Chapter 5

Numeric Data Types and Expression Evaluation

In the preceding chapters we have introduced all the basic tools needed to write programs in C: the control constructs and operators of the language, as well as the basic data types for integer, floating point, and character data. Using these basic tools, we have been able to write programs for both numeric processing and non-numeric, character, processing.

In this chapter we will introduce several useful features of C that allow greater flexibility in program writing and allow a greater range of values and precision. We will first take a closer look at integer and floating point data types; their size, and limitations, and will introduce sub-types of integers, and double precision floating point numbers. We will formalize the order of evaluation of operators in expressions as well as the type of the expression value when several data types are are present as operands. We will also introduce several C statements that are possible alternatives for statements already discussed and describe some new operators.

5.1 Representing Numbers

As we saw in Chapter 1, the range of possible values of objects depends on the sizes used to represent them. The finite size of an object puts a limit on the range of values that can be stored in it. Integer objects have a limit on the range of positive and negative integers. Floating point numbers have limits on the number of significant digits (known as the **precision**) as well as on the range of the exponents (limiting the range of numbers). We will illustrate the reasons for these limits by analogy with decimal representation.

Let us represent integers using a finite number of decimal digits, say only five digits are allowed. We can use these digits to represent unsigned positive integers in the range 0 to 99999. If we wish to represent both positive and negative numbers, we need one digit to encode the sign, `+` or `-`, and can then use only the remaining four digits to represent the absolute value of an integer. So, with five digits, we can represent positive and negative integers in the range -9999 to +9999. If we had more digits to represent integers, the range of values will be appropriately greater.

Now let us use the same five digits to represent floating point numbers in scientific notation, i.e. a fractional part multiplied by a power of ten. For our discussion, we will assume that the fractional part is less than 1 and that the exponent of ten can be positive or negative. For example:

```
.234E3
.987E-2
-.345E2
```

The numbers are shown as a fraction times some power of ten where the exponent is shown after the `E`. The first number is 234.0, the second is .00987, the third is -34.5.

When we represent numbers using this system, we do not need to store the decimal point (it is always in the same place) or the base (it is always `E`, standing for 10). So, of our 5 digits, let us say that we use three digits for the fractional part and two digits for the exponent. One digit of the fractional part and one digit of the exponent is reserved for the sign. This leaves only two digits for the absolute value of the fractional part, and it leaves one digit for the absolute value of the exponent. Thus, the range of values for the fractional part is $-.99$ to $+.99$ and the range for exponents is -9 to $+9$.

Even though the range of actual values is quite large (we can represent numbers from almost negative one billion to positive one billion), there are only two significant digits of precision; all other digits will be zeros contributed by the power of ten. So, the range of numbers is from $-990,000,000$ to $+990,000,000$ (`-.99E+9` to `+.99E+9`). With this scheme, it would be impossible to represent a number such as 123.4567 *exactly.* The best we can do is represent it as `+.12E+3`, which is the number 120 — not nearly as accurate as 123.4567. We have a loss of precision (or accuracy) because of the limited number of digits we have for representing floating point numbers. There is a slight distinction between *precisions* and *accuracy.* In the above representation scheme, we can always say there are 2 digits of precision; however, the accuracy depends on the value of the exponent. The smallest number we can represent is .00000000099 (`+.99E-9`), which is pretty darn accurate. However, if the exponent is +9, our accuracy is only $\pm$ 5 million. If more digits are used to represent floating point numbers, the precision and the range can be greater. For example, if 6 digits were allowed, with four digits for a signed fractional part, we could represent 123.4567 as `+.123E3`, which is 123.0. If 7 digits were allowed, with 5 digits for a signed fractional part, we could represent the same number as `+.1234E3`, which is 123.4, and so forth.

Conceptually, binary representation of numbers is no different from decimal representation. The finite size imposes a limit on the range of integers and on the precision and range of floating point numbers. Binary representation is also tailored to facilitate the basic operations in hardware, such as addition and subtraction. For example, as we saw in Chapter 1, integers are typically represented in what is called the two's complement number system. However, one does not need to know the number system to realize that the limits on the range of values will be similar in nature and will depend on the sizes used to represent the numbers.

Recall that, in a computer, memory is organized as a sequence of bytes, each byte with an address, and storage is allocated in units of bytes. For example, if 1 byte is used for signed integers, the range of values (in decimal) is -128 to 127; and unsigned integers have the range 0 to 255. If 2 bytes are used to represent signed integers, the range is -32768 to +32767; and 0 to 65535 for unsigned integers. If 4 bytes are used to represent integers, the range will be appropriately greater. Similarly for floating point numbers; with 4 bytes to represent floating point numbers, the precision is equivalent to about 7 significant decimal digits and a magnitude between approximately 10E38 and 10E-38. If more bytes are used for floating point numbers, the precision and the range are both appropriately greater.

So far we have used `char`, `int`, and `float` data types in our programs. Character data type is usually encoded as an ASCII integer value (signed or unsigned) in one byte of memory. Integers are at least two bytes in size, and floating point numbers are at least four bytes in size. C provides additional integer sizes and floating point data types that provide greater range and/or precision.

5.1.1 Signed and Unsigned Integer Types

For integer data types, there are three sizes: `int`, and two additional sizes called **long** and **short**, which are declared as `long int` and `short int`. The keywords `long` and `short` are called **sub-type qualifiers**. The `long` is intended to provide a larger size of integer, and `short` is intended to provide a smaller size of integer. However, not all implementations provide distinct sizes for them. The requirement is that `short` and `int` must be at least 16 bits, `long` must be at least 32 bits, and that `short` is no longer than `int`, which is no longer than `long`. Typically, `short` is 16 bits, `long` is 32 bits, and `int` is either 16 or 32 bits.

Unless otherwise specified, all integer data types are *signed* data types, i.e. they have values which can be positive or negative. Recall, `char` types, without qualifiers, may be signed or unsigned depending on the implementation. However, all sizes of integers and `char` type may be explicitly qualified as `signed` or `unsigned`. (Unsigned numbers are always non-negative numbers).

For integers, `long`, `short`, and `unsigned` may be declared with the keyword `int` or without it. In C, whenever a data type is left out in a declaration, `int` is assumed by default. Here are some example declarations:

```
long int light_year;
short int n;
signed char ch;
unsigned char letter;
unsigned int age;
long distance;
short m, n;
unsigned memory_address;
unsigned long zip_code;
```

The data type of a constant, written directly into a program, is ascertained from the way it is written. Integer constants are written as a string of digits, optionally preceded by a unary positive or a negative operator. Commas are not allowed. Decimal integer constants should be written without leading zeros, for example:

```
   29
 -173
    0
    1
 -525
+7890
```

Alternate number systems may also be used to express integer constants in C programs. Octal numbers are written with a leading zero, and hexadecimal numbers are written with a preceding zero followed by the letter `x` or `X`:

```
Constants           Octal/Hexadecimal Integers

0234                octal number 234
0101                octal number 101
0x34                hexadecimal number 34
0X1F                hexadecimal number 1F
```

A constant to be represented as a `long int` may be explicitly written using the suffix `l` or `L`, as in:

```
123L
45678l
```

Any integer constant that is too big to fit into the integer size is interpreted by the compiler as `long`.

Unsigned integers can be of all sizes, `int`, `long`, and `short`. The range of unsigned integers is 0 through 2^{k-1}, where k is the number of bits, so for 16 bits the maximum unsigned integer is 65535. Unsigned integer constants are written using the suffix, `u` or `U`:

```
0xFFFFU
123U
0777u
```

The two suffixes can be combined to write an `unsigned long`:

```
12345678UL
0X8FFF FFFFLU
```

5.1.2 Single and Double Precision Floating Point Numbers

Different sizes of floating point data can also be declared with the keywords `float` and `double`. The type specifier `double` is used to declare *double precision* floating point numbers. The size of `float` is typically 32 bits, and that of `double` is 64 bits. For greater precision, most scientific and engineering computation should be performed using the `double` data type. Furthermore, *extra precision* may be provided for floating point numbers by declaring them `long double`. (This may be the same as or more bits of precision as `double`, depending on implementation). Here are example declarations for floating point numbers:

```
float  x;
double GPR;
long double y;
```

Decimal `float` constants in programs have an integer part and a fractional part with a decimal point between them. They may also be written in scientific (or exponential) notation, i.e. a decimal number multiplied by a power of ten to indicate the actual position of the decimal point. Positive and negative numbers may be written with an explicit positive or negative unary operator.

```
123.789
0.5534
+9635.0000
-8942.3214
-0.765E5
1.4523e12
0.786345e-10
```

The last three numbers are written in exponential notation with the exponent of ten shown after the letter `e` or `E`. The exponent may be a positive or a negative integer. For clarity, always write `float` numbers with at least one digit before and one after the decimal point; for example, zero is `0.0` in `float` representation.

Floating point constants are taken to be of double precision type by default. Single precision floating point constants may be specified with a suffix `f` or `F`.

```
34.567f
3.141516F
```

Extra precision for constants may be written with the suffix `l` or `L`:

```
23456789.171819L
```

5.2 New Control Constructs

So far, we have seen all of the basic control constructs of the C language for calling functions, branching, and looping. In this section we introduce two new looping constructs that can be used in place of `while`; namely `for` loops and `do...while` loops.

5.2.1 The `for` Statement

The logic of the loops we have constructed so far has included three components: some form of initialization before the loop, a test for loop termination, and some form of data update within the body of the loop. We implemented these loops using three separate statements in the program, with a `while` statment forming the condition test and loop body. Another looping construct combines all three components of a loop in a single statement: the `for` statement.

The syntax for the `for` statement is:

```
for (<expr1>; <expr2>; <expr3>) <statement>
```

The keyword, `for`, and the parentheses are required as shown. Notice the three expressions are separated by semi-colons (`;`). The semantics of the `for` statement is as follows. The expression, <expr1>, is evaluated once before the loop condition is tested for the first time; <expr2> is the loop condition which is evaluated prior to each execution of the loop body; and <expr3> is evaluated at the end of the loop body, just prior to testing the condition. The process repeats until the loop condition becomes False. The body of the loop is <statement>, which, as usual, may be any valid type of C statement; empty, simple, or compound. As with the `while` loop, if the loop condition evaluates to True, the loop body is executed; otherwise, if the loop condition evaluates to False, the loop is terminated, and control passes to the next statement following the `for` statement. In typical use, the expressions, <expr1> and <expr3> initialize and update a variable, respectively. Figure 5.1 shows the control flow for a `for` statement.

A `for` statement includes all the necessary features of a loop: an initialization expression, a loop condition, and an update expression. Thus, the following two forms of implementing a loop are equivalent:

```
<expr1>;
while (<expr2>) {
        <statement>    and        for (<expr1>; <expr2>; <expr3>) <statement>
        <expr3>;
}
```

The `break` and `continue` statements can also be used in the body of a `for` statement, just as in a `while` statement. The use of a `for` statement or a `while` statement to implement a loop is a matter of choice, based on the logic of the algorithm. One advantage might be that writing a `for` statement reminds one that initialization and update expressions are usually necessary for a loop.

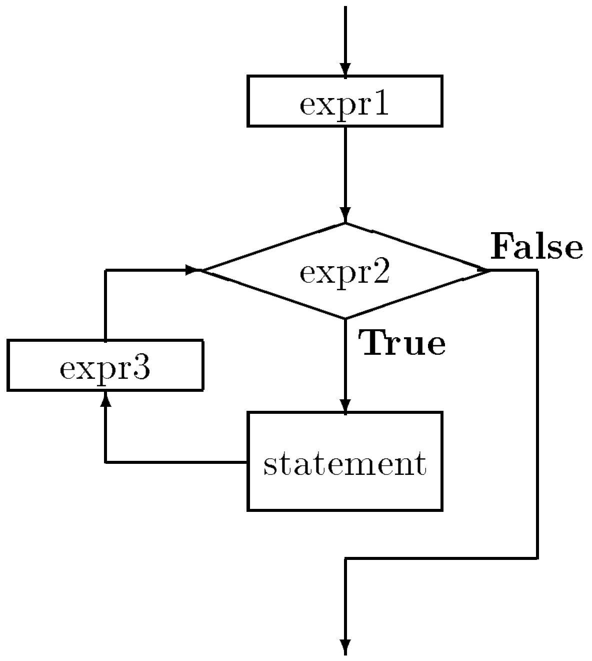

Figure 5.1: Control Flow of `for` Loop

An Example: Factorial

Let us consider an example task which may require a bigger range of integers than the one provided by `int` on many machines. The task is to determine a cumulative product from 1 to a positive integer, n. The product from 1 to n is called the factorial of n, written $n!$. The algorithm is very simple: read an integer `n`; call a function `fact(n)` which returns the factorial of `n`; print the result.

The function `fact()` merely needs to multiply a cumulative product variable, initialized to 1, by all integers from 1 through `n`:

```
initialize product to 1
repeat for values of i = 1, 2, 3,..., n
    product = product * i
return product
```

The variable, `product`, must be initialized to 1 before the loop, otherwise the cumulative product will be garbage. Each iteration brings us closer to the result. We will use a `for` statement to implement the iterative algorithm for a factorial function as shown in Figure 5.2. The `for` loop executes as follows. The first expression in parentheses is an initialization expression, i.e. `i` is initialized to 1. The second expression is the loop condition. If the second expression, `i` <= `n`, evaluates to True, then the loop body is executed. The third expression is the update expression; it is evaluated after the loop body is executed, and control then passes to the loop condition. In our example, the expression, `i = i + 1`, is evaluated to update the variable, `i`, after the loop body

```
/*   File: fact.c
     Program computes the factorial of integers using function
     fact().
*/
#include <stdio.h>
int fact(int n);
main()
{    int n;

     printf("***Factorial Program***\n");
     printf("Type positive integers, EOF to terminate\n");
     while (scanf("%d", &n) != EOF)
          if (n <= 0)
               printf("%d typed, type a positive integer\n", n);
          else
               printf("Factorial of %d is %d\n", n, fact(n));
}

/*   Function computes factorial of n using a for loop. */
int fact(int n)
{    int i, product;

     product = 1;
     for (i = 1; i <= n; i = i + 1)
          product = product * i;
     return product;
}
```

Figure 5.2: Code for factorial

is executed. The loop condition is then tested, and the process repeats until the loop condition becomes False. The above loop executes for `i = 1, 2, 3, ...`, and `n` and the variable `product` accumulates the factorial value of $1 * 2 * 3 * ... * n$.

The driver uses a `while` condition:

```
     (scanf("%d", &n) != EOF)
```

where `scanf()` reads an integer item if possible and stores it in `n`. The value returned by `scanf()` is then compared with `EOF` and if the value returned is NOT `EOF`, the loop executes. As soon as `scanf()` returns `EOF`, the loop is terminated. The while expression serves both to read an item and to check if the returned value is `EOF`. The loop body tests the value of `n`; if it not a positive integer, the user is asked to retype a positive number; otherwise, the value of `fact(n)` is printed.

Here is a sample session run on an IBM PC:

```
***Factorial Program***
Type positive integers, EOF to terminate
4
Factorial of 4 is 24
5
Factorial of 5 is 120
-3
Negative number -3 typed, type positive integers
6
Factorial of 6 is 720
7
Factorial of 7 is 5040
8
Factorial of 8 is -25216
^Z
```

The cumulative product in the factorial function grows very fast with `n`. For moderately large values of `n`, the cumulative product *overflows* the `int` type object; the number is too large for the size of the object. When this occurs, the results are meaningless. Usually, an overflow is indicated when a program, working correctly for smaller numbers, gives ridiculous results for larger numbers. In the case of the factorial function, the first sign of trouble is a negative result for the factorial of 8. We know the result must be positive since we are multiplying only positive numbers. What has happened is the result has overflowed into the sign bit resulting in a negative integer. If factorial of larger numbers is desired, a `long int` variable should be used for the variable `product` as well as for the function `fact`. Here is a revised version of the factorial function.

```
/*   Function computes a long factorial of n using a for loop. */
long longfact(int n)
{    long int product;
     int i;

     product = 1;
     for (i = 1; i <= n; i = i + 1)
          product = product * i;
     return product;
}
```

We must keep several things in mind when using the function, `longfact()`, in the driver program. In the calling function, if the value returned by `longfact()` is saved, it must be assigned to a long integer; otherwise, a `long` result would be converted to `int` by dropping higher order bits and the result would be meaningless. In addition, to print the `long` value of `longfact()`, the conversion specifier must be qualified by the prefix `l`:

```
printf("Factorial of %d is %ld\n", n, longfact(n));
```

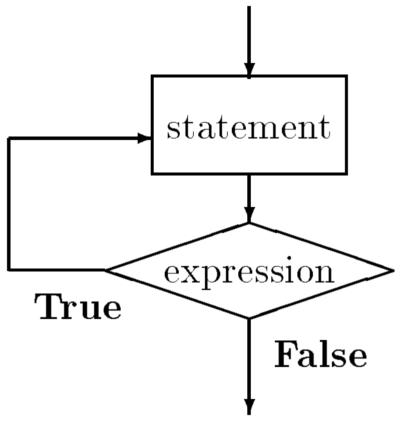

Figure 5.3: Control Flow of `do...while` Loop

The conversion specifier, `%ld`, prints a long decimal integer.

This example has shown a case where the size of the type `int` is smaller than the type `long`, as it is in some implementations. The situation could be corrected by using a larger size data type to accumulate the factorial. However, even this type has limitations; the factorial of 13 will overflow the size of a long integer. The only possibility provided for even larger numbers is to use a floating point representation, which has a larger range, at the expense of loss of precision.

5.2.2 The `do...while` Statement

In `while` statements and `for` statements, the condition is tested for each iteration *before* the loop body is executed. Thus, it is possible that the loop may not be executed even once if the loop condition evaluates to False the first time. The C language provides another looping construct which guarantees that the body will be executed at least once: the `do...while` statement. The loop condition is tested *after* the body is executed, and the loop continues or terminates depending on the condition value. The syntax for the `do...while` statement is:

```
do
        <statement>
while (<expression>);
```

Figure 5.3 shows the control flow for this construct. As with the other loop constructs, the `break` and `continue` statements can also be used with the `do...while` statement. The choice of a loop construct depends on the program logic. There are situations when one construct may be preferable to another.

An Example: Square Root

Programs are often written to find a solution (or solutions) to an algebraic equation; for example:

$$y^2 - x = 0$$

Here, the solution for the variable, y, is the square root of x. In general, such solutions are real numbers, and as we have seen, floating point representations of real numbers use a finite number of bits, and are therefore limited in the precision of the result. Solutions to most numeric problems can never be exact (all solutions are precise only up to a certain number of decimal digits) but the result may be sufficiently close to the real solution to be acceptable.

One important numeric computation method to find solutions to equations involves successive approximations. This method starts with a *guess* for the solution to the problem, and tests if the guess satisfies the equation. If the guess is *close enough* for a solution, it is accepted and computation terminates; otherwise, the guess is *improved*, i.e. brought closer to the solution and the process is repeated. After each iteration, the guess is closer and closer to the solution, until it is acceptably close enough.

One successive approximation algorithm we will use is Newton's method to compute the square root of a number, x. Newton's method starts with an arbitrary guess, and if it is not good enough, it is improved by averaging the guess with $x/guess$. The process continues until the guess is close enough. Here is an example of the process for square root of 9.0:

$guess$	$x/guess$	Average
1.0	9.0	$(1.0 + 9.0)/2.0$
5.0	1.8	3.4
3.4	2.647	3.023
3.023	...	

In just three iterations, we have arrived close to the square root of 9.0 (which is 3.0). We will say a guess is close enough to the solution, if x and the square of $guess$ differ by a small value, say 0.001, or less. The algorithm is simple:

```
begin with an initial guess
repeatedly do the following
     improve the guess
while it is not close enough
```

We will start with an arbitrary guess, say 1.0, for the square root of the number, x. In a loop, each iteration improves the guess of the square root of x until the guess is close enough. In our implementation, we assume two functions: one to test if a guess is close enough, and the second to improve the guess. This algorithm works for any successive approximation method; the only difference would be how to improve the guess, and how to check the guess for closeness to the solution. Here is the code fragment for square root using a `do...while` statement:

```
    guess = 1.0;
    do
         guess = improve(guess, x);
    while (!close(guess, x))
```

The body of the loop follows the keyword `do`. The loop body is executed and then the `while` expression is tested for True or False. If it is True, the loop is repeated; otherwise, the loop is terminated. The above loop body calls on a function `improve()` to improve the guess and the condition is then tested to see if the improved guess is close enough by the function `close()`.

As we said, the difference between `do...while` and the other loop constructs is that in this case the loop is executed at least once; `while` loops and `for` loops may be executed zero times if the loop condition is initially False. In the case of successive approximations, we always expect the initial guess to need improvement; so, the loop must be executed at least once.

Figure 5.4 shows the implementation of the driver. The source file includes a header file `mathutil.h` that declares the function prototypes for `close()`, `improve()`, and other functions defined in a source file, `mathutil.c`, shown in Figure 5.5. The two source files `sqroot.c` and `mathutil.c` must be compiled and linked to create an executable file. Here is `mathutil.h`:

```
/* File: mathutil.h */
/* File contains prototypes for functions defined in mathutil.c */
double improve(double guess, double x);
int close(double guess, double x);
double absolute(double x);
```

Notice we have used the type, `double` for the parameters and return values of the functions because precision is important in successive approximation algorithms. It is best to use double precision in all such computations. We have also included the header file, `tfdef.h`, which defines the symbolic constants `TRUE` and `FALSE`.

The program driver uses a loop to read a positive, double precision number into `x` using the conversion specification `%lf`. (When a double precision number is printed, conversion specification is still `%f` since a printed double precision floating point number looks the same as a single precision number). If the number read into `x` is negative or zero, a message is printed and the loop is repeated until a positive number is read. We have used the `do...while` construct here, since we know that the loop must be executed at least once to get the desired data.

Next, `guess` is initialized to 1.0 and the loop body improves `guess` We have included a debug statement to print the value of the improved guess during program testing. The loop repeats until `guess` is close enough to be an acceptable solution.

We still need to write the functions `improve()` and `close()`. The function `close()` tests if the absolute value of the difference between the square of `guess` and `x` is small enough. We will use a function, `absolute()`, that returns the absolute value of its argument. Figure 5.5 shows `close()` and `absolute()` in the source file, `mathutil.c`. Some of the functions defined in this source file

```
/*   File: sqroot.c
     Other Files: mathutil.c
     Header Files: tfdef.h, mathutil.h
     Program computes and prints square roots of numbers. Uses Newton's
     method to compute square root of x: Start with any guess. Test if
     it is acceptable. If not, improve guess by averaging it with x/guess.
*/
#include <stdio.h>
#include "tfdef.h"
#include "mathutil.h"
#define DEBUG
main()
{    int i;
     double x, guess;

     printf("***Square Root Program: Newton's Method***\n\n");
     printf("Type a positive number: ");
     do {
          scanf("%lf", &x);
          if (x <= 0)
               printf("%f typed, type a positive number\n", x);
     } while (x <= 0);
     guess = 1.0;
     do {
          guess = improve(guess, x);         /* improve guess.     */
          #ifdef DEBUG                            /* debug stmt     */
               printf("guess = %f\n", guess);     /* Print guess.  */
          #endif                                  /* end of debug  */
     } while (!close(guess, x));        /* terminate if guess is close */
     /* exit loop if guess is close enough */
     printf("Sq.Rt. of %f is %f\n", x, guess);    /* Print sq. rt. */
}
```

Figure 5.4: Code for Square Root

```
/* File: mathutil.c */
#include <stdio.h>
#include "tfdef.h"
#include "mathutil.h"
/* Tests if square of guess approximately equals x. */
int close(double guess, double x)
{
     if (absolute(guess * guess - x) < 0.001)
          return TRUE;
     else
          return FALSE;
}

/* Returns absolute value of x. */
double absolute(double x)
{
     if (x < 0)
          return -x;
     else
          return x;
}

/* Returns average of guess and x / guess. */
double improve(double guess, double x)
{
     return (guess + x / guess) / 2;
}
```

Figure 5.5: Code for Math Utilities

are also called within it, e.g. `absolute()`, so we have included `mathutil.h` in this source file, as well as `tfdef.h`, which defines `TRUE` and `FALSE`. Finally, we write the function `improve()` which merely returns the average of `guess` and `x / guess`.

Sample Session:

```
***Square Root Program***

Type a number:  16
guess = 8.500000
guess = 5.191176
guess = 4.136665
guess = 4.002257
guess = 4.000000
Sq.Rt.  of 16.000000 is 4.000000
```

The debug statement shows how guess is changed at each step. Once we are satisfied with the program, we can remove the definition of `DEBUG`.

Next, we modify our program to encapsulate it into a function, `sqroot()`, and to provide user control over the precision desired for the solution instead of building it into the function, `close()`. The `sqroot()` function requires two arguments, a number and an acceptable error in the solution. We also require a new function `close2()` that checks if a given guess is close enough to a solution with a specified margin of error. With this modification, it is not necessary to use `double` for numbers in `main()`. Only the actual computations need to be `double` type for greater precision. Figure 5.6 shows the revised driver in which `float` numbers are used in `main()` and the function `sqroot()` is called to find the square root. Figure 5.7 shows the prototypes added to `mathutil.h` and the new functions in `mathutil.c`. The driver simply repeats the following loop: read a number; if the number is negative, continue the loop; otherwise, call `sqroot()` to find the square root of the number within specified margin; print the value. The function `sqroot()` merely starts with a guess and improves it in a loop until it is within an allowable margin of error. The final acceptable guess is returned. The function `close2()` tests if a guess is close to the solution within a specified error.

In `main()`, numbers are read into `float` variables, so when arguments are passed to `sqroot()`, they are cast to `double`. Likewise, the returned `double` value is cast to `float` before assigning it to the variable `root`. Here is the statement that uses cast operators to convert types:

```
root = (float) sqroot((double) x, 0.001);
```

Recall that a floating point constant is always assumed to be of type `double`. If function prototypes are declared, we don't have to convert the types explicitly by cast operators, the compiler will take care of that for both the arguments and the returned value. However, the explicit cast operators improve readability by showing that conversions are taking place.

Sample Session:

```
/*   File: sqrt2.c
     Other Files: mathutil.c
     Header Files: tfdef.h, mathutil.h
     Program computes and prints square roots of numbers until the end of
     file. Uses Newton's method to compute the square root of x to within a
     specified error margin.
*/
#include <stdio.h>
#include "tfdef.h"
#include "mathutil.h"
main()
{    int i;
     float x, root;

     printf("***Square Root Program***\n\n");
     printf("Type positive numbers, EOF to quit: ");
     while (scanf("%f", &x) != EOF) {
          if (x <= 0) {
               printf("%f typed, type positive numbers \n");
               continue;
          }
          root = (float) sqroot((double) x, 0.001);
          printf("Sq.Rt. of %f is %f\n", x, root);
     }
}
```

Figure 5.6: Modified Square Root Driver

```
/* File: mathutil.h - continued */
double sqroot(double y, double error);
int close2(double g, double y, double error);


/* File: mathutil.c - continued */
/* Uses Newton's method to compute square root within the margin
   allowed by error.
*/
double sqroot(double y, double error)
{    double guess = 1.0;

     do
          guess = improve(guess, y);    /* improve guess. */
     while (!close2(guess, y, error));  /* while guess not close */
     return guess;                 /* when close enough, return guess.*/
}

/* Tests if square of g equals y within the error limits specified. */
int close2(double g, double y, double error)
{
     if (absolute(g * g - y) < error)
          return TRUE;
     else
          return FALSE;
}
```

Figure 5.7: Modified Square Root Utilities

```
***Square Root Program***

Type positive numbers, EOF to quit:  16
Sq.Rt.  of 16.000000 is 4.000000
13
Sq.Rt.  of 13.000000 is 3.605551
19
Sq.Rt.  of 19.000000 is 4.358902
25
Sq.Rt.  of 25.000000 is 5.000023
^D
```

The last example shows the square root of 25.0 to be slightly different from the correct value of 5.0, but within our allowed error of 0.001. It must be remembered that floating point representation cannot be exact due to the finite number of bits used. Therefore, if the error specified were very small, it may not be possible to arrive at an answer with the desired accuracy. That is, the guess may never converge to a value such that `close2()` returns True and the loop in `sqroot()` would never terminate. In successive approximations algorithms, one must guard against possible lack of convergence such as by putting a limit on the number of loop iterations allowed.

In Chapter 6 we will see that standard library functions are available to compute the square root and the absolute value of a number. Our emphasis here has been to illustrate program development using just the basics of a programming language, viz. expressions including assignments, branching, and looping.

5.3 Scalar Data Types

All of the data types we have seen so far, `char`, `int`, `short long`, `float`, and `double` are called **scalar** (or base) data types because they hold a single data item. (Chapters 7 and 12 describe compound data types provided in C). There are two other scalar types in the language: `enum` and `void` which are described in this section. We will refer to `float` and `double` types as floating point types and to all sizes of integers, `char` and `enum` types as integral types. In addition, we describe how a user defined type may be declared.

5.3.1 Data Type `void`

The data type `void` actually refers to an object that does not have a value of any type. The most common example of its use is when we define a function that returns no value. For example, a function may only print a message and no return value is needed. Such a function is used for its side effect and not for its value. In the function declaration and definition, it is necessary to indicate that the function does not return a value by using the data type `void` to indicate an empty type, i.e. no value. Similarly, when a function has no formal parameters, the keyword `void` is used in the function prototype and header to signify that there is no information passed to the function.

Here is a simple program using a message printing function which takes a `void` parameter and returns type `void`:

```
/*   File: msg.c
     This program introduces data type void.
*/

void printmsg(void);

main()
{

     /* print a message */
     printmsg();
}

/* Function prints a message. */
void printmsg(void)
{
     printf("****HOME IS WHERE THE HEART IS****\n");
}
```

No parameters are required for the function, `printmsg()`, and it returns no value; it merely prints its message. In the function call in `main()`, parentheses must be used without any arguments. Observe that no return statement is present in `printmsg()`. When a function is called, the body is executed and, when the end of the body is reached, program control returns to the calling function. Such a return from a called function without a `return` statement is often called returning by *falling off the end*. There are times when it is necessary to return from a `void` function before the end of the body. In such case, a `return` statement, with an empty expression may be used to return nothing:

```
void printmsg(void)
{
     printf("****HOME IS WHERE THE HEART IS****\n");
     return;
}
```

A `return` statement can also be used elsewhere in the body to return control immediately to the calling function. Consider a function which prints the values of its arguments if they are all positive; otherwise it does nothing:

```
     void func(int x, in y)
```

```
{
     if (x <= 0 || y <= 0)
          return;
     printf("x = %d, y = %d\n", x, y);
}
```

If either of the arguments is not positive, the function returns to the calling function. If it does not return, then it prints the values of the arguments.

The use of `void` for a function returning no value is not strictly necessary. We could declare the function as being type `int` (or any other type) and simply not return any value and never use the value of the function in an expression. However, the `void` declaration makes the nature of the function explicit to someone reading the code and may allow the compiler to generate more efficient object code.

5.3.2 Enumeration

The data type, `enum` (for enumeration) also allows improvement in program clarity by specifying a list of names, the enumeration constants, which are associated with constant integer values. It is similar to using `#define` directives to define constant integer values for a set of symbolic names; however, with `enum` the compiler can generate the values for you, and may check for proper use of `enum` type variables. A variable of `enum` type is declared as follows:

```
enum { FALSE, TRUE } flag;
```

The variable, `flag`, is defined here to be of a type which can take on the two enumerated constant values, `FALSE` and `TRUE`. Normally, enumeration constants are identifiers whose values start at zero and increase in sequence: here, `FALSE` is 0, and `TRUE` is 1. However, the enumeration can have explicit constant values specified in the enumeration:

```
enum { SUN = 1, MON, TUE, WED, THU, FRI, SAT } day;
```

Here, `SUN` is associated with value 1, and the rest of the names have values in increasing sequence: `MON` is 2, `TUE` is 3, and so on until `SAT` which is 7. The variable, `day` can hold any of the enumerated values.

An enumeration type can be given a *tag*, i.e a name which can be used later to declare variables of that tagged enumeration type. For example, we can name an enumeration:

```
enum boolean { FALSE, TRUE };
```

where the name `boolean` can then be used to declare variables of that enumeration type:

```
enum boolean flag1, flag2;
```

This declaration defines variables, `flag1` and `flag2`, which are of a type specified by the boolean enumeration; that is, `flag1` and `flag2` can have values `FALSE` or `TRUE`. It is also possible to specify a tag and declare variables in the same declaration:

```
enum boolean {FALSE, TRUE} done;
enum boolean found;
```

The first declaration specifies a tag, `boolean`, for the enumeration as well as declaring a variable, `done` of this type. The second declaration defines a variable, `found`, of the enumeration `boolean` type. Here is a function, `digitp()`, that returns a `boolean` value to the calling function. (The calling function must also declare the enumeration in order to use the returned value correctly).

```
enum boolean { FALSE, TRUE };

enum boolean digitp(char c)
{
     if (c >= '0' && c <= '9')
          return TRUE;
     else
          return FALSE;
}
```

Remember, the *value* of an `enum` type variable is an integer. An enumerated data type is primarily a convenience for writing the source code; information about the symbolic names are not retained at run time. For example, if we were to execute a statement:

```
printf("digitp returns %d\n",digitp('0'));
```

it would print

```
digitp returns 1
```

NOT

```
digitp returns TRUE
```

However, some symbolic debuggers may use the enumerated names for displaying debugging information.

5.3.3 Defining User Types: `typedef`

The C language provides a facility for defining *synonyms* for data types to make programs more readable. New data types that are equivalent to existing data types may be created using the `typedef` declaration. The syntax is:

typedef <existing-type-specifier> <new-type-specifier>;

The scope of a type definition is from the point of definition to the end of the source file. Variables can then be defined in terms of these new types. For example, variables used to represent values of age of people or objects can be defined to be of a new type, `age`.

```
typedef int age;
age yrs;
```

The variable `yrs` can have `age` type values. In this case, the primary difference is that we can have more meaningful names for data types than the generic name `int`.

A `typedef` definition is also commonly used to "hide the details" of more complicated declarations:

```
typedef enum { FALSE, TRUE } boolean;

boolean flag;
```

The type definition defines data type, `boolean` which is a synonym for an enumerated type consisting of two constant values `FALSE` and `TRUE`. Variables of type `boolean` can now be defined, and they can be assigned one of the enumerated values. In fact, the name, `boolean`, can be used like any other data type. Functions can have `boolean` parameters and can return `boolean` values. For example, we could write:

```
flag = TRUE;

if (flag)
     printf("Flag is true\n");
```

Let us consider the task of a simple calculator. It should read two numbers and then read an operator that is to be applied to the operands. The operator should be applied to the operands and the result printed. (When an operator appears after the operands, the expression is said to be in **postfix form**). The algorithm for a postfix calculator is:

```
repeat until end of file or error in reading numbers
     read two numbers and an operator
     apply operator to the numbers and get result
     print result
```

The program must make sure that two valid numbers and an operator are read correctly. We will ensure that two numbers are read correctly by examining the value returned by `scanf()`. The buffer will then be scanned and flushed until a valid operator is found. The program is shown in Figure 5.8.

The `while` loop continues until `scanf()` is unable to read two numbers. If `scanf()` reads two numbers, it returns a value of 2, and the loop is executed. In the loop, we use `get_operator()` to get a valid operator. The function, `get_operator()` will scan each new character in the keyboard buffer until an acceptable operator is found. Once an operator is read, an error flag of type `boolean` is initialized to `FALSE`.

A switch statement is used to determine the result of applying the operator to the operands. The division operator can lead to trouble if `oprnd2` is zero; divide by zero is a fatal error and the program would be aborted. We trap this error by testing for a zero value of `oprnd2`, in which case we set `error` to `TRUE`. If there is no error, the result is printed ; otherwise, an error message is printed. The loop repeats until `scanf()` does not read 2 `floats` (including detecting `EOF`).

The function `get_operator()` consists of a loop that continues to read a character until a valid operator is read; skipping over any white space and any erroneous characters. It uses a `boolean` type function, `operatorp()`, to test if an argument is an acceptable operator. Figure 5.9 shows the required functions.

Sample Session:

```
***Postfix Calculator***

Type two numbers, followed by an operator:  +, -, *, or /
EOF to quit
12    12
   +
12.000000 + 12.000000 = 24.000000
50    0
/
Runtime error:  50 / 0
^D
```

We have purposely used a lot of white space to show that the calculator functions correctly.

5.4 Operators and Expression Evaluation

Once we can declare data to be the type and size with the appropriate precision for our task, we would like to perform operations with the data. We have already discussed some of the basic C operators, and in this section we provide the complete precedence table for all C operators. We

```
/*   File: calc.c
     This program is a postfix calculator. Two operands followed by an
     operator must be entered. The program prints the result. The program
     repeats until end of file.
*/
#include <stdio.h>
typedef enum { FALSE, TRUE } boolean;
char get_operator(void);
boolean operatorp(char c);

main()
{    float oprnd1, oprnd2, result;
     char c;
     boolean error;

     printf("***Postfix Calculator***\n\n");
     printf("Type two numbers, followed by an operator: +, -, *, or /\n");
     printf("EOF to quit\n");

     while (scanf("%f %f", &oprnd1, &oprnd2) == 2) {
          c = get_operator();
          error = FALSE;

          switch(c) {
               case '+': result = oprnd1 + oprnd2; break;
               case '-': result = oprnd1 - oprnd2; break;
               case '*': result = oprnd1 * oprnd2; break;
               case '/': if (oprnd2)
                              result = oprnd1 / oprnd2;
                         else
                              error = TRUE;
                         break;
          }

          if (error == FALSE)
               printf("%f %c %f = %f\n", oprnd1, c, oprnd2, result);
          else
               printf("Runtime error: %f %c %f\n", oprnd1, c, oprnd2);
     } /* end of while loop */
} /* end of program */
```

Figure 5.8: Code for Simple Postfix Calculator

```
/* File: calc.c - continued */
/* Gets one of the allowed operator, +, - , *, /. */
char get_operator(void)
{    char c;

     while ((c = getchar()) && operatorp(c) != TRUE)
          ;
     return c;
}

/* Function tests if c is one of the operators +, -, *, /. */
boolean operatorp(char c)
{
     switch(c) {
          case '+':
          case '-':
          case '*':
          case '/': return TRUE;
          default: return FALSE;
     }
}
```

Figure 5.9: Code for `get_operator()`

present a few new operators here, and others shown in the table will be discussed in detail in later chapters.

5.4.1 Precedence and Associativity

The data type and the value of an expression depends on the data types of the operands and the order of evaluation of operators which is determined by the precedence and associativity of operators. Let us first consider the order of evaluation. When expressions contain more than one operator, the order in which the operators are evaluated depends on their precedence levels. A higher precedence operator is evaluated before a lower precedence operator. If the precedence levels of operators are the same, then the order of evaluation depends on their associativity (or, grouping). In Chapter 2 we briefly discussed the precedence and associativity of arithmetic operators. Table 5.1 shows the precedence levels and associativity of all C operators.

In the table, there are 15 precedence levels 0 through 14: higher level implies higher precedence. The precedence levels of operators are separated by solid lines with operators within solid lines having the same precedence level. For example, binary arithmetic operators $*$, $/$, and `%` have the same precedence level which is higher than binary $+$, and $-$. Observe that the precedence of the assignment operator is lower than all but the "comma" operator (described below). This is in accordance with the rule that the expression on the right side of an assignment is evaluated first, and then its value is assigned to the left hand side object. On the other hand, "function call" has the highest precedence, since a function *value* is treated like a variable reference in an expression. In any expression, parentheses may be used to over ride the precedence of the operators — innermost parentheses are always evaluated first. The precedence of binary logical operators is lower than that of binary relational operators; that of binary relational operators is lower than that of binary arithmetic operators, and so forth. The unary NOT operator has a precedence higher than that of all binary operators.

When operators of the same precedence level appear in an expression, the order of evaluation is determined by the associativity. Except for the assignment operator, associativity of most binary operators is left to right; associativity of the assignment operator and most unary operators is right to left. Consider the following program fragment:

```
int x = 10, y = 7, z = 20;
```

`x = -20 + 10 * 5;`	By the precedence, the unary minus ($-$) is evaluated first; followed by the multiplication ($*$) and then the addition. So the expression is evaluated as `(-20) + (10 * 5)` and finally the result is assigned to `x` which now has the value 30.
`x = x / y * z;`	Here the / and $*$ have the same precedence, so by associativity are evaluated left to right: $(x/y)*z$. This is $30/7*20$, or $4*20$ (integer division); so 80 is assigned to `x`.

	Operator	Associativity	Precedence
()	Function call	Left-to-Right	Highest 14
[]	Array subscript		
.	Dot (Member of structure)		
− >	Arrow (Member of structure)		
!	Logical NOT	Right-to-Left	13
~	One's-complement		
−	Unary minus (Negation)		
++	Increment		
−−	Decrement		
&	Address-of		
∗	Indirection		
(type)	Cast		
sizeof	Sizeof		
∗	Multiplication	Left-to-Right	12
/	Division		
%	Modulus (Remainder)		
+	Addition	Left-to-Right	11
−	Subtraction		
<<	Left-shift	Left-to-Right	10
>>	Right-shift		
<	Less than	Left-to-Right	8
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equal to	Left-to-Right	8
! =	Not equal to		
&	Bitwise AND	Left-to-Right	7
^	Bitwise XOR	Left-to-Right	6
|	Bitwise OR	Left-to-Right	5
&&	Logical AND	Left-to-Right	4
||	Logical OR	Left-to-Right	3
? :	Conditional	Right-to-Left	2
=, +=	Assignment operators	Right-to-Left	1
∗=, etc.			
,	Comma	Left-to-Right	Lowest 0

Table 5.1: Precedence and Associativity Table

`x / y / z`	Again, associativity causes the operators to be evaluated left to right. (80/7)/20, i.e. 11/20 or 0. (No assignment is made here).
`x % y % z`	Evaluated as (80%7)%20; 3%20 or 3.
`x = y = 10;`	The assignment operator associates right to left; so `y` is assigned 10, and then the result value, 10, is assigned to `x`.
`x + y <= 2 * z`	The highest precedence operator is $*$, so it is evaluated first, followed by $+$, and finally the comparison operator, $<=$. The result, 87, is not less than or equal to 40, so this expression evaluates to False, namely 0.
`(x + y >= 2 * z) && (x - y != z)`	The parentheses force the logical operator `&&` to be evaluated last. Its left operand is similar to the last expression; only the result is now True, or 1. The right operand evaluates the subtraction followed by the comparison, not equal. Since 73 is not equal to 20, the result is True, and therefore, the entire expression is True, or 1.

When a logical operator is used in an expression, the entire expression is not evaluated if the result of the entire logical expression is clear. For example,

```
(x > 0) && (y > 0)
(x > 0) || (y > 0)
```

In the first expression, if `x > 0` is False, there is no need to evaluate the second part of the logical AND expression since the AND operation will be False. Similarly, in the second expression, the logical OR expression is True if the first part, `x > 0`, is True; there is no need to evaluate the second part. C evaluates only those parts of a logical expression that are required in order to arrive at the result of the expression.

When in doubt as to the order of evaluation within an expression, parentheses may be used to ensure evaluation is performed as intended.

5.4.2 The Data Type of the Result

The data type of an expression value depends on the operators and the types of operands. If the operands are all of the same type, the result is of that same type. When there are operands of mixed type in an assignment expression, the right hand side is always converted to the data type of the object on the left hand side. This follows common sense since the type of the object on the left of an assignment is fixed and cannot be changed. When any other binary operator is applied to operands of mixed type, the operand of a type with lower range is converted to the type of the higher range operand before the operator is applied; and the result is of the higher range type. Of course, values of characters in an expression are considered to be `int` type. Again, some examples will illustrate:

```
int n = 3, m = 2;
long large;
float x = 9.0, y = 5.0;
double z = 4.0;
```

`large = n;`	The *integer value* of `n` is converted to `long` and assigned to `large`.
`large = n / m;`	Since `n` and `m` are both type `int`, integer division is performed (3 / 2 which is 1); and then converted to `long` (1L) which is assigned to `large`.
`large = n / x;`	Since `x` is a `float`, the *integer value* of `n` is converted to `float` and real division (3.0 / 9.0) is performed yielding 0.33. This result is then converted to a long integer (by truncating), namely 0L, and assigned to `large`.
`z = z * y;`	Because `z` is type `double`, the *value* of `y` is converted to `double` and the double precision result of `z * y` is assigned to `z`.
`n / x + z / y;`	In the first division, `n / x`, since `x` is type `float` the division will be done at `float` precision by first converting the value of `n`, yielding a `float` result. The second division will be performed using double precision because `z` is a `double`, by first converting `y` to a `double`. The addition is now of a `float` and a `double`, so the left operand is first converted to `double` yielding a `double` result. This is equivalent to: `(double) ((float) n / x) + z / (double) y;`

As with the precedence and associativity rules, when in doubt as to the type and/or precision of an expression evaluation, cast operators may be used to force conversions to the desired type. Remember, only *values* of variables are converted for the purpose of computation, NOT the variables themselves.

5.4.3 Some New Operators

In Table 5.1 there are several operators which we have not yet discussed. Some of these are described below; the remainder will be delayed until later chapters when we discuss the appropriate data types.

Increment and Decrements Operators

A common operation in many programs is to increase or decrease a variable value by one; for example, this is how we keep a count of how many times a loop is executed. C provides a "shorthand" way of performing this operation with special increment and decrement operators, `++` and `--` respectively. These are unary operators requiring one operand and may be used either as **prefix** or **postfix** operators meaning they either precede or follow their operands. In postfix form, `x++` increases the value of `x` by one, and `y--` decreases the value of `y` by one. Likewise, in prefix form, `++x` increases the value of `x` by one, and `--y` decreases the value of `y` by one. However, there is a difference between the prefix and postfix operators. In the case of prefix operators, the operation is performed first *and then* the expression evaluates to the new value of its operand. For postfix operators, the expression first evaluates to the `current` value of the operand and then the operators are applied. For example, if `x` is 1, the expression `++x` first increments `x` to 2 and then evaluates to the value 2. On the other hand, again if `x` is 1, the expression `x++` first evaluates to the value of `x`, namely 1, and then increments `x` to 2. Here is a code fragment showing the use of the increment and decrement operators:

```
int x, y, z1, z2, z3;

x = 4;
y = 4;
```

`x++;`	The value of `x` is incremented to 5. The value of this expression is 4, but is discarded.
`y--;`	The value of `y` is decremented to 3. The value of this expression is also 4, but is also discarded.
`z1 = x++ - y++;`	The expression `x++` evaluates to the current value of `x`, 5, and then `x` gets the value 6. Likewise, `y++` evaluates to 3 and then `y` is incremented to 4. The variable `z1` gets the value of `5 - 3`, or 2.
`z2 = ++x - ++y;`	First, `x` is incremented to 7, and `++x` evaluates to 7. Likewise `++y` increments to `y` to 5 and evaluates to 5, so `z2` gets the value of `7 - 5` or 2.

```
z3 = x++ + --y;
```

The expression, `x++`, evaluates to 7 and then increments `x` to 8. The expression, `--y`, decrements `y` to 4 and evaluates to 4. So `z3` gets the value of `7 + 4`, or 11.

The value of

```
++x - x++
```

is implementation dependent. A compiler may either evaluate the first term first or the second term first. It is therefore not possible to say what the expression will evaluate to. For example, assume that x is initially 1. If the first expression is evaluated first, then the expression is:

```
2 - 2
```

i.e. 0, and `x` is 3. On the other hand, if the second term is evaluated first, then the expression is:

```
3 - 1
```

i.e. 2, and `x` is 3.

Increment and decrement operations can just as well be written as assignment expressions:

```
x = x + 1;
y = y - 1;
```

The use of increment and decrement operators does not accomplish anything that cannot be done by appropriately placed assignments. These operators were designed to be used with machines that have increment and decrement registers; in which case the compiler can take advantage of these registers and improve the performance of the program. However, many machines today do not have these registers, so most compilers translate expressions with increment and decrement operators in exactly the same manner as they do assignment expressions, but these operators remain as a "shorthand" syntax for compact programs.

The syntax of the increment and decrement operators is:

```
++ <Lvalue>
-- <Lvalue>
<Lvalue> ++
<Lvalue> --
```

The operand must be and <Lvalue>, i.e. a location into which a value can be placed. (So far, we have seen that only a variable name may be used as an <Lvalue>. We will see other possibilities

Composite	Equivalent
`x += 5;`	`x = x + 5;`
`y -= 12;`	`y = y - 12;`
`x *= 3;`	`x = x * 3;`
`y /= 5;`	`y = y / 5;`
`x %= 7;`	`x = x % 7;`

Table 5.2: Composite Assignment Operators and Their Equivalents

in Chapter 6). The precedence and associativity of increment and decrement operators is given in Table 5.1. Here are some examples of their use in program code:

`for (i = 0; i < MAX; i++)`
`     printf("This is a test\n");`
	The message, This is a test will be printed `MAX` times.

`n = 0;`
`while (n++ < 10)`
`    printf("Value of n is %d\n", n);`
	The expression `n++` evaluates to the value of `n` before it is incremented. The loop will print the values 1,2, ...,10 for `n`.

`n = 0;`
`while (++n < 10)`
`   printf("value of n is %d\n", n);`
	The expression `++n` evaluates to the value of `n` after it is incremented. The loop will print the values 1,2, ...,9 for `n`.

Composite Assignment Statements

The above operators provide a "shorthand" way of increasing or decreasing a variable by one; but sometimes we would like to increase or decrease (or multiply, divide or mod) by some other value. C provides "short hand" operators for these as well, called the **composite assignment operators**. These operators and their equivalent are shown in Table 5.2.

The general syntax of a composite assignment operator is:

<Lvalue> <op>= <expression>

where <op> may be one of the binary arithmetic operators, `+`, `-`, `*`, `/`, or `%`. The left operand of these operators must be an <Lvalue>, but the right operand may be an arbitrary <expression>.

Again, there is no particular advantage in using the composite assignment operators over the simple assignment operator except that they produce a somewhat more compact program. The precedence and grouping for composite assignment operators given in Table 5.1 shows they are the same as the assignment operator. Figure 5.10 shows the factorial function (see Figure 5.2) using these new operators.

```
/*   File: mathutil.c - continued */
/*   Function returns long factorial of n. */
long factcomp(int n)
{    int i;
     long prod;

     prod = 1;                      /* initialize */
     for (i = 1; i <= n; i++)       /* loop from 1 to n */
          prod *= i;                /* compute cumulative product */
     return prod;                   /* return product */
}
```

Figure 5.10: Factorial Function Using Composite Operators

Conditional Expression

Sometimes in a program we would like to determine the value of an expression based on some condition. For example, if we had two variables, `x` and `y`, and we wanted to assign the larger value to the variable, `z`. We could write and `if` statement to perform this task as follows:

```
if (x < y)  z = y;
else z = x;
```

Another way of stating this in words is that `z` should be assigned the value of `y` if `x` $<$ `y` or `x`, otherwise. The (operator) symbols ? and : may be used to form such a conditional expression as follows:

```
z = x < y ? y : x;
```

The expression to the right of the assignment operator is evaluated first as follows. If `x` $<$ `y`, the expression evaluates to the value of the expression after ?, i.e. `y`. Otherwise, it evaluates to the value of the expression after :, i.e. `x`. In other words, the expression evaluates to the larger of `x` and `y` which is then assigned to the variable, `z`.

As another example, we can write an expression that evaluates to the absolute value of `x`:

```
x < 0 ? -x : x
```

If `x` is negative, the expression evaluates to `-x` (a positive value); otherwise to `x`.

The syntax for writing a conditional expression is:

<expr1> ? <expr2> : <expr3>

```
/* File: mathutil.c - continued*/
double maxdbl(double x, double y)
{
     return (x > y ? x : y);
}
```

Figure 5.11: Function `maxdbl` Using a Conditional Expression

The first operand, <expr1>, is evaluated; if true, the result of the entire expression is the value of < expr2>. Otherwise, the result is the value of <expr3>. The conditional operator is a ternary operator since it requires three operands.

An `if` statement can always perform the task that a conditional expression does. Whether to use one or the other is a matter of choice and convenience. Figure 5.11 shows a function which returns the value of the larger of two `double` arguments.

The Comma Operator

The comma operator, `,`, provides a way to combine several expressions into a single expression. The syntax is:

<expression1> , <expression2>

The semantics are that <expression1> is evaluated first, followed by <expression2> with the value of the entire expression being that of <expression2>. These expressions may be arbitrary expressions, including another comma expression.

The comma expression is useful where the syntax of a statement requires a single expression, but we have several expressions to be evaluated, such as a `for` statement where several variables are used to control the loop. Here, the comma operator may be used to write multiple initialization and update expressions. As an example, we will use comma operators to write a function that computes and prints Fibonacci numbers. Fibonacci numbers are natural numbers in the sequence:

```
1, 1, 2, 3, 5, 8, 13, ...
```

Each number of the sequence is computed by adding the previous two numbers of the sequence. Thus, we must start with the first two numbers, which are both 1, then the next number is $1 + 1 = 2$, the next one is $1 + 2 = 3$, the next one is $2 + 3 = 5$, and so on.

We will write a driver, `main()`, which calls a function, `fib()`, to print the Fibonacci numbers. The function starts with two variables, which are initialized to the values of the first two numbers 1 and 1. Each new number is computed as a sum of the previous two until the limit is reached. Figure 5.12 shows the code.

```
/* File: fib.c
     Program computes and prints Fibonacci numbers less than a
     specified limit of 100.
*/
#include <stdio.h>
#define LIM 100
void fib(int lim);

main()
{
     printf("***Fibonacci Numbers***\n");
     printf("Limit is %d \n", LIM);
     fib(LIM);
}


/* Function computes and prints the Fibonacci numbers less than lim. */
void fib(int lim)
{    int i, j, n;

     printf("1\n1\n");   /* print the first two fib. numbers */
     for (i = 1, j = 1, n = 0; n < lim; i = j, j = n) {
          n = i + j;          /* compute the next fib. number */

          if (n < lim)
               printf("%d\n", n);  /* print the next fib. number */
     }
}
```

Figure 5.12: Revised Fibonacci

The function, `fib()`, prints Fibonacci numbers less than its argument, `lim`. It uses a `for` loop with comma expressions for the first and last expressions. The first expression initializes two variables, `i` and `j` to 1, with `i` assumed to be the first and `j` assumed to be the second number in the sequence. The variable, `n`, the next number in the sequence, is initialized to zero so that the loop condition may be tested the first time with some value of `n` less than `lim`. The sum of `i` and `j` is the next number in the sequence, `n`, which is computed and printed in the loop body. The variables are then updated to the new values, `j` assigned the value of `n` and `i` assigned the value of `j`. Thus, `i` and `j` always have the values of the last two Fibonacci numbers in the sequence. The process is repeated until `n` exceeds `lim`.

The output of the program is shown below:

```
***Fibonacci Numbers***
Limit is 100
1
1
2
3
5
8
13
21
34
55
89
```

The `sizeof` Operator

The exact amount of space reserved in memory for different data types depends on the implementation. Typically, a character is assigned 8 bits or one byte of space; integers are generally assigned 2 or 4 bytes of storage; `float` numbers usually require at least four bytes, and `double` at least eight bytes. Table 5.3 shows some typical examples for the HP9000, an HP_UX Unix system, and the IBM PC, a DOS environment. It is sometimes necessary to use the sizes of objects in expressions, and since the sizes are implementation dependent, to make our programs portable, we should not build the values into our programs as constants. For any implementation, size of an object can be easily determined by the use of the `sizeof` operator with syntax:

sizeof <expression>

The unary operator, `sizeof`, yields the size, in bytes, of the type of its operand. The operand may be an arbitrary expression, however, the expression is NOT evaluated; the `sizeof` expression simply evaluates to the number of bytes used for the *type* of the result. For example, the expression, `sizeof x`, evaluates to the size of `x` in bytes. Here is a code fragment using the `sizeof` operator:

Data types	HP9000 Bytes	IBM PC Bytes
`char`	1	1
`int`	4	2
`short int`	2	2
`long int`	4	4
`float`	4	4
`double`	8	8
`long double`	16	8

Table 5.3: Space allocation in Bytes for data types

```
int x;
double y;

printf("Size of x is %d bytes\n",  sizeof x);
printf("Size of x+y is %d bytes\n",  sizeof (x+y));
```

The first `printf()` statement will print the size (in bytes) of the `int` type object, `x`. The second will print the size of the value of the expression, `x+y`. As we saw earlier, this addition would be done in double precision and the result would be a `double`. Remember, the expression, `x+y` is not evaluated; only its size is used by the `sizeof` operator. Also remember that `sizeof` is an operator, like `+`; not a function call. It has a precedence and associativity like any other operator (shown in Table 5.1). That is why the parentheses are required in that second `printf()`, the precedence of `sizeof` is higher than `+`. Without the parentheses, the expression would be evaluated as:

```
(sizeof x) + y
```

It is also possible for the operand of `sizeof` to be a parenthesized type name, like a cast operator, rather than a variable name, for example:

```
sizeof (int)
sizeof (float)
sizeof (long int)
sizeof (unsigned long int)
```

We can easily write a program to determine the sizes of different types for the host implementation. The code is shown in Figure 5.13. A sample output for the HP9000 is:

```
***Sizeof operator***
```

```
/* File: size.c */
main()
{   int x;
    double y;

    printf("***Sizeof operator***\n\n");
    printf("Size of x is %d bytes\n", sizeof  x);
    printf("Size of x+y is %d bytes\n\n", sizeof (x+y));
    printf("Size of data types in bytes:\n");
    printf("Size of int type is %d\n", sizeof(int));
    printf("Size of long int is %d\n", sizeof(long int));
    printf("Size of short int is %d\n", sizeof(short int));
    printf("Size of unsigned int  is %d\n", sizeof(unsigned int));
    printf("Size of float is %d\n", sizeof(float));
    printf("Size of double is %d\n", sizeof(double));
}
```

Figure 5.13: Testing `sizeof` Operator

```
Size of x is 4 bytes
Size of x+y is 8 bytes

Size of data types in bytes:
Size of int type is 4
Size of long int is 4
Size of short int is 2
Size of unsigned int is 4
Size of float is 4
Size of double is 8
```

Whenever the size of a type is required in a program, the `sizeof` operator should be used rather than the actual size, since the actual value is implementation dependent. Such a use of the `sizeof` operator in a program ensures that the program will be portable from one type of computer to another.

5.5 Common Errors

1. A result may be outside the range of values possible for a given data type. Use a data type with greater range and/or precision.

2. Prototypes are not declared; instead, default integer type declaration is assumed for functions. If there is no prototype declaration for a function and if the argument in the function call is a `float`, it is converted to `double`. If the formal parameter in the function definition

is declared as a `float`, there is a possible mismatch. A `double` object passed as an argument might be accessed as a `float` resulting in a possible wrong value. The actual situation depends on the compiler. Here is an example:

```
/*   File: default.c
     Program illustrates problems with default declarations for functions.
*/
#include <stdio.h>
main()
{    float x;

     x = 3.0;
     printf("Truncated Square of %f = %d\n", x, trunc_square(x));
}

int trunc_square(float z)
{
     return (int) (z * z);
}
```

The function `trunc_square()` returns integer type and `main()` uses the default declaration for `trunc_square()`. The `float` argument, `x` in the function call in `main()` is converted to `double`. But `trunc_square()` declares a `float` formal parameter, `z`. An attempt will be made to access a `double` object as a `float`. The function may not access the correct value passed as an argument. Thus, it is always best to use function prototypes to avoid confusion.

3. An expression is written without consideration of precedence and associativity of the operators. For example,

```
while (x = scanf("%d", &n) != EOF)
     ...
```

Wrong! The `scanf()` value is compared first with `EOF` and the result of the comparison is assigned to `x`. Using parentheses:

```
while ((x = scanf("%d", &n)) != EOF)
     ...
```

`x` is assigned the value returned by `scanf()`, and the value of `x` is then compared with `EOF`. Examples where associativity must be considered include:

```
a = 10; b = 5; c = 20; d = 4;

a - b - c      is -15
a / b / c / d  is 0
a % d % b % c  is 2
```

4. Increment and decrement operators are used incorrectly. Remember that postfix implies increment/decrement after evaluation and prefix implies increment/decrement before evaluation.

5.6 Summary

In this chapter we have tied up some loose ends and formalized some of the concepts from previous chapters. We have seen how the finite number of bits available to represent numbers limits the range and precision of the numbers stored in the computer. We have introduced additional data types which can extend the range and increase precision as needed for some applications. We have discussed the data types `void` (when no value is expected) and `enum` (for improving program readability). We have also shown how user defined names for data types can be defined using `typedef` with syntax:

```
typedef <existing-type-specifier> <new-type-specifier>;
```

We have extended our available control constructs by introducing two variations on the looping constructs provided in the language: the `for` statement and the `do...while` statement, with syntax:

```
                                                              <expr1>;
                                                              while (<expr2>) {
    for (<expr1>; <expr2>; <expr3>) <statement>  equivalent to        <statement>
                                                                      <expr3>;
                                                              }
```

and

```
do
        <statement>
while (<expression>);
```

We have also described how expressions are evaluated, including the determination of the type of the result and the order of applying operators, giving the full precedence and associativity table for all C operators (Table 5.1). We have described some new operators, such as the increment/decrement operators:

```
++ <Lvalue>
-- <Lvalue>
<Lvalue> ++
<Lvalue> --
```

composite assignment operators:

```
<Lvalue> <op>= <expression>
```

the conditional expression:

```
<expr1> ? <expr2> : <expr3>
```

the comma operator:

```
<expression1> , <expression2>
```

and the `sizeof` operator:

```
sizeof <expression>
```

Other operators in the table such as the indirection, array subscripting, structure accessing, and bitwise operators will be described in later chapters.

5.7 Exercises

1. If `x` is 100 and `z` is 200, what is the output of the following:

```
if (z = x)
     printf("z = %d, x = %d\n", z, x);
```

2. With the following declarations:

```
int a = 10, b = 15, c = 25;
float x = 15;
double y = 30;
long int m = 25L;
```

   What are the values and types of the following expressions:

```
a + b / c * x;
a + b / x * c;
a + b / y * c;
a + b / m * c;

x + a / b;
x + (int) a / b;
```

3. Evaluate the expressions following the declarations:

```
int x, y, z;
float u, v, w;

x = 10; y= 20; z = 30;

x = z / y + y;
x = x / y / z;
x = x % y % z
```

4. Evaluate the expressions:

```
int x, y, z;
float u, v, w;
x = 10; y= 20; z = 30;
u = 5.0; v = 10.0; w = 30.0;

x = w / y + y;
u = z / y + y;
u = w / y + y;
u = x / y / w + u / v;
```

5. What is the output of the following program?

```
#define PRHAPS
#define TWICEZ z + z

main()
{    int w, x, y, z;
     float a, b, c;
     w = 16; x = 5; y = 15; z = 8;
     a = 1.0; b = 2.0; c = 4.0;

     #ifdef PRHAPS
          x = 15;
          y = 5;
     #endif

     printf("(a). %d %d\n", x, y);
     printf("(b). %d\n", TWICEZ * 2);
     printf("(c). %f %f\n", w / z * a + c, z / w * b + c);
     printf("(d). %d\n", z % y % x);
}
```

6. What will be the output in the following cases:

   (a)
   ```
   #define SWAP(x, y)  int temp; temp = x; x = y; y = temp
   main()
   {    int x1 = 10, x2 = 20;

        SWAP(x1, x2);
        printf("x1 = %d, x2 = %d\n", x1, x2);
   }
   ```

   (b)
   ```
   #define SWAP(x, y)  {int temp; temp = x; x = y; y = temp;}
   main()
   {    int x1 = 10, x2 = 20;

        SWAP(x1, x2);
        printf("x1 = %d, x2 = %d\n", x1, x2);
   }
   ```

   (c)
   ```
   #define SWAP(x, y)  int temp; temp = x; x = y; y = temp
   main()
   {    int x1 = 10, x2 = 20;

        printf("Swapping Values\n");
        SWAP(x1, x2);
        printf("x1 = %d, x2 = %d\n", x1, x2);
   }
   ```

7. Write a `while` and a `do...while` loop to read and echo long integers until end of file. Allow for the possibility that the first input is an end of file.

8. Write a for loop to print out squares of integers in the sequence 5, 10, 15, 20, 25, etc. until 100.

9. Given the following declarations:

   ```
   int x = 100, y;
   ```

   What are the values of `x` and `y` after each of the following expressions is evaluated (the expressions are evaluated in sequence)?

   ```
   y = x++;
   y = ++x;
   y = --x;
   y = x--;
   ```

10. What are the values of the following expressions considered sequentially:

    ```
    x = 100; y = 200;
    y = y++ - ++x;
    y = ++y - x++;
    y = ++y * 2;
    y = 2 * x++;
    ```

11. Evaluate the following:

    ```
    x = 100; y = 200;
    y += 2 * x++;
    y -= 2 * --x;
    y += x;
    ```

12. Evaluate the following:

    ```
    x = 100; y = 200; z = 25;
    z = y > x ? x : y;
    z = (z >= x && z >= y) ? z - x * y : z + x * y;
    ```

5.8 Problems

1. Write a program to calculate the roots of a quadratic equation:

$$a * x^2 + b * x + c = 0$$

The program should repeatedly read the set of coefficients a, b, and c. For each set, calculate the roots if and only if $b * b$ is not less than $4 * a * c$. Otherwise, write a message that the roots are not real and proceed to the next set of coefficients. The two roots of a quadratic are:

$$x_1 = \frac{-b + \sqrt{b^2 - 4 * a * c}}{2 * a}$$
$$x_2 = \frac{-b - \sqrt{b^2 - 4 * a * c}}{2 * a}$$

Use the `sq_root()` function defined in the chapter.

2. Write a function to find `exp(x)` whose value is given by the Taylor series:

$$1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

where $n!$ is n factorial. Write and use a function, `power(x, n)`, which returns the $\mathtt{n}^{th}$ power of `x`, where `n` is an integer. Use a function, `fact()`, to compute the factorial. Write a driver that reads input values of `x`, and finds `exp(x)`. Use as many terms as needed to make values before and after an additional term very close.

3. Write a function to evaluate `sin(x)` using the expansion shown below. Use it in a program to find the sine of values read until end of file.

$$sin(x) = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

4. Write a function, `cos(x)`, using the expansion below and use it in a program to find the cosine of values read until `EOF`.

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$

5. What are the limitations on the accuracy of the above expansions?

6. Write a function that returns the number of ways that `r` items can be taken together out of `n` items. The value of combination is:

$$comb(n, r) = \frac{n!}{(n-r)! * r!}$$

Use long integers for factorials.

7. Extend the range of possible values for Problem 6 by cancelling out common factors in numerator and denominator.

8. Write a program that uses Newton's method to find the roots of the equation:

$$f(x) = x^2 + 5 * x + 6 = 0$$

Newton's method uses successive approximations. Start with a guess value for root. The improved value of root is given by:

$$newroot = root - \frac{f(root)}{f'(root)}$$

where $f(root)$ is the value of the function when x equals $root$, and $f'(root)$ is the value of the function below when x equals $root$:

$$f'(x) = 2 * x + 5$$

9. Write a program that finds the approximate value of an integral of a function whose four sample values s_1, s_2, s_3, s_4 are specified at time instants t_1, $t_1 + h$, $t_1 + 2 * h$, $t_1 + 3 * h$. The user should be asked for the value of the interval size, h, and starting instant, t_1. The approximate value of an integral from t_1 to $t_1 + 4 * h$ is the sum of the area under each rectangle made up of the sample value and the inter-sample distance, i.e.:

$$s_1 * h + s_2 * h + s_3 * h + s_4 * h$$

10. Write a program that reads in the coefficients and the right hand side values for two linear simultaneous equations. Solve the equations for the unknowns and print the solution values. The equations are:

$$a_{(1,1)} * x_1 + a_{(1,2)} * x_2 = c_1$$

$$a_{(2,1)} * x_1 + a_{(2,2)} * x_2 = c_2$$

where $a_{(1,1)}$, $a_{(1,2)}$, c_1, $a_{(2,1)}$, $a_{(2,2)}$, and c_2 are the coefficients to be read, and x_1 and x_2 are the unknowns. To solve the equations, multiply the first equation coefficients and right hand side by $-\frac{a_{(2,1)}}{a_{(1,1)}}$ and add the corresponding values to those of the second equation. The new, modified value of $a_{(2,1)}$ will be zero, so the second equation can be solved for x_2, and, substituting the value of x_2 in the first equation, solve for x_1.

11. Given coefficients and the right hand side of two simultaneous equations, verify if a given set of values for x_1 and x_2 is correct. If the left hand side and the right hand side are within a small error margin the solution is assumed to be correct. Let the margin of error be a specifiable value with an assumed default value.

12. Write a menu-driven program to solve and verify two linear equations as per Problems 10 and 11. Allow the following commands: get data, display data, solve equations, display solution, verify solution, help, and quit.

13. Write a program to determine the current and the power consumed in an electrical resistor (load) of 10000 ohms if it is connected to a battery of 12 volts. Power consumed in a resistor is V^2/R, where V is the volts across the resistor and R is the resistor value in ohms. The current in a resistor is given by V/R.

14. Use `for` loops to write a program that finds all prime numbers less than a specified value.

15. Use `do...while` loops to write Problem 14.

16. Write a program that reads a year, a month, and a day of the month. It then determines the number of the day in the year. (Use the definition of a leap year given in Problem 3.6). Use enumeration type for the months, and a `switch` statement which uses the number of days in the year prior to the first of each month.

17. Modify Problem 16 so the program reads the day of the week on the first of January and determines the day of the week for the specified date.

18. Write a program to read the current date in the order: year, month, and day of the month. The program then prints the date in words: Today is the nth day of Month of the year Year. Example:

```
Today is the 24th day of December of the year 2000.
```

19. If the GCD of two numbers, m and n is 1, they have no common divisor. Write a program to find all pairs of numbers, in the range 2 to 20, that have no common divisors. (Refer to Problem 3.12 for the definition of GCD).

20. A rational number is maintained as a ratio of two integers, e.g. 20/23, 35/46, etc. Rational number arithmetic adds, subtracts, multiplies and divides two rational numbers. Write a program that repeatedly reads and adds two rational numbers. The program should print the result in each case as a rational number.

21. Write a function to subtract two rational numbers.

22. Write a function to multiply two rational numbers.

23. Write a function to divide two rational numbers.

24. Write a function to reduce a rational number. A reduced rational number is one in which all common factors in the numerator and the denominator have been cancelled out. For example, 20/30 is reduce to 2/3, 24/18 is reduced to 4/3, and so forth. The GCD can be used to reduce a rational number.

25. Modify the rational numbers programs in Problems 20 through 24 so the result is first reduced before it is printed.