to the three integer variables simply to get started again with some defined values. The first compare of the third group contains the new control &&, which is the logical "and" which results in a true if both sides of the and are true. The entire statement reads, if **x** equals **y** and if **x** equals 77 then the result is true. Since this is true, the variable **z** is set equal to 33.

The next compare in this group introduces the || operator which is the logical "or" operator which results in a true if either side of the "or" is true. The statement reads, if **x** is greater than **y** or if **z** is greater than 12 then the result is true. Since **z** is greater than 12, it doesn't matter if **x** is greater than **y** or not, because only one of the two conditions must be true for the result to be true. The result is true, so therefore **z** will be assigned the value of 22.

LOGICAL EVALUATION (SHORT CIRCUIT)

When a compound expression is evaluated, the evaluation proceeds from left to right and as soon as the result of the outcome is assured, evaluation stops. Therefore, in the case of an "and" evaluation, when one of the terms evaluates to false, evaluation is discontinued because additional true terms cannot make the result ever become true. In the case of an "or" evaluation, if any of the terms is found to be true, evaluation stops because it will be impossible for additional terms to cause the result to be false. In the case of additionally nested terms, the above rules will be applied to each of the nested levels. This is called short-circuit evaluation since the remaining terms are not evaluated.

PRECEDENCE OF OPERATORS

The question will come up concerning the precedence of operators. Which operators are evaluated first and which last ? There are many rules about this topic, but I would suggest that you don't worry about it at this point. Instead, use lots of parentheses to group variables, constants, and operators in a way meaningful to you. Parentheses always have the highest precedence and will remove any question of which operations will be done first in any particular statement.

Going on to the next example in group three in line 29, we find three simple variables used in the conditional part of the compare. Since all three are non-zero, all three are true, and therefore the "and" of the three variables is true, leading to the result being true, and **z** is assigned the value of 11. Note that since the variables, **r**, **s**, and **t** are **float** type variables, they could not be used this way, but they could each be compared to zero and the same type of expression could be used as that used in line 29.

Continuing on to line 30 we find three assignment statements in the compare part of the if statement. If you understood the above discussion, you should have no difficulty understanding that the three variables are assigned their respective new values, and the result of all three are non-zero, leading to a resulting value of true.

THIS IS A TRICK, BE CAREFUL

The last example of the third group contains a bit of a trick, but since we have covered it above, it is nothing new to you. Notice that the first part of the compare evaluates to false. The remaining parts of the compare are not evaluated, because it is a logical "and" so it will definitely be resolved as a false because the first term is false. If the program was dependent on the value of **y** being set to 3 in the next part of the compare, it will fail because evaluation will cease following the false found in the first term. Likewise, the variable named **z** will not be set to 4, and the variable **r** will not be changed. This is because C uses short circuit evaluation as discussed earlier.

POTENTIAL PROBLEM AREAS

The last group of compares illustrate three possibilities for getting into a bit of trouble. All three have the common result that the variable **z** will not get set to the desired value, but for different reasons. In line 37, the compare evaluates as true, but the semicolon following the second parentheses terminates the if clause, and the assignment statement involving z is always executed as the next statement. The **if** therefore has no effect because of the misplaced semicolon. This is actually a null statement and is legal in C.

The statement in line 38 is much more straightforward because the variable **x** will always be equal to itself, therefore the inequality will never be true, and the entire statement will never do a thing, but is wasted effort. The statement in line 39 will always assign 0 to **x** and the compare will therefore always be false, never executing the conditional part of the if statement.

The conditional statement is extremely important and must be thoroughly understood to write efficient C programs. If any part of this discussion is unclear in your mind, restudy it until you are confident that you understand it thoroughly before proceeding onward. Compile and run this program. Add some printout to see the results of some of the operations.

THE CRYPTIC PART OF C

There are three constructs used in C that make no sense at all when first encountered because they are not intuitive, but they greatly increase the efficiency of the compiled code and are used

extensively by experienced C programmers. You should therefore be exposed to them and learn to use them because they will appear in most, if not all, of the programs you see in the publications. Load and examine the file named cryptic.c for examples of the three new constructs.

In this program, some variables are defined and initialized in the same statements for use later. The statement in line 8 simply adds 1 to the value of **x**, and should come as no surprise to you. The next two statements also add one to the value of **x**, but it is not intuitive that this is what happens. It is simply by definition that this is true. Therefore, by definition of the C language, a double plus sign either before or after a variable increments that variable by 1. Additionally, if the plus signs are before the variable, the variable is incremented before it is used, and if the plus signs are after the variable, the variable is used, then incremented. In line 11, the value of **y** is assigned to the variable **z**, then **y** is incremented because the plus signs are after the variable **y**. In the last statement of the incrementing group of example statements, line 12, the value of **y** is incremented then its value is assigned to the variable **z**. To use the proper terminology, line 9 uses the postincrement operator and line 10 uses the preincrement operator.

The next group of statements illustrate decrementing a variable by one. The definition works exactly the same way for decrementing as it does for incrementing. If the minus signs are before the variable, the variable is decremented, then used, and if the minus signs are after the variable, the variable is used, then decremented. The proper terminology is the postdecrement operator and the predecrement operator.

THE CRYPTIC ARITHMETIC OPERATOR

Another useful but cryptic operator is the arithmetic operator. This operator is used to modify any variable by some constant value. The statement in line 23 adds 12 to the value of the variable **a**. The statement in line 24 does the same, but once again, it is not intuitive that they are the same. Any of the four basic functions of arithmetic, +, -, *, or /, can be handled in this way, by putting the function desired in front of the equal sign and eliminating the second reference to the variable name. It should be noted that the expression on the right side of the arithmetic operator can be any valid expression, the examples are kept simple for your introduction to this new operator.

Just like the incrementing and decrementing operators, the arithmetic operator is used extensively by experienced C programmers and it would pay you well to understand it thoroughly.

THE CONDITIONAL EXPRESSION

The conditional expression is just as cryptic as the last two, but once again it is very useful so it would pay you to understand it. It consists of three expressions within parentheses separated by a question mark and a colon. The expression prior to the question mark is evaluated to determine if it is true or false. If it is true, the expression between the question mark and the colon is evaluated, and if the compare expression is not true, the expression following the colon is evaluated. The result of the evaluation is used for the assignment as illustrated in line 30. The final result is identical to that of an **if** statement with an **else** clause. This is illustrated by the example in lines 32 through 35 of this group of statements. The conditional expression has the added advantage of more compact code that will compile to fewer machine instructions in the final program.

Lines 37 and 38 of this example program are given to illustrate a very compact way to assign the greater of the two variables **a** or **b** to **c**, and to assign the lessor of the same two variables to **c**. Notice how efficient the code is in these two examples.

TO BE CRYPTIC OR NOT TO BE CRYPTIC

Several students of C have stated that they didn't like these three cryptic constructs and that they

would simply never use them. This will be fine if they never have to read anybody else's program, or use any other programs within their own. I have found many functions that I wished to use within a program but needed a small modification to use it, requiring me to understand another person's code. It would therefore be to your advantage to learn these new constructs, and use them. They will be used in the remainder of this tutorial, so you will be constantly exposed to them.

This has been a long chapter but it contained important material to get you started in using C. In the next chapter, we will go on to the building blocks of C, the functions. At that point, you will have enough of the basic materials to allow you to begin writing meaningful programs.

STYLE ISSUES

We have no specific issues of style in this chapter other than some of the coding styles illustrated in the example programs. Most of these programs are very nontypical of real C programs because there is never a need to list all of the possible compares in a real program, for example. You can use the example programs as a guide to good style even though they are not real programs.

WHAT IS AN l-value AND AN r-value ?

You will sometimes see a reference to an l-value or a r-value in writings about C or in the documentation for your C compiler. Every variable has a r-value which is defined as the actual value stored in the variable, and it also has an l-value which is defined as its address, or the name of the variable. Therefore, the variable depicted graphically in figure 4-1 has an l-value of index, and the r-value of 137 since 137 is the value stored in the variable at this time.

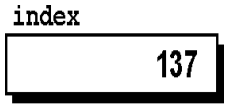

Figure 4-1

The definition for this variable would be given as follows;

```
int index = 137;
```

Programming Exercise:

---

1. Write a program that will count from 1 to 12 and print the count, and its square, for each count.
2. Write a program that counts from 1 to 12 and prints the count and its inversion to 5 decimal places for each count. This will require a floating point number.
3. Write a program that will count from 1 to 100 and print only those values between 32 and 39, one to a line. Use the incrementing operator for this program.
