EMBREE, P. and KIMBLE, B. (1991). *C Language Algorithms for Digital Signal Processing.* Englewood Cliffs, NJ: Prentice Hall.

HARRIS, F. (1978). On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform. *Proceedings of the IEEE., 66,* (1), 51–83.

HAYKIN, S. (1986). *Adaptive Filter Theory.* Englewood Cliffs, NJ: Prentice Hall.

MCCLELLAN, J., PARKS, T. and RABINER, L.R. (1973). A Computer Program for Designing Optimum FIR Linear Phase Digital Filters. *IEEE Transactions on Audio and Electro-acoustics, AU-21.* (6), 506–526.

MOLER, C., LITTLE, J. and BANGERT, S. (1987). *PC-MATLAB User's Guide.* Sherbourne, MA: The Math Works.

OPPENHEIM, A. and SCHAFER, R. (1975). *Digital Signal Processing.* Englewood Cliffs, NJ: Prentice Hall.

OPPENHEIM, A. and SCHAFER, R. (1989). *Discrete-time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall.

PAPOULIS, A. (1965). *Probability, Random Variables and Stochastic Processes.* New York: McGraw-Hill.

RABINER, L. and GOLD, B. (1975). *Theory and Application of Digital Signal Processing.* Englewood Cliffs, NJ: Prentice Hall.

STEARNS, S. and DAVID, R. (1988). *Signal Processing Algorithms.* Englewood Cliffs, NJ: Prentice Hall.

STEARNS, S. and DAVID, R. (1993). *Signal Processing Algorithms in FORTRAN and C.* Englewood Cliffs, NJ: Prentice Hall.

VAIDYANATHAN, P. (1993). *Multirate Systems and Filter Banks.* Englewood Cliffs, NJ: Prentice Hall.

WIDROW, B. and STEARNS, S. (1985). *Adaptive Signal Processing.* Englewood Cliffs, NJ: Prentice Hall.

CHAPTER 2

C PROGRAMMING FUNDAMENTALS

The purpose of this chapter is to provide the programmer with a complete overview of the fundamentals of the C programming language that are important in DSP applications. In particular, text manipulation, bitfields, enumerated data types, and unions are not discussed, because they have limited utility in the majority of DSP programs. Readers with C programming experience may wish to skip the bulk of this chapter with the possible exception of the more advanced concepts related to pointers and structures presented in sections 2.7 and 2.8. The proper use of pointers and data structures in C can make a DSP program easier to write and much easier for others to understand. Example DSP programs in this chapter and those which follow will clarify the importance of pointers and data structures in DSP programs.

2.1 THE ELEMENTS OF REAL-TIME DSP PROGRAMMING

The purpose of a *programming language* is to provide a tool so that a programmer can easily solve a problem involving the manipulation of some type of information. Based on this definition, the purpose of a DSP program is to manipulate a signal (a special kind of information) in such a way that the program solves a signal-processing problem. To do this, a DSP programming language must have five basic elements:

(1) A method of organizing different types of data (variables and data types)
(2) A method of describing the operations to be done (operators)
(3) A method of controlling the operations performed based on the results of operations (program control)

1.7.2 LMS Algorithms

The LMS algorithm is the simplest and most used adaptive algorithm in use today. In this brief section, the LMS algorithm as it is applied to the adaptation of time-varying FIR filters (MA systems) and IIR filters (adaptive recursive filters or ARMA systems) is described. A detailed derivation, justification and convergence properties can be found in the references.

For the adaptive FIR system the transfer function is described by

$$y(n) = \sum_{q=0}^{Q-1} b_q(k)x(n-q), \tag{1.115}$$

where $b(k)$ indicates the time-varying coefficients of the filter. With an FIR filter the mean squared error performance surface in the multidimensional space of the filter coefficients is a quadratic function and has a single minimum mean squared error (MMSE). The coefficient values at the optimal solution is called the MMSE solution. The goal of the adaptive process is to adjust the filter coefficients in such a way that they move from their current position toward the MMSE solution. If the input signal changes with time, the adaptive system must continually adjust the coefficients to follow the MMSE solution. In practice, the MMSE solution is often never reached.

The LMS algorithm updates the filter coefficients based on the method of steepest descent. This can be described in vector notation as follows:

$$\mathbf{B}_{k+1} = \mathbf{B}_k - \mu\nabla_k \tag{1.116}$$

where $\mathbf{B}_k$ is the coefficient column vector, μ is a parameter that controls the rate of convergence and the gradient is approximated as

$$\nabla_k = \frac{\partial E[\epsilon_k^2]}{\partial \mathbf{B}_k} \cong -2\epsilon_k\mathbf{X}_k \tag{1.117}$$

where $\mathbf{X}_k$ is the input signal column vector and ε_k is the error signal as shown on Figure 1.27. Thus, the basic LMS algorithm can be written as

$$\mathbf{B}_{k+1} = \mathbf{B}_k + 2\mu\epsilon_k\mathbf{X}_k \tag{1.118}$$

The selection of the convergence parameter must be done carefully, because if it is too small the coefficient vector will adapt very slowly and may not react to changes in the input signal. If the convergence parameter is too large, the system will adapt to noise in the signal and may never converge to the MMSE solution.

For the adaptive IIR system the transfer function is described by

$$y(n) = \sum_{q=0}^{Q-1} b_q(k)x(n-q) - \sum_{p=1}^{P-1} a_p(k)y(n-p), \tag{1.119}$$

where $b(k)$ and $a(k)$ indicate the time-varying coefficients of the filter. With an IIR filter, the mean squared error performance surface in the multidimensional space of the filter coefficients is not a quadratic function and can have multiple minimums that may cause the adaptive algorithm to never reach the MMSE solution. Because the IIR system has poles, the system can become unstable if the poles ever move outside the unit circle during the adaptive process. These two potential problems are serious disadvantages of adaptive recursive filters that limit their application and complexity. For this reason, most applications are limited to a small number of poles. The LMS algorithm can again be used to update the filter coefficients based on the method of steepest descent. This can be described in vector notation as follows:

$$\mathbf{W}_{k+1} = \mathbf{W}_k - \mathbf{M}\nabla_k, \tag{1.120}$$

where $\mathbf{W}_k$ is the coefficient column vector containing the a and b coefficients, $\mathbf{M}$ is a diagonal matrix containing convergence parameters μ for the a coefficients and ν_0 through ν_{P-1} that controls the rate of convergence of the b coefficients. In this case, the gradient is approximated as

$$\nabla_k \cong -2\epsilon_k\left[\alpha_0 \ldots \alpha_{Q-1}\beta_1 \ldots \beta_P\right]^T, \tag{1.121}$$

where ε_k is the error signal as shown in Figure 1.27, and

$$\alpha_n(k) = x(k-n) + \sum_{q=0}^{Q-1} b_q(k)\alpha_n(k-q) \tag{1.122}$$

$$\beta_n(k) = y(k-n) + \sum_{p=0}^{P-1} b_q(k)\beta_n(k-p). \tag{1.123}$$

The selection of the convergence parameters must be done carefully because if they are too small the coefficient vector will adapt very slowly and may not react to changes in the input signal. If the convergence parameters are too large the system will adapt to noise in the signal or may become unstable. The proposed new location of the poles should also be tested before each update to determine if an unstable adaptive filter is about to be used. If an unstable pole location is found the update should not take place and the next update value may lead to a better solution.

1.8 REFERENCES

BRIGHAM, E. (1974). *The Fast Fourier Transform.* Englewood Cliffs, NJ: Prentice Hall.

CLARKSON, P. (1993). *Optimal and Adaptive Signal Processing.* FL: CRC Press.

ELIOTT, D.F. (Ed.). (1987). *Handbook of Digital Signal Processing.* San Diego, CA: Academic Press.

(4) A method of organizing the data and the operations so that a sequence of program steps can be executed from anywhere in the program (functions and data structures) and
(5) A method to move data back and forth between the outside world and the program (input/output)

These five elements are required for efficient programming of DSP algorithms. Their implementation in C is described in the remainder of this chapter.

As a preview of the C programming language, a simple real-time DSP program is shown in Listing 2.1. It illustrates each of the five elements of DSP programming. The listing is divided into six sections as indicated by the comments in the program. This simple DSP program gets a series of numbers from an input source such as an A/D converter (the function **getinput()** is not shown, since it would be hardware specific) and determines the average and variance of the numbers which were sampled. In signal-processing terms, the output of the program is the DC level and total AC power of the signal.

The first line of Listing 2.1, **main()**, declares that the program called *main*, which has no arguments, will be defined after the next left brace (**{** on the next line). The main program (called main because it is executed first and is responsible for the main control of the program) is declared in the same way as the functions. Between the left brace on the second line and the right brace half way down the page (before the line that starts **float average**...) are the statements that form the main program. As shown in this example, all statements in C end in a semicolon (**;**) and may be placed anywhere on the input line. In fact, all spaces and carriage control characters are ignored by most C compilers. Listing 2.1 is shown in a format intended to make it easier to follow and modify.

The third and fourth lines of Listing 2.1 are statements declaring the *functions* (**average, variance, sqrt**) that will be used in the rest of the main program (the function **sqrt()** is defined in the standard C library as discussed in the Appendix. This first section of Listing 2.1 relates to program organization (element four of the above list). The beginning of each section of the program is indicated by comments in the *program source code* (i. e., **/* section 1 */**). Most C compilers allow any sequence of characters (including multiple lines and, in some cases, nested comments) between the **/*** and ***/** delimiters.

Section two of the program declares the variables to be used. Some variables are declared as *single floating-point numbers* (such as **ave** and **var**); some variables are declared as *single integers* (such as **i, count,** and **number**); and some variables are *arrays* (such as **signal[100]**). This program section relates to element one, data organization.

Section three reads 100 floating-point values into an array called signal using a **for** *loop* (similar to a DO loop in FORTRAN). This loop is inside an infinite **while** *loop* that is common in real-time programs. For every 100 samples, the program will display the results and then get another 100 samples. Thus, the results are displayed in real-time. This section relates to element five (input/output) and element three (program control).

Section four of the example program uses the functions **average** and **variance**

```
main()                                  /* section 1 */
{
  float average(),variance(),sqrt();    /* declare functions */

  float signal[100],ave,var;            /*section 2 */
  int count,i;                          /* declare variables */

  while(1) {
    for(count = 0 ; count < 100 ; count++) { /* section 3 */
        signal[count] = getinput();     /* read input signal */
    }

    ave = average(signal,count);        /* section 4 */
    var = variance(signal,count);       /* calculate results */

    printf("\n\nAverage = %f",ave);     /* section 5 */
    printf(" Variance = %f",var);       /* print results */
  }
}
float average(float array[],int size)   /* section 6 */
{                                       /* function calculates average */
    int i;
    float sum = 0.0;                    /* intialize and declare sum */
    for(i = 0 ; i < size ; i++)
        sum = sum + array[i];           /* calculate sum */
    return(sum/size);                   /* return average */
}
float variance(float array[],int size)  /* function calculates variance */
{
    int i;                              /* declare local variables */
    float ave;
    float sum = 0.0;                    /* intialize sum of signal */
    float sum2 = 0.0;                   /* sum of signal squared */
    for(i = 0 ; i < size ; i++) {
        sum = sum + array[i];
        sum2 = sum2 + array[i]*array[i]; /* calculate both sums */
    }
    ave = sum/size;                      /* calculate average */
    return((sum2 - sum*ave)/(size-1));   /* return variance */
}
```

Listing 2.1 Example C program that calculates the average and variance of a sequence of numbers.

to calculate the statistics to be printed. The variables **ave** and **var** are used to store the results and the library function **printf** is used to display the results. This part of the program relates to element four (functions and data structures) because the operations defined in functions **average** and **variance** are executed and stored.

Section five uses the library function **printf** to display the results **ave**, **var**, and also calls the function **sqrt** in order to display the standard deviation. This part of the program relates to element four (functions) and element five (input/output), because the operations defined in function **sqrt** are executed and the results are also displayed.

The two functions, **average** and **variance**, are defined in the remaining part of Listing 2.1. This last section relates primarily to element two (operators), since the detailed operation of each function is defined in the same way that the main program was defined. The function and argument types are defined and the local variables to be used in each function are declared. The operations required by each function are then defined followed by a return statement that passes the result back to the main program.

2.2 VARIABLES AND DATA TYPES

All programs work by manipulating some kind of information. A variable in C is defined by declaring that a sequence of characters (the variable identifier or name) are to be treated as a particular predefined type of data. An identifier may be any sequence of characters (usually with some length restrictions) that obeys the following three rules:

(1) All identifiers start with a letter or an underscore (_).
(2) The rest of the identifier can consist of letters, underscores, and/or digits.
(3) The rest of the identifier does not match any of the C keywords. (Check compiler implementation for a list of these.)

In particular, C is case sensitive; making the variables **Average**, **AVERAGE**, and **AVeRaGe** all different.

The C language supports several different data types that represent integers (declared **int**), floating-point numbers (declared **float** or **double**), and text data (declared **char**). Also, arrays of each variable type and pointers of each type may be declared. The first two types of numbers will be covered first followed by a brief introduction to arrays (covered in more detail with pointers in section 2.7). The special treatment of text using character arrays and strings will be discussed in Section 2.2.3.

2.2.1 Types of Numbers

A C program must declare the variable before it is used in the program. There are several types of numbers used depending on the format in which the numbers are stored (floating-point format or integer format) and the accuracy of the numbers (single-precision versus double-precision floating-point, for example). The following example program illustrates the use of five different types of numbers:

```
main()
{
    int i;                   /* size dependent on implementation */
    short j;                 /* 16 bit integer */
    long k;                  /* 32 bit integer */
    float a;                 /* single precision floating-point */
    double b;                /* double precision floating-point */
    k = 72000;
    j = k;
    i = k;
    b = 0.1;
    a = b;

    printf("\n%ld %d %d\n%20.15f\n%20.15f",k,j,i,b,a);
}
```

Three types of integer numbers (**int**, **short int**, and **long int**) and two types of floating-point numbers (**float** and **double**) are illustrated in this example. The actual sizes (in terms of the number of bytes used to store the variable) of these five types depends upon the implementation; all that is guaranteed is that a **short int** variable will not be larger than a **long int** and a **double** will be twice as large as a **float**. The size of a variable declared as just **int** depends on the compiler implementation. It is normally the size most conveniently manipulated by the target computer, thereby making programs using **int**s the most efficient on a particular machine. However, if the size of the integer representation is important in a program (as it often is) then declaring variables as **int** could make the program behave differently on different machines. For example, on a 16-bit machine, the above program would produce the following results:

```
72000 6464 6464
0.100000000000000
0.100000001490116
```

But on a 32-bit machine (using 32-bit **int**s), the output would be as follows:

```
72000 6464 72000
0.100000000000000
0.100000001490116
```

Note that in both cases the **short** and **long** variables, **k** and **j**, (the first two numbers displayed) are the same, while the third number, indicating the **int i**, differs. In both cases, the value 6464 is obtained by masking the lower 16 bits of the 32-bit **k** value. Also, in both cases, the floating-point representation of 0.1 with 32 bits (**float**) is accurate to eight decimal places (seven places is typical). With 64 bits it is accurate to at least 15 places.

Thus, to make a program truly portable, the program should contain only **short int** and **long int** declarations (these may be abbreviated **short** and **long**). In addition to the five types illustrated above, the three **int**s can be declared as unsigned by preceding the declaration with unsigned. Also, as will be discussed in more detail in the next section concerning text data, a variable may be declared to be only one byte long by declaring it a **char** (**signed** or **unsigned**). The following table gives the typical sizes and ranges of the different variable types for a 32-bit machine (such as a VAX) and a 16-bit machine (such as the IBM PC).

Variable Declaration	16-bit Machine Size (bits)	16-bit Machine Range	32-bit Machine Size (bits)	32-bit Machine Range
char	8	−128 to 127	8	−128 to 127
unsigned char	8	0 to 255	8	0 to 255
int	16	−32768 to 32767	32	±2.1e9
unsigned int	16	0 to 65535	32	0 to 4.3e9
short	16	−32768 to 32767	16	−32768 to 32767
unsigned short	16	0 to 65535	16	0 to 65535
long	32	±2.1e9	32	±2.1e9
unsigned long	32	0 to 4.3e9	32	0 to 4.3e9
float	32	±1.0e±38	32	±1e±38
double	64	±1.0e±306	64	±1e±308

2.2.2 Arrays

Almost all high-level languages allow the definition of indexed lists of a given data type, commonly referred to as *arrays*. In C, all data types can be declared as an array simply by placing the number of elements to be assigned to the array in brackets after the array name. *Multidimensional arrays* can be defined simply by appending more brackets containing the array size in each dimension. Any N-dimensional array is defined as follows:

```
type name[size1][size2] ... [sizeN];
```

For example, each of the following statements are valid array definitions:

```
unsigned int list[10];
double input[5];
short int x[2000];
char input_buffer[20];
unsigned char image[256][256];
int matrix[4][3][2];
```

Note that the array definition **unsigned char image[256][256]** could define an 8-bit, 256 by 256 image plane where a grey scale image is represented by values from 0 to 255. The last definition defines a three-dimensional matrix in a similar fashion. One difference between C and other languages is that arrays are referenced using brackets to enclose each index. Thus, the image array, as defined above, would be referenced as **image[i][j]** where **i** and **j** are row and column indices, respectively. Also, the first element in all array indices is zero and the last element is *N–1*, where *N* is the size of the array in a particular dimension. Thus, an assignment of the first element of the five element, one-dimensional array **input** (as defined above) such as **input[0]=1.3;** is legal while **input[5]=1.3;** is not.

Arrays may be *initialized* when they are declared. The values to initialize the array are enclosed in one or more sets of braces (**{}**) and the values are separated by commas. For example, a one-dimensional array called vector can be declared and initialized as follows:

```
int vector[6] = { 1, 2, 3, 5, 8, 13 };
```

A two-dimensional array of six double-precision floating-point numbers can be declared and initialized using the following statement:

```
double a[3][2] = {
                   { 1.5, 2.5 },
                   { 1.1e-5 , 1.7e5 },
                   { 1.765 , 12.678 }
                 };
```

Note that commas separate the three sets of inner braces that designate each of the three rows of the matrix **a**, and that each array initialization is a statement that must end in a semicolon.

2.3 OPERATORS

Once variables are defined to be a given size and type, some sort of manipulation must be performed using the variables. This is done by using *operators*. The C language has more operators than most languages; in addition to the usual assignment and arithmetic operators, C also has bitwise operators and a full set of logical operators. Some of these operators (such as bitwise operators) are especially important in order to write DSP programs that utilize the target processor efficiently.

2.3.1 Assignment Operators

The most basic operator is the *assignment operator* which, in C, is the single equal sign (**=**). The value on the right of the equal sign is assigned to the variable on the left. Assignment statements can also be stacked, as in the statement **a=b=1;**. In this case, the statement is evaluated right to left so that 1 is assigned to **b** and **b** is assigned to **a**. In C,

a=ave(x) is an expression, while **a=ave(x);** is a statement. The addition of the semicolon tells the compiler that this is all that will be done with the result from the function **ave(x)**. An expression always has a value that can be used in other expressions. Thus, **a=b+(c=ave(x));** is a legal statement. The result of this statement would be that the result returned by **ave(x)** is assigned to **c** and **b+c** is assigned to **a**. C also allows multiple expressions to be placed within one statement by separating them with the commas. Each expression is evaluated left to right, and the entire expression (comprised of more than one expression) assumes the value of the last expression which is evaluated. For example, **a=(olda=a,ave(x));** assigns the current value of **a** to **olda**, calls the function **ave(x)** and then assigns the value returned by **ave(x)** to **a**.

2.3.2 Arithmetic and Bitwise Operators

The usual set of binary arithmetic operators (operators which perform arithmetic on two operands) are supported in C using the following symbols:

`*`	multiplication
`/`	division
`+`	addition
`-`	subtraction
`%`	modulus (integer remainder after division)

The first four operators listed are defined for all types of variables (**char**, **int**, **float**, and **double**). The modulus operator is only defined for integer operands. Also, there is no exponent operator in C; this floating-point operation is supported using a *simple function call* (see the Appendix for a description of the **pow** function).

In C, there are three unary arithmetic operators which require only one operand. First is the unary minus operator (for example, **-i**, where **i** is an **int**) that performs a two's-complement change of sign of the integer operand. The unary minus is often useful when the exact hardware implementation of a digital-signal processing algorithm must be simulated. The other two unary arithmetic operators are increment and decrement, represented by the symbols **++** and **--**, respectively. These operators add or subtract one from any integer variable or pointer. The operand is often used in the middle of an expression, and the increment or decrement can be done before or after the variable is used in the expression (depending on whether the operator is before or after the variable). Although the use of **++** and **--** is often associated with pointers (see section 2.7), the following example illustrates these two powerful operators with the ints **i**, **j**, and **k**:

```
i = 4;
j = 7;
k = i++ + j;          /* i is incremented to 5, k = 11 */
k = k + --j;          /* j is decremented to 6, k = 17 */
k = k + i++;          /* i is incremented to 6, k = 22 */
```

Binary bitwise operations are performed on integer operands using the following symbols:

`&`	bitwise AND
`|`	bitwise OR
`^`	bitwise exclusive OR
`<<`	arithmetic shift left (number of bits is operand)
`>>`	arithmetic shift right (number of bits is operand)

The unary bitwise NOT operator, which inverts all the bits in the operand, is implemented with the ~ symbol. For example, if **i** is declared as an **unsigned int**, then **i = ~0;** sets **i** to the maximum integer value for an **unsigned int**.

2.3.3 Combined Operators

C allows operators to be combined with the assignment operator (**=**) so that almost any statement of the form

```
<variable> = <variable> <operator> <expression>
```

can be replaced with

```
<variable> <operator> = <expression>
```

where **<variable>** represents the same variable name in all cases. For example, the following pairs of expressions involving **x** and **y** perform the same function:

```
x = x + y;          x += y;
x = x - y;          x -= y;
x = x * y;          x *= y;
x = x / y;          x /= y;
x = x % y;          x %= y;
x = x & y;          x &= y;
x = x : y;          x := y;
x = x ^ y;          x ^= y;
x = x << y;         x <<= y;
x = x >> y;         x >>= y;
```

In many cases, the left-hand column of statements will result in a more readable and easier to understand program. For this reason, use of combined operators is often avoided. Unfortunately, some compiler implementations may generate more efficient code if the combined operator is used.

2.3.4 Logical Operators

Like all C expressions, an expression involving a logical operator also has a value. A logical operator is any operator that gives a result of true or false. This could be a comparison between two values, or the result of a series of ANDs and ORs. If the result of a logical operation is true, it has a nonzero value; if it is false, it has the value 0. Loops and if statements (covered in section 2.4) check the result of logical operations and change program flow accordingly. The nine logical operators are as follows:

`<`	less than
`<=`	less than or equal to
`==`	equal to
`>=`	greater than or equal to
`>`	greater than
`!=`	not equal to
`&&`	logical AND
`||`	logical OR
`!`	logical NOT (unary operator)

Note that `==` can easily be confused with the assignment operator (`=`) and will result in a valid expression because the assignment also has a value, which is then interpreted as true or false. Also, `&&` and `||` should not be confused with their bitwise counterparts (`&` and `|`) as this may result in hard to find logic problems, because the bitwise results may not give true or false when expected.

2.3.5 Operator Precedence and Type Conversion

Like all computer languages, C has an operator precedence that defines which operators in an expression are evaluated first. If this order is not desired, then parentheses can be used to change the order. Thus, things in parentheses are evaluated first and items of equal precedence are evaluated from left to right. The operators contained in the parentheses or expression are evaluated in the following order (listed by decreasing precedence):

`++,--`	increment, decrement
`-`	unary minus
`*,/,%`	multiplication, division, modulus
`+,-`	addition, subtraction
`<<,>>`	shift left, shift right
`<,<=,>=,>`	relational with less than or greater than
`==,!=`	equal, not equal
`&`	bitwise AND
`^`	bitwise exclusive OR
`|`	bitwise OR
`&&`	logical AND
`||`	logical OR

Statements and expressions using the operators just described should normally use variables and constants of the same type. If, however, you mix types, C doesn't stop dead

(like Pascal) or produce a strange unexpected result (like FORTRAN). Instead, C uses a set of rules to make type conversions automatically. The two basic rules are:

(1) If an operation involves two types, the value with a lower rank is converted to the type of higher rank. This process is called promotion and the ranking from highest to lowest type is double, float, long, int, short, and char. Unsigned of each of the types outranks the individual signed type.

(2) In an assignment statement, the final result is converted to the type of the variable that is being assigned. This may result in promotion or demotion where the value is truncated to a lower ranking type.

Usually these rules work quite well, but sometimes the conversions must be stated explicitly in order to demand that a conversion be done in a certain way. This is accomplished by *type casting* the quantity by placing the name of the desired type in parentheses before the variable or expression. Thus, if **`i`** is an **`int`**, then the statement **`i=10*(1.55+1.67);`** would set **`i`** to 32 (the truncation of 32.2), while the statement **`i=10*((int)1.55+1.67);`** would set **`i`** to 26 (the truncation of 26.7 since **`(int)1.55`** is truncated to 1).

2.4 PROGRAM CONTROL

The large set of operators in C allows a great deal of programming flexibility for DSP applications. Programs that must perform fast binary or logical operations can do so without using special functions to do the bitwise operations. C also has a complete set of program control features that allow conditional execution or repetition of statements based on the result of an expression. Proper use of these control structures is discussed in section 2.11.2, where structured programming techniques are considered.

2.4.1 Conditional Execution: `if-else`

In C, as in many languages, the **`if`** statement is used to conditionally execute a series of statements based on the result of an expression. The **`if`** statement has the following generic format:

```
if(value)
    statement1;
else
    statement2;
```

where **`value`** is any expression that results in (or can be converted to) an integer value. If value is nonzero (indicating a true result), then **`statement1`** is executed; otherwise, **`statement2`** is executed. Note that the result of an expression used for **`value`** need not be the result of a logical operation—all that is required is that the expression results in a zero value when **`statement2`** should be executed instead of **`statement1`**. Also, the

else statement2; portion of the above form is optional, allowing **statement1** to be skipped if value is false.

When more than one statement needs to be executed if a particular value is true, a compound statement is used. A *compound statement* consists of a left brace (**{**), some number of statements (each ending with a semicolon), and a right brace (**}**). Note that the body of the **main()** program and functions in Listing 2.1 are compound statements. In fact, a single statement can be replaced by a compound statement in any of the control structures described in this section. By using compound statements, the **if-else** control structure can be nested as in the following example, which converts a floating-point number (**result**) to a 2-bit twos complement number (**out**):

```
if(result > 0) {          /* positive outputs */
    if(result > sigma)
        out = 1;          /* biggest output */
    else
        out = 0;          /* 0 < result <= sigma */
}
else {                    /* negative outputs */
    if(result < sigma)
        out = 2;          /* smallest output */
    else
        out = 1;          /* sigma <= result <= 0 */
}
```

Note that the inner **if-else** statements are compound statements (each consisting of two statements), which make the braces necessary in the outer **if-else** control structure (without the braces there would be too many **else** statements, resulting in a compilation error).

2.4.2 The switch Statement

When a program must choose between several alternatives, the **if-else** statement becomes inconvenient and sometimes inefficient. When more than four alternatives from a single expression are chosen, the **switch** statement is very useful. The basic form of the **switch** statement is as follows:

```
switch(integer expression) {
    case constant1:
        statements;     (optional)
        break;          (optional)
    case constant2:
        statements;     (optional)
        break;          (optional)
        . . . . .       (more optional statements)
    default:            (optional)
        statements;     (optional)
}
```

Program control jumps to the statement after the case label with the constant (an integer or single character in quotes) that matches the result of the integer expression in the **switch** statement. If no constant matches the expression value, control goes to the statement following the default label. If the default label is not present and no matching case labels are found, then control proceeds with the next statement following the **switch** statement. When a matching constant is found, the remaining statements after the corresponding case label are executed until the end of the switch statement is reached, or a **break** statement is reached that redirects control to the next statement after the **switch** statement. A simple example is as follows:

```
switch(i) {
    case 0:
        printf("\nError: I is zero");
        break;
    case 1:
        j = k*k;
        break;
    default:
        j = k*k/i;
}
```

The use of the **break** statement after the first two case statements is required in order to prevent the next statements from being executed (a **break** is not required after the last **case** or **default** statement). Thus, the above code segment sets **j** equal to **k*k/i**, unless **i** is zero, in which case it will indicate an error and leave **j** unchanged. Note that since the divide operation usually takes more time than the case statement branch, some execution time will be saved whenever **i** equals 1.

2.4.3 Single-Line Conditional Expressions

C offers a way to express one **if-else** control structure in a single line. It is called a *conditional expression,* because it uses the conditional operator, **?:**, which is the only trinary operator in C. The general form of the conditional expression is:

```
expression1 ? expression2 : expression3
```

If **expression1** is true (nonzero), then the whole conditional expression has the value of **expression2**. If **expression1** is false (0), the whole expression has the value of **expression3**. One simple example is finding the maximum of two expressions:

```
maxdif = (a0 > a2) ? a0-a1 : a2-a1;
```

Conditional expressions are not necessary, since **if-else** statements can provide the same function. Conditional expressions are more compact and sometimes lead to more efficient machine code. On the other hand, they are often more confusing than the familiar **if-else** control structure.

2.4.4 Loops: while, do-while, and for

C has three control structures that allow a statement or group of statements to be repeated a fixed or variable number of times. The **while** loop repeats the statements until a test expression becomes false, or zero. The decision to go through the loop is made before the loop is ever started. Thus, it is possible that the loop is never traversed. The general form is:

```
while(expression)
    statement
```

where **statement** can be a single statement or a compound statement enclosed in braces. An example of the latter that counts the number of spaces in a null-terminated string (an array of characters) follows:

```
space_count = 0;     /* space_count is an int */
i = 0;               /* array index, i = 0 */
while(string[i]) {
    if(string[i] == ' ') space_count++;.
    i++;             /* next char */
}
```

Note that if the string is zero length, then the value of **string[i]** will initially point to the null terminator (which has a zero or false value) and the **while** loop will not be executed. Normally, the **while** loop will continue counting the spaces in the string until the null terminator is reached.

The **do-while** loop is used when a group of statements need to be repeated and the exit condition should be tested at the end of the loop. The decision to go through the loop one more time is made after the loop is traversed so that the loop is always executed at least once. The format of **do-while** is similar to the **while** loop, except that the **do** keyword starts the statement and **while(expression)** ends the statement. A single or compound statement may appear between the do and the while keywords. A common use for this loop is in testing the bounds on an input variable as the following example illustrates:

```
do {
    printf("\nEnter FFT length (less than 1025) :");
    scanf("%d",&fft_length);
} while(fft_length > 1024);
```

In this code segment, if the integer **fft_length** entered by the user is larger than 1024, the user is prompted again until the **fft_length** entered is 1024 or less.

The **for** loop combines an initialization statement, an end condition statement, and an action statement (executed at the end of the loop) into one very powerful control structure. The standard form is:

```
for(initialize ; test condition ; end update)
    statement;
```

The three expressions are all optional (**for(;;);** is an infinite loop) and the statement may be a single statement, a compound statement or just a semicolon (a null statement). The most frequent use of the **for** loop is indexing an array through its elements. For example,

```
for(i = 0 ; i < length ; i++) a[i] = 0;
```

sets the elements of the array **a** to zero from **a[0]** up to and including **a[length-1]**. This **for** statement sets **i** to zero, checks to see if **i** is less than **length**, if so it executes the statement **a[i]=0;**, increments **i**, and then repeats the loop until **i** is equal to **length**. The integer **i** is incremented or updated at the end of the loop and then the test condition statement is executed. Thus, the statement after a **for** loop is only executed if the test condition in the **for** loop is true. **For** loops can be much more complicated, because each statement can be multiple expressions as the following example illustrates:

```
for(i = 0 , i3 = 1 ; i < 25 ; i++ , i3 = 3*i3)
    printf("\n%d %d",i,i3);
```

This statement uses two **int**s in the **for** loop (**i**, **i3**) to print the first 25 powers of 3. Note that the end condition is still a single expression (**i < 25**), but that the initialization and end expressions are two assignments for the two integers separated by a comma.

2.4.5 Program Jumps: break, continue, and goto

The loop control structures just discussed and the conditional statements (**if**, **if-else**, and **switch**) are the most important control structures in C. They should be used exclusively in the majority of programs. The last three control statements (**break**, **continue**, and **goto**) allow for conditional program jumps. If used excessively, they will make a program harder to follow, more difficult to debug, and harder to modify.

The **break** statement, which was already illustrated in conjunction with the switch statement, causes the program flow to break free of the **switch**, **for**, **while**, or **do-while** that encloses it and proceed to the next statement after the associated control structure. Sometimes **break** is used to leave a loop when there are two or more reasons to end the loop. Usually, however, it is much clearer to combine the end conditions in a single logical expression in the loop test condition. The exception to this is when a large number of executable statements are contained in the loop and the result of some statement should cause a premature end of the loop (for example, an end of file or other error condition).

The **continue** statement is almost the opposite of **break**; the **continue** causes the rest of an iteration to be skipped and the next iteration to be started. The **continue** statement can be used with **for**, **while**, and **do-while** loops, but cannot be used with **switch**. The flow of the loop in which the **continue** statement appears is interrupted, but the loop is not terminated. Although the **continue** statement can result in very hard-to-follow code, it can shorten programs with nested **if-else** statements inside one of three loop structures.

The **goto** statement is available in C, even though it is never required in C programming. Most programmers with a background in FORTRAN or BASIC computer languages (both of which require the **goto** for program control) have developed bad programming habits that make them depend on the **goto**. The **goto** statement in C uses a label rather than a number making things a little better. For example, one possible legitimate use of **goto** is for consolidated error detection and cleanup as the following simple example illustrates:

```
.
.
program statements
.
.
status = function_one(alpha,beta,constant);
if(status != 0) goto error_exit;
.
.
more program statements
.
.
status = function_two(delta,time);
if(status != 0) goto error_exit;
.
.
.
error_exit:         /*end up here from all errors */
    switch(status) {
        case 1:
            printf("\nDivide by zero error\n");
            exit();
        case 2:
            printf("\nOut of memory error\n");
            exit();
        case 3:
            printf("\nLog overflow error\n");
            exit();
        default:
            printf("\nUnknown error\n");
            exit();
    }
```

In the above example, both of the fictitious functions, **function_one** and **function_two** (see the next section concerning the definition and use of functions), perform some set of operations that can result in one of several errors. If no errors are detected, the function returns zero and the program proceeds normally. If an error is detected, the integer **status** is set to an error code and the program jumps to the label **error_exit** where a message indicating the type of error is printed before the program is terminated.

2.5 FUNCTIONS

All C programs consist of one or more functions. Even the program executed first is a function called **main()**, as illustrated in Listing 2.1. Thus, unlike other programming languages, there is no distinction between the main program and programs that are called by the main program (sometimes called *subroutines*). A C function may or may not return a value thereby removing another distinction between subroutines and functions in languages such as FORTRAN. Each C function is a program equal to every other function. Any function can call any other function (a function can even call itself), or be called by any other function. This makes C functions somewhat different than Pascal procedures, where procedures nested inside one procedure are ignorant of procedures elsewhere in the program. It should also be pointed out that unlike FORTRAN and several other languages, C always passes functions arguments by value not by reference. Because arguments are passed by value, when a function must modify a variable in the calling program, the C programmer must specify the function argument as a pointer to the beginning of the variable in the calling program's memory (see section 2.7 for a discussion of pointers).

2.5.1 Defining and Declaring Functions

A function is defined by the function type, a function name, a pair of parentheses containing an optional formal argument list, and a pair of braces containing the optional executable statements. The general format for ANSI C is as follows:

```
type name(formal argument list with declarations)
{
    function body
}
```

The **type** determines the type of value the function returns, not the type of arguments. If no **type** is given, the function is assumed to return an **int** (actually, a variable is also assumed to be of type **int** if no type specifier is provided). If a function does not return a value, it should be declared with the type **void**. For example, Listing 2.1 contains the function average as follows:

```
float average(float array[],int size)
{
    int i;
    float sum = 0.0;    /* initialize and declare sum */
    for(i = 0 ; i < size ; i++)
        sum = sum + array[i];    /* calculate sum */
    return(sum/size);    /* return average */
}
```

The first line in the above code segment declares a function called **average** will return a single-precision floating-point value and will accept two arguments. The two *argument names* (**array** and **size**) are defined in the formal *argument list* (also called the formal parameter list). The type of the two arguments specify that **array** is a one-dimensional array (of unknown length) and **size** is an **int**. Most modern C compilers allow the argument declarations for a function to be condensed into the argument list.

Note that the variable **array** is actually just a pointer to the beginning of the **float** array that was allocated by the calling program. By passing the pointer, only *one value* is passed to the function and not the large floating-point array. In fact, the function could also be declared as follows:

```
float average(float *array,int size)
```

This method, although more correct in the sense that it conveys what is passed to the function, may be more confusing because the function body references the variable as **array[i]**.

The body of the function that defines the executable statements and local variables to be used by the function are contained between the two braces. Before the ending brace (**}**), a return statement is used to return the **float** result back to the calling program. If the function did not return a value (in which case it should be declared **void**), simply omitting the return statement would return control to the calling program after the last statement before the ending brace. When a function with no return value must be terminated before the ending brace (if an error is detected, for example), a **return;** statement without a value should be used. The parentheses following the return statement are only required when the result of an expression is returned. Otherwise, a constant or variable may be returned without enclosing it in parentheses (for example, **return 0;** or **return n;**).

Arguments are used to convey values from the calling program to the function. Because the arguments are passed by value, a local copy of each argument is made for the function to use (usually the variables are stored on the stack by the calling program). The local copy of the arguments may be freely modified by the function body, but will not change the values in the calling program since only the copy is changed. The return statement can communicate one value from the function to the calling program. Other than this returned value, the function may not directly communicate back to the calling program. This method of passing arguments by value, such that the calling program's

variables are isolated from the function, avoids the common problem in FORTRAN where modifications of arguments by a function get passed back to the calling program, resulting in the occasional modification of constants within the calling program.

When a function must return more than one value, one or more pointer arguments must be used. The calling program must allocate the storage for the result and pass the function a pointer to the memory area to be modified. The function then gets a copy of the pointer, which it uses (with the indirection operator, *, discussed in more detail in Section 2.7.1) to modify the variable allocated by the calling program. For example, the functions **average** and **variance** in Listing 2.1 can be combined into one function that passes the arguments back to the calling program in two **float** pointers called **ave** and **var**, as follows:

```
void stats(float *array,int size,float *ave,float *var)
{
    int i;
    float sum = 0.0;     /* initialize sum of signal */
    float sum2 = 0.0;    /* sum of signal squared */
    for(i = 0 ; i < size ; i++) {
        sum = sum + array[i];    /* calculate sums */
        sum2 = sum2 + array[i]*array[i];
}
    *ave = sum/size;     /* pass average and variance */
    *var = (sum2-sum*(*ave))/(size-1);
}
```

In this function, no value is returned, so it is declared type **void** and no return statement is used. This **stats** function is more efficient than the functions **average** and **variance** together, because the sum of the array elements was calculated by both the average function and the variance function. If the variance is not required by the calling program, then the average function alone is much more efficient, because the sum of the squares of the array elements is not required to determine the average alone.

2.5.2 Storage Class, Privacy, and Scope

In addition to type, variables and functions have a property called *storage class*. There are four storage classes with four storage class designators: **auto** for *automatic variables* stored on the stack, **extern** for *external variables* stored outside the current module, **static** for variables known *only in the current module*, and **register** for *temporary variables* to be stored in one of the registers of the target computer. Each of these four storage classes defines the scope or degree of the privacy a particular variable or function holds. The storage class designator keyword (**auto**, **extern**, **static**, or **register**) must appear first in the variable declaration before any type specification. The privacy of a variable or function is the degree to which other modules or functions cannot access a variable or call a function. Scope is, in some ways, the complement of privacy because the scope of a variable describes how many modules or functions have access to the variable.

Auto variables can only be declared within a function, are created when the function is invoked, and are lost when the function is exited. **Auto** variables are known only to the function in which they are declared and do not retain their value from one invocation of a function to another. Because **auto** variables are stored on a stack, a function that uses only **auto** variables can call itself recursively. The **auto** keyword is rarely used in C programs, since variables declared within functions default to the **auto** storage class.

Another important distinction of the **auto** storage class is that an **auto** variable is only defined within the control structure that surrounds it. That is, the scope of an **auto** variable is limited to the expressions between the braces (**{** and **}**) containing the variable declaration. For example, the following simple program would generate a compiler error, since **j** is unknown outside of the **for** loop:

```
main()
{
    int i;
    for(i = 0 ; i < 10 ; i++) {
        int j;          /* declare j here */
        j = i*i;
        printf("%d",j);
    }
    printf("%d",j);     /* j unknown here */
}
```

Register variables have the same scope as **auto** variables, but are stored in some type of register in the target computer. If the target computer does not have registers, or if no more registers are available in the target computer, a variable declared as **register** will revert to **auto**. Because almost all microprocessors have a large number of registers that can be accessed much faster than outside memory, **register** variables can be used to speed up program execution significantly. Most compilers limit the use of **register** variables to pointers, integers, and characters, because the target machines rarely have the ability to use registers for floating-point or double-precision operations.

Extern variables have the broadest scope. They are known to all functions in a module and are even known outside of the module in that they are declared. **Extern** variables are stored in their own separate data area and must be declared outside of any functions. Functions that access **extern** variables must be careful not to call themselves or call other functions that access the same **extern** variables, since **extern** variables retain their values as functions are entered and exited. **Extern** is the default storage class for variables declared outside of functions and for the functions themselves. Thus, functions not declared otherwise may be invoked by any function in a module as well as by functions in other modules.

Static variables differ from **extern** variables only in scope. A **static** variable declared outside of a function in one module is known only to the functions in that module. A **static** variable declared inside a function is known only to the function in which it is declared. Unlike an **auto** variable, a **static** variable retains its value from one invocation of a function to the next. Thus, **static** refers to the memory area assigned to the variable and does not indicate that the value of the variable cannot be changed. Functions may also be declared **static**, in which case the function is only known to other functions in the same module. In this way, the programmer can prevent other modules (and, thereby, other users of the object module) from invoking a particular function.

2.5.3 Function Prototypes

Although not in the original definition of the C language, *function prototypes*, in one form or another, have become a standard C compiler feature. A function prototype is a statement (which must end with a semicolon) describing a particular function. It tells the compiler the type of the function (that is, the type of the variable it will return) and the type of each argument in the formal argument list. The function named in the function prototype may or may not be contained in the module where it is used. If the function is not defined in the module containing the prototype, the prototype must be declared external. All C compilers provide a series of header files that contain the function prototypes for all of the standard C functions. For example, the prototype for the **stats** function defined in Section 2.5.1 is as follows:

```
extern void stats(float *,int,float *,float *);
```

This prototype indicates that **stats** (which is assumed to be in another module) returns no value and takes four arguments. The first argument is a pointer to a **float** (in this case, the array to do statsistics on). The second argument is an integer (in this case, giving the size of the array) and the last two arguments are pointers to **floats** which will return the average and variance results.

The result of using function prototypes for all functions used by a program is that the compiler now knows what type of arguments are expected by each function. This information can be used in different ways. Some compilers convert whatever type of actual argument is used by the calling program to the type specified in the function prototype and issue a warning that a data conversion has taken place. Other compilers simply issue a warning indicating that the argument types do not agree and assume that the programmer will fix it if such a mismatch is a problem. The ANSI C method of declaring functions also allows the use of a dummy variable with each formal parameter. In fact, when this ANSI C approach is used with dummy arguments, the only difference between function prototypes and function declarations is the semicolon at the end of the function prototype and the possible use of **extern** to indicate that the function is defined in another module.

2.6 MACROS AND THE C PREPROCESSOR

The C *preprocessor* is one of the most useful features of the C programming language. Although most languages allow compiler constants to be defined and used for conditional compilation, few languages (except for assembly language) allow the user to define macros. Perhaps this is why C is occasionally referred to as a *portable macro assembly language.* The large set of preprocessor directives can be used to completely change the look of a C program such that it is very difficult for anyone to decipher. On the other hand, the C preprocessor can be used to make complicated programs easy to follow, very efficient, and easy to code. The remainder of this chapter and the programs discussed in this book hopefully will serve to illustrate the latter advantages of the C preprocessor.

The C preprocessor allows *conditional compilation* of program segments, *user-defined symbolic replacement* of any text in the program (called *aliases* as discussed in Section 2.6.2), and *user-defined multiple parameter macros.* All of the preprocessor directives are evaluated before any C code is compiled and the directives themselves are removed from the program before compilation begins. Each preprocessor directive begins with a pound sign (**#**) followed by the preprocessor keyword. The following list indicates the basic use of each of the most commonly used preprocessor directives:

#define NAME macro	Associate symbol **NAME** with **macro** definition (optional parameters)
#include "file"	Copy named **file** (with directory specified) into current compilation
#include <file>	Include **file** from standard C library
#if expression	Conditionally compile the following code if result of **expression** is true
#ifdef symbol	Conditionally compile the following code if the **symbol** is defined
#ifndef symbol	Conditionally compile the following code if the **symbol** is not defined
#else	Conditionally compile the following code if the associated **#if** is not true
#endif	Indicates the end of previous **#else**, **#if**, **#ifdef**, or **#ifndef**
#undef macro	Undefine previously defined **macro**

2.6.1 Conditional Preprocessor Directives

Most of the above preprocessor directives are used for conditional compilation of portions of a program. For example, in the following version of the **stats** function (described previously in section 2.5.1), the definition of DEBUG is used to indicate that the print statements should be compiled:

```
void stats(float *array,int size,float *ave,float *var)
{
    int i;
    float sum = 0.0;     /* initialize sum of signal */
    float sum2 = 0.0;    /* sum of signal squared */
    for(i = 0 ; i < size ; 1++) {
        sum = sum + array[i];
        sum2 = sum2 + array[i]*array][i];    /* calculate sums */
}

#ifdef DEBUG
    printf("\nIn stats sum = %f sum2 = %f",sum,sum2);
    printf("\nNumber of array elements = %d",size);
#endif

    *ave = sum/size;     /* pass average */
    *var = (sum2 - sum* (*ave))/(size-1);    /* pass variance */
}
```

If the preprocessor parameter DEBUG is defined anywhere before the **#ifdef DEBUG** statement, then the **printf** statements will be compiled as part of the program to aid in debugging **stats** (or perhaps even the calling program). Many compilers allow the definition of preprocessor directives when the compiler is invoked. This allows the DEBUG option to be used with no changes to the program text.

2.6.2 Aliases and Macros

Of all the preprocessor directives, the **#define** directive is the most powerful because it allows aliases and multiple parameter macros to be defined in a relatively simple way. The most common use of **#define** is a macro with no arguments that replaces one string (the macro name) with another string (the macro definition). In this way, an alias can be given to any string including all of the C keywords. For example:

```
#define DO for(
```

replaces every occurrence of the string **DO** (all capital letters so that it is not confused with the C keyword **do**) with the four-character string **for(**. Similarly, new aliases of all the C keywords could be created with several **#define** statements (although this seems silly since the C keywords seem good enough). Even single characters can be aliased. For example, **BEGIN** could be aliased to **{** and **END** could be aliased to **}**, which makes a C program look more like Pascal.

The **#define** directive is much more powerful when parameters are used to create a true macro. The above **DO** macro can be expanded to define a simple FORTRAN style DO loop as follows:

```
#define DO(var,beg,end) for(var=beg; var<=end; var++)
```

The three macro parameters **var**, **beg**, and **end** are the variable, the beginning value, and the ending value of the DO loop. In each case, the macro is invoked and the string placed in each argument is used to expand the macro. For example,

```
DO(i,1,10)
```

expands to

```
for(i=1; i<=10; i++)
```

which is the valid beginning of a **for** loop that will start the variable **i** at 1 and stop it at 10. Although this DO macro does shorten the amount of typing required to create such a simple **for** loop, it must be used with caution. When macros are used with other operators, other macros, or other functions, unexpected program bugs can occur. For example, the above macro will not work at all with a pointer as the **var** argument, because **DO(*ptr,1,10)** would increment the pointer's value and not the value it points to (see section 2.7.1). This would probably result in a very strange number of cycles through the loop (if the loop ever terminated). As another example, consider the following **CUBE** macro, which will determine the cube of a variable:

```
#define CUBE(x)  (x)*(x)*(x)
```

This macro will work fine (although inefficiently) with **CUBE(i+j)**, since it would expand to **(i+j)*(i+j)*(i+j)**. However, **CUBE(i++)** expands to **(i++)*(i++)*(i++)**, resulting in **i** getting incremented three times instead of once. The resulting value would be $x(x+1)(x+2)$ not x^3.

The ternary conditional operator (see section 2.4.3) can be used with macro definitions to make fast implementations of the absolute value of a variable (**ABS**), the minimum of two variables (**MIN**), the maximum of two variables (**MAX**), and the integer rounded value of a floating-point variable (**ROUND**) as follows:

```
#define ABS(a)        (((a) < 0) ? (-a) : (a)
#define MAX(a,b)      (((a) > (b)) ? (a) : (b))
#define MIN(a,b)      (((a) < (b)) ? (a): (b))
#define ROUND(a)      (((a)<0)?(int)((a)-0.5):(int)((a)+0.5))
```

Note that each of the above macros is enclosed in parentheses so that it can be used freely in expressions without uncertainty about the order of operations. Parentheses are also required around each of the macro parameters, since these may contain operators as well as simple variables.

All of the macros defined so far have names that contain only capital letters. While this is not required, it does make it easy to separate macros from normal C keywords in programs where macros may be defined in one module and included (using the **#include** directive) in another. This practice of capitalizing all macro names and using

lower case for variable and function names will be used in all programs in this book and on the accompanying disk.

2.7 POINTERS AND ARRAYS

A *pointer* is a variable that holds an address of some data, rather than the data itself. The use of pointers is usually closely related to manipulating (assigning or changing) the elements of an array of data. Pointers are used primarily for three purposes:

(1) To point to different data elements within an array
(2) To allow a program to create new variables while a program is executing (dynamic memory allocation)
(3) To access different locations in a data structure

The first two uses of pointers will be discussed in this section; pointers to data structures are considered in section 2.8.2.

2.7.1 Special Pointer Operators

Two *special pointer operators* are required to effectively manipulate pointers: the indirection operator (*****) and the address of operator (**&**). The indirection operator (*****) is used whenever the data stored at the address pointed to by a pointer is required, that is, whenever indirect addressing is required. Consider the following simple program:

```
main()
{
    int i,*ptr;
    i = 7;                     /* set the value of i */
    ptr = &i;                  /* point to address of i */
    printf("\n%d",i);          /* print i two ways */
    printf("\n%d",*ptr);
    *ptr = 11;                 /* change i with pointer */
    printf("\n%d %d",*ptr,i);     /* print change */
}
```

This program declares that **i** is an integer variable and that **ptr** is a pointer to an integer variable. The program first sets **i** to 7 and then sets the pointer to the address of **i** by the statement **ptr=&i;**. The compiler assigns **i** and **ptr** storage locations somewhere in memory. At run time, **ptr** is set to the starting address of the integer variable **i**. The above program uses the function **printf** (see section 2.9.1) to print the integer value of i in two different ways—by printing the contents of the variable **i** (**printf("\n%d",i);**), and by using the indirection operator (**printf("\n%d",*ptr);**). The presence of the ***** operator in front of **ptr** directs the compiler to pass the value stored at the address

ptr to the **printf** function (in this case, 7). If only **ptr** were used, then the address assigned to **ptr** would be displayed instead of the value 7. The last two lines of the example illustrate indirect storage; the data at the **ptr** address is changed to 11. This results in changing the value of **i** only because **ptr** is pointing to the address of **i**.

An *array* is essentially a section of memory that is allocated by the compiler and assigned the name given in the declaration statement. In fact, the name given is nothing more than a fixed pointer to the beginning of the array. In C, the array name can be used as a pointer or it can be used to reference an element of the array (i.e., **a[2]**). If **a** is declared as some type of array then ***a** and **a[0]** are exactly equivalent. Furthermore, ***(a+i)** and **a[i]** are also the same (as long as **i** is declared as an integer), although the meaning of the second is often more clear. Arrays can be rapidly and sequentially accessed by using pointers and the increment operator (**++**). For example, the following three statements set the first 100 elements of the array **a** to 10:

```
int *pointer;
pointer = a;
for(i = 0; i < 100 ; i++) *pointer++ = 10;
```

On many computers this code will execute faster than the single statement **for(i=0; i<100; i++) a[i]=10;**, because the post increment of the pointer is faster than the array index calculation.

2.7.2 Pointers and Dynamic Memory Allocation

C has a set of four standard functions that allow the programmer to dynamically change the type and size of variables and arrays of variables stored in the computer's memory. C programs can use the same memory for different purposes and not waste large sections of memory on arrays only used in one small section of a program. In addition, **auto** variables are automatically allocated on the stack at the beginning of a function (or any section of code where the variable is declared within a pair of braces) and removed from the stack when a function is exited (or at the right brace, **}**). By proper use of **auto** variables (see section 2.5.2) and the dynamic memory allocation functions, the memory used by a particular C program can be very little more than the memory required by the program at every step of execution. This feature of C is especially attractive in multiuser environments where the product of the memory size required by a user and the time that memory is used ultimately determines the overall system performance. In many DSP applications, the proper use of dynamic memory allocation can enable a complicated DSP function to be performed with an inexpensive single chip signal processor with a small limited internal memory size instead of a more costly processor with a larger external memory.

Four standard functions are used to manipulate the memory available to a particular program (sometimes called the heap to differentiate it from the stack). **Malloc** allocates bytes of storage, **calloc** allocates items which may be any number of bytes long, **free** removes a previously allocated item from the heap, and **realloc** changes the size of a previously allocated item.

When using each function, the size of the item to be allocated must be passed to the

function. The function then returns a pointer to a block of memory at least the size of the item or items requested. In order to make the use of the memory allocation functions portable from one machine to another, the built-in compiler macro **sizeof** must be used. For example:

```
int *ptr;
ptr = (int *) malloc(sizeof(int));
```

allocates storage for one integer and points the integer pointer, **ptr**, to the beginning of the memory block. On 32-bit machines this will be a four-byte memory block (or one word) and on 16-bit machines (such as the IBM PC) this will typically be only two bytes. Because **malloc** (as well as **calloc** and **realloc**) returns a character pointer, it must be cast to the integer type of pointer by the **(int *)** cast operator. Similarly, **calloc** and a pointer, **array**, can be used to define a 25-element integer array as follows:

```
int *array;
array = (int *) calloc(25,sizeof(int));
```

This statement will allocate an array of 25 elements, each of which is the size of an **int** on the target machine. The array can then be referenced by using another pointer (changing the pointer array is unwise, because it holds the position of the beginning of the allocated memory) or by an array reference such as **array[i]** (where **i** may be from 0 to 24). The memory block allocated by **calloc** is also initialized to zeros.

Malloc, **calloc**, and **free** provide a simple general purpose memory allocation package. The argument to **free** (cast as a character pointer) is a pointer to a block previously allocated by **malloc** or **calloc**; this space is made available for further allocation, but its contents are left undisturbed. Needless to say, grave disorder will result if the space assigned by **malloc** is overrun, or if some random number is handed to **free**. The function **free** has no return value, because memory is always assumed to be happily given up by the operating system.

Realloc changes the size of the block previously allocated to a new size in bytes and returns a pointer to the (possibly moved) block. The contents of the old memory block will be unchanged up to the lesser of the new and old sizes. **Realloc** is used less than **calloc** and **malloc**, because the size of an array is usually known ahead of time. However, if the size of the integer array of 25 elements allocated in the last example must be increased to 100 elements, the following statement can be used:

```
array = (int *) realloc( (char *)array, 100*sizeof(int));
```

Note that unlike **calloc**, which takes two arguments (one for the number of items and one for the item size), **realloc** works similar to **malloc** and takes the total size of the array in bytes. It is also important to recognize that the following two statements are not equivalent to the previous **realloc** statement:

```
free((char *)array);
array = (int *) calloc(100,sizeof(int));
```

These statements do change the size of the integer array from 25 to 100 elements, but do not preserve the contents of the first 25 elements. In fact, **calloc** will initialize all 100 integers to zero, while **realloc** will retain the first 25 and not set the remaining 75 array elements to any particular value.

Unlike **free**, which returns no value, **malloc**, **realloc**, and **calloc** return a null pointer (0) if there is no available memory or if the area has been corrupted by storing outside the bounds of the memory block. When **realloc** returns 0, the block pointed to by the original pointer may be destroyed.

2.7.3 Arrays of Pointers

Any of the C data types or pointers to each of the data types can be declared as an array. Arrays of pointers are especially useful in accessing large matrices. An array of pointers to 10 rows each of 20 integer elements can be dynamically allocated as follows:

```
int *mat[10];
int i;
for(i = 0 ; i < 10 ; i++) {
    mat[i] = (int *)calloc(20,sizeof(int));
    if(!mat[i]) {
        printf("\nError in matrix allocation\n");
        exit(1);
    }
}
```

In this code segment, the array of 10 integer pointers is declared and then each pointer is set to 10 different memory blocks allocated by 10 successive calls to **calloc**. After each call to **calloc**, the pointer must be checked to insure that the memory was available (**!mat[i]** will be true if **mat[i]** is null). Each element in the matrix **mat** can now be accessed by using pointers and the indirection operator. For example, ***(mat[i] + j)** gives the value of the matrix element at the **i**th row (0-9) and the **j**th column (0-19) and is exactly equivalent to **mat[i][j]**. In fact, the above code segment is equivalent (in the way **mat** may be referenced at least) to the array declaration **int mat[10][20];**, except that **mat[10][20]** is allocated as an **auto** variable on the stack and the above calls to **calloc** allocates the space for **mat** on the heap. Note, however, that when **mat** is allocated on the stack as an **auto** variable, it cannot be used with **free** or **realloc** and may be accessed by the resulting code in a completely different way.

The calculations required by the compiler to access a particular element in a two-dimensional matrix (by using **matrix[i][j]**, for example) usually take more instructions and more execution time than accessing the same matrix using pointers. This is especially true if many references to the same matrix row or column are required. However, depending on the compiler and the speed of pointer operations on the target machine, access to a two-dimensional array with pointers and simple pointers operands (even increment and decrement) may take almost the same time as a reference to a matrix such as

a[i][j]. For example, the product of two 100 x 100 matrices could be coded using two-dimensional array references as follows:

```
int a[100][100],b[100][100],c[100][100];      /* 3 matrices */
int i,j,k;                                    /* indices */

/* code to set up mat and vec goes here */

/* do matrix multiply c = a * b */
for(i = 0 ; i < 100 ; j++) {
    for(j = 0 ; j < 100 ; j++) {
        c[i][j] = 0;
        for(k = 0 ; k < 100 ; k++)
            c[i][j] += a[i][k] * b[k][j];
    }
}
```

The same matrix product could also be performed using arrays of pointers as follows:

```
int a[100][100],b[100][100],c[100][100];      /* 3 matrices */
int *aptr,*bptr,*cptr;                        /* pointers to a,b,c */
int i,j,k;                                    /* indicies */.

/* code to set up mat and vec goes here */

/* do c = a * b */
for(i = 0 ; i < 100 ; i++) {
    cptr = c[i];
    bptr = b[0];
    for(j = 0 ; j < 100 ; j++) {
        aptr = a[i];
        *cptr = (*aptr++) * (*bptr++);
        for(k = 1 ; k < 100; k++) {
            *cptr += (*aptr++) * b[k][j];
        }
        cptr++;
    }
}
```

The latter form of the matrix multiply code using arrays of pointers runs 10 to 20 percent faster, depending on the degree of optimization done by the compiler and the capabilities of the target machine. Note that **c[i]** and **a[i]** are references to arrays of pointers each pointing to 100 integer values. Three factors help make the program with pointers faster:

(1) Pointer increments (such as ***aptr++**) are usually faster than pointer adds.

(2) No multiplies or shifts are required to access a particular element of each matrix.

(3) The first add in the inner most loop (the one involving **k**) was taken outside the loop (using pointers **aptr** and **bptr**) and the initialization of **c[i][j]** to zero was removed.

2.8 STRUCTURES

Pointers and arrays allow the same type of data to be arranged in a list and easily accessed by a program. Pointers also allow arrays to be passed to functions efficiently and dynamically created in memory. When unlike logically related data types must be manipulated, the use of several arrays becomes cumbersome. While it is always necessary to process the individual data types separately, it is often desirable to move all of the related data types as a single unit. The powerful C data construct called a *structure* allows new data types to be defined as a combination of any number of the standard C data types. Once the size and data types contained in a structure are defined (as described in the next section), the named structure may be used as any of the other data types in C. Arrays of structures, pointers to structures, and structures containing other structures may all be defined.

One drawback of the user-defined structure is that the standard operators in C do not work with the new data structure. Although the enhancements to C available with the C++ programming language do allow the user to define structure operators (see *The C++ Programming Language,* Stroustrup, 1986), the widely used standard C language does not support such concepts. Thus, functions or macros are usually created to manipulate the structures defined by the user. As an example, some of the functions and macros required to manipulate structures of complex floating-point data are discussed in section 2.8.3.

2.8.1 Declaring and Referencing Structures

A structure is defined by a structure template indicating the type and name to be used to reference each element listed between a pair of braces. The general form of an N-element structure is as follows:

```
struct tag_name {
    type1 element_name1;
    type2 element_name2;
              .
              .
              .
    typeN element_nameN;
        } variable_name;
```

In each case, **type1**, **type2**, . . . , **typeN** refer to a valid C data type (**char**, **int**, **float**, or **double** without any storage class descriptor) and **element_name1**, **element_name2**, . . . , **element_nameN** refer to the name of one of the elements of the data structure. The **tag_name** is an optional name used for referencing the structure later. The optional **variable_name**, or list of variable names, defines the names of the structures to be defined. The following structure template with a tag name of **record** defines a structure containing an integer called **length**, a **float** called **sample_rate**, a character pointer called **name**, and a pointer to an integer array called **data**:

```
struct record {
    int length;
    float sample_rate;
    char *name;
    int *data;
        };
```

This structure template can be used to declare a structure called voice as follows:

```
struct record voice;
```

The structure called **voice** of type **record** can then be initialized as follows:

```
voice.length = 1000;
voice.sample_rate = 10.e3;
voice.name = "voice signal";
```

The last element of the structure is a pointer to the data and must be set to the beginning of a 1000-element integer array (because length is 1000 in the above initialization). Each element of the structure is referenced with the form **struct_name.element**. Thus, the 1000-element array associated with the voice structure can be allocated as follows:

```
voice.data = (int *) calloc(1000,sizeof(int));
```

Similarly, the other three elements of the structure can be displayed with the following code segment:

```
printf("\nLength = %d",voice.length);
printf("\nSampling rate = %f",voice.sample_rate);
printf("\nRecord name = %s",voice.name);
```

A **typedef** statement can be used with a structure to make a user-defined data type and make declaring a structure even easier. The **typedef** defines an alternative name for the structure data type, but is more powerful than **#define**, since it is a compiler directive as opposed to a preprocessor directive. An alternative to the **record** structure is a **typedef** called **RECORD** as follows:

```
typedef struct record RECORD;
```

This statement essentially replaces all occurrences of **RECORD** in the program with the **struct record** definition thereby defining a new type of variable called **RECORD** that may be used to define the voice structure with the simple statement **RECORD voice;**.

The **typedef** statement and the structure definition can be combined so that the tag name **record** is avoided as follows:

```
typedef struct {
    int length;
    float sample_rate;
    char *name;
    int *data;
    } RECORD;
```

In fact, the **typedef** statement can be used to define a shorthand form of any type of data type including pointers, arrays, arrays of pointers, or another **typedef**. For example,

```
typedef char STRING[80];
```

allows 80-character arrays to be easily defined with the simple statement **STRING name1,name2;**. This shorthand form using the **typedef** is an exact replacement for the statement **char name1[80],name2[80];**.

2.8.2 Pointers to Structures

Pointers to structures can be used to dynamically allocate arrays of structures and efficiently access structures within functions. Using the **RECORD typedef** defined in the last section, the following code segment can be used to dynamically allocate a five-element array of **RECORD** structures:

```
RECORD *voices;
voices = (RECORD *) calloc(5,sizeof(RECORD));
```

These two statements are equivalent to the single-array definition **RECORD voices[5];** except that the memory block allocated by **calloc** can be deallocated by a call to the **free** function. With either definition, the length of each element of the array could be printed as follows:

```
int i;
for(i = 0 ; i < 5 ; i++)
    printf("\nLength of voice %d = %d",i,voices[i].length);
```

The **voices** array can also be accessed by using a pointer to the array of structures. If **voice_ptr** is a **RECORD** pointer (by declaring it with **RECORD *voice_ptr;**), then

(*voice_ptr).length could be used to give the length of the **RECORD** which was pointed to by **voice_ptr**. Because this form of pointer operation occurs with structures often in C, a special operator (->) was defined. Thus, **voice_ptr->length** is equivalent to **(*voice_ptr).length**. This shorthand is very useful when used with functions, since a local copy of a structure pointer is passed to the function. For example, the following function will print the length of each record in an array of **RECORD** of length **size:**

```
void print_record_length(RECORD *rec,int size)
{
    int i;
    for(i = 0 ; i < size ; i++) {
        printf("\nLength of record %d = %d",i,rec_>length);
       rec++;
    }
}
```

Thus, a statement like **print_record_length(voices,5);** will print the lengths stored in each of the five elements of the array of **RECORD** structures.

2.8.3 Complex Numbers

A complex number can be defined in C by using a structure of two **floats** and a **typedef** as follows:

```
typedef struct {
    float real;
    float imag;
       } COMPLEX;
```

Three complex numbers, **x**, **y**, and **z** can be defined using the above structure as follows:

```
COMPLEX x,y,z;
```

In order to perform the complex addition z = x + y without functions or macros, the following two C statements are required:

```
z.real = x.real + y.real;
z.imag = x.imag + y.imag;
```

These two statements are required because the C operator **+** can only work with the individual parts of the complex structure and not the structure itself. In fact, a statement involving any operator and a structure should give a compiler error. Assignment of any structure (like **z = x;**) works just fine, because only data movement is involved. A simple function to perform the complex addition can be defined as follows:

```
COMPLEX cadd(COMPLEX a,COMPLEX b)   /* pass by value */
{
    COMPLEX sum;      /* define return value */
    sum.real = a.real + b.real;
    sum.imag = a.imag + b.imag;
    return(sum);
}
```

This function passes the value of the **a** and **b** structures, forms the sum of **a** and **b**, and then returns the complex summation (some compilers may not allow this method of passing structures by value, thus requiring pointers to each of the structures). The **cadd** function may be used to set **z** equal to the sum of **x** and **y** as follows:

```
z = cadd(x,y);
```

The same complex sum can also be performed with a rather complicated single line macro defined as follows:

```
#define CADD(a,b)\
  (C_t.real=a.real+b.real,C_t.imag=a.imag+b.imag,C_t)
```

This macro can be used to replace the **cadd** function used above as follows:

```
COMPLEX C_t;
z = CADD(z,y);
```

This **CADD** macro works as desired because the macro expands to three operations separated by commas. The one-line macro in this case is equivalent to the following three statements:

```
C_t.real = x.real + y.real;
C_t.imag = x.imag + y.real;
z = C_t;
```

The first two operations in the macro are the two sums for the real and imaginary parts. The sums are followed by the variable **C_t** (which must be defined as **COMPLEX** before using the macro). The expression formed is evaluated from left to right and the whole expression in parentheses takes on the value of the last expression, the complex structure **C_t**, which gets assigned to **z** as the last statement above shows.

The complex add macro **CADD** will execute faster than the **cadd** function because the time required to pass the complex structures **x** and **y** to the function, and then pass the sum back to the calling program, is a significant part of the time required for the function call. Unfortunately, the complex add macro cannot be used in the same manner as the function. For example:

```
COMPLEX a,b,c,d;
d = cadd(a,cadd(b,c));
```

will form the sum **d=a+b+c;** as expected. However, the same format using the **CADD** macro would cause a compiler error, because the macro expansion performed by the C preprocessor results in an illegal expression. Thus, the **CADD** may only be used with simple single-variable arguments. If speed is more important than ease of programming, then the macro form should be used by breaking complicated expressions into simpler two-operand expressions. Numerical C extensions to the C language support complex numbers in an optimum way and are discussed in section 2.10.1.

2.9 COMMON C PROGRAMMING PITFALLS

The following sections describe some of the more common errors made by programmers when they first start coding in C and give a few suggestions how to avoid them.

2.9.1 Array Indexing

In C, all array indices start with zero rather than one. This makes the last index of a *N* long array *N*-1. This is very useful in digital signal processing, because many of the expressions for filters, *z*-transforms, and FFTs are easier to understand and use with the index starting at zero instead of one. For example, the FFT output for $k = 0$ gives the zero frequency (DC) spectral component of a discrete time signal. A typical indexing problem is illustrated in the following code segment, which is intended to determine the first 10 powers of 2 and store the results in an array called **power2:**

```
int power2[10];
int i,p;
p = 1;
for (i = 1 ; i<= 10 ; i++) {
    power2[i] = p;
    p = 2*p;
}
```

This code segment will compile well and may even run without any difficulty. The problem is that the **for** loop index **i** stops on **i=10**, and **power2[10]** is not a valid index to the **power2** array. Also, the **for** loop starts with the index 1 causing **power2[0]** to not be initialized. This results in the first power of two (2^0, which should be stored in **power2[0]**) to be placed in **power2[1]**. One way to correct this code is to change the for loop to read **for(i = 0; i<10; i++)**, so that the index to **power2** starts at 0 and stops at 9.

2.9.2 Failure to Pass-by-Address

This problem is most often encountered when first using **scanf** to read in a set of variables. If **i** is an integer (declared as **int i;**), then a statement like **scanf("%d",i);** is wrong because **scanf** expects the address of (or pointer to) the location to store the integer that is read by **scanf**. The correct statement to read in the integer **i** is **scanf("%d",&i);**, where the address of operator (**&**) was used to point to the address of **i** and pass the address to **scanf** as required. On many compilers these types of errors can be detected and avoided by using function prototypes (see section 2.5.3) for all user written functions and the appropriate include files for all C library functions. By using function prototypes, the compiler is informed what type of variable the function expects and will issue a warning if the specified type is not used in the calling program. On many UNIX systems, a C program checker called LINT can be used to perform parameter-type checking, as well as other syntax checking.

2.9.3 Misusing Pointers

Because pointers are new to many programmers, the misuse of pointers in C can be particularly difficult, because most C compilers will not indicate any pointer errors. Some compilers issue a warning for some pointer errors. Some pointer errors will result in the programs not working correctly or, worse yet, the program may seem to work, but will not work with a certain type of data or when the program is in a certain mode of operation. On many small single-user systems (such as the IBM PC), misused pointers can easily result in writing to memory used by the operating system, often resulting in a system crash and requiring a subsequent reboot.

There are two types of pointer abuses: setting a pointer to the wrong value (or not initializing it at all) and confusing arrays with pointers. The following code segment shows both of these problems:

```
char *string;
char msg[10];
int i;
printf("\nEnter title");
scanf("%s",string);
i = 0;
while(*string != ' ') {
    i++;
    string++;
}
msg="Title = ";
printf("%s %s %d before space", msg, string,i);
```

The first three statements declare that memory be allocated to a pointer variable called **string**, a 10-element **char** array called **msg** and an integer called **i**. Next, the user is asked to enter a title into the variable called **string**. The **while** loop is intended to search for the first space in the string and the last **printf** statement is intended to display the string and the number of characters before the first space.

There are three pointer problems in this program, although the program will compile with only one fatal error (and a possible warning). The fatal error message will reference the **msg="Title =";** statement. This line tells the compiler to set the address of the **msg** array to the constant string **"Title ="**. This is not allowed so the error

"Lvalue required" (or something less useful) will be produced. The role of an array and a pointer have been confused and the **msg** variable should have been declared as a pointer and used to point to the constant string **"Title ="**, which was already allocated storage by the compiler.

The next problem with the code segment is that **scanf** will read the string into the address specified by the argument **string**. Unfortunately, the value of **string** at execution time could be anything (some compilers will set it to zero), which will probably not point to a place where the title string could be stored. Some compilers will issue a warning indicating that the pointer called **string** may have been used before it was defined. The problem can be solved by initializing the string pointer to a memory area allocated for storing the title string. The memory can be dynamically allocated by a simple call to **calloc** as shown in the following corrected code segment:

```
char *string,*msg;
int i;
string=calloc(80,sizeof(char));
printf("\nEnter title");
scanf("%s", string);
i = 0;
while(*string != ' ') {
    i++;
    string++;
}
msg="Title =";
printf("%s %s %d before space",msg,string,i);
```

The code will now compile and run but will not give the correct response when a title string is entered. In fact, the first characters of the title string before the first space will not be printed because the pointer **string** was moved to this point by the execution of the **while** loop. This may be useful for finding the first space in the **while** loop, but results in the address of the beginning of the string being lost. It is best not to change a pointer which points to a dynamically allocated section of memory. This pointer problem can be fixed by using another pointer (called **cp**) for the **while** loop as follows:

```
char *string,*cp,*msg;
int i;
string=calloc(80,sizeof(char));
printf("\nEnter title");
scanf("%s",string);
i = 0;
cp = string;
while(*cp != ' ') {
    i++;
    cp++;
}
msg="Title =";
printf("%s %s %d before space", msg,string,i);
```

Another problem with this program segment is that if the string entered contains no spaces, then the **while** loop will continue to search through memory until it finds a space. On a PC, the program will almost always find a space (in the operating system perhaps) and will set **i** to some large value. On larger multiuser systems, this may result in a fatal run-time error because the operating system must protect memory not allocated to the program. Although this programming problem is not unique to C, it does illustrate an important characteristic of pointers—pointers can and will point to any memory location without regard to what may be stored there.

10 NUMERICAL C EXTENSIONS

Some ANSI C compilers designed for DSP processors are now available with numeric C extensions. These language extensions were developed by the ANSI NCEG (Numeric C Extensions Group), a working committee reporting to ANSI X3J11. This section gives an overview of the *Numerical C* language recommended by the ANSI standards committee. Numerical C has several features of interest to DSP programmers:

(1) Fewer lines of code are required to perform vector and matrix operations.

(2) Data types and operators for complex numbers (with real and imaginary components) are defined and can be optimized by the compiler for the target processor. This avoids the use of structures and macros as discussed in section 2.8.3.

(3) The compiler can perform better optimizations of programs containing iteration which allows the target processor to complete DSP tasks in fewer instruction cycles.

2.10.1 Complex Data Types

Complex numbers are supported using the keywords **complex**, **creal**, **cimag**, and **conj**. These keywords are defined when the header file **complex.h** is included. There are six integer complex types and three floating-point complex types, defined as shown in the following example:

```
short int complex i;
int complex j;
long int complex k;
unsigned short int complex ui;
unsigned int complex uj;
unsigned long int complex uk;
float complex x;
double complex y;
long double complex z;
```

The real and imaginary parts of the complex types each have the same representations as the type defined without the complex keyword. Complex constants are represented as a

sum of a real constant and an imaginary constant, which is defined by using the suffix **i** after the imaginary part of the number. For example, initialization of complex numbers is performed as follows:

```
short int complex i = 3 + 2i;
float complex x[3] = {1.0+2.0i, 3.0i, 4.0};
```

The following operators are defined for complex types: **&** (address of), ***** (point to complex number), **+** (add), – (subtract), ***** (multiply), **/** (divide). Bitwise, relational, and logical operators are not defined. If any one of the operands are **complex**, the other operands will be converted to **complex**, and the result of the expression will be **complex**. The **creal** and **cimag** operators can be used in expressions to access the real or imaginary part of a complex variable or constant. The **conj** operator returns the complex conjugate of its complex argument. The following code segment illustrates these operators:

```
float complex a,b,c;
creal(a)=1.0;
cimag(a)=2.0;
creal(b)=2.0*cimag(a);
cimag(b)=3.0;
c=conj(b);          /* c will be 4 - 3i */
```

2.10.2 Iteration Operators

Numerical C offers *iterators* to make writing mathematical expressions that are computed iteratively more simple and more efficient. The two new keywords are **iter** and **sum**. Iterators are variables that expand the execution of an expression to iterate the expression so that the iteration variable is automatically incremented from 0 to the value of the iterator. This effectively places the expression inside an efficient **for** loop. For example, the following three lines can be used to set the 10 elements of array **ix** to the integers 0 to 9:

```
iter I=10;
int ix[10];
ix[I]=I;
```

The **sum** operator can be used to represent the sum of values computed from values of an iterator. The argument to **sum** must be an expression that has a value for each of the iterated variables, and the order of the iteration cannot change the result. The following code segment illustrates the **sum** operator:

```
float a[10],b[10],c[10],d[10][10],e[10][10],f[10][10];
float s;
iter I=10, J=10, K=10;
s=sum(a[I]);     /* computes the sum of a into s */
```

```
b[J]=sum(a[I]);  /* sum of a calculated 10 times and stored
                    in the elements of b */
c[J]=sum(d[I][J]);  /* computes the sum of the column
                       elements of d, the statement is
                       iterated over J */
s=sum(d[I][J]);     /* sums all the elements in d */
f[I][J]=sum(d[I][K]*e[K][J]); /* matrix multiply */
c[I]=sum(d[I][K]*a[K]);       /* matrix * vector */
```

11 COMMENTS ON PROGRAMMING STYLE

The four common measures of good DSP software are *reliability, maintainability, exten- sibility,* and *efficiency.*

A reliable program is one that seldom (if ever) fails. This is especially important in DSP because tremendous amounts of data are often processed using the same program. If the program fails due to one sequence of data passing through the program, it may be dif- ficult, or impossible, to ever determine what caused the problem.

Since most programs of any size will occasionally fail, a maintainable program is one that is easy to fix. A truly maintainable program is one that can be fixed by someone other than the original programmer. It is also sometimes important to be able to maintain a program on more than one type of processor, which means that in order for a program to be truly maintainable, it must be portable.

An extensible program is one that can be easily modified when the requirements change, new functions need to be added, or new hardware features need to be exploited.

An efficient program is often the key to a successful DSP implementation of a de- sired function. An efficient DSP program will use the processing capabilities of the target computer (whether general purpose or dedicated) to minimize the execution time. In a typical DSP system this often means minimizing the number of operations per input sam- ple or maximizing the number of operations that can be performed in parallel. In either case, minimizing the number of operations per second usually means a lower overall sys- tem cost as fast computers typically cost more than slow computers. For example, it could be said that the FFT algorithm reduced the cost of speech processing (both imple- mentation cost and development cost) such that inexpensive speech recognition and gen- eration processors are now available for use by the general public.

Unfortunately, DSP programs often forsake maintainability and extensibility for ef- ficiency. Such is the case for most currently available programmable signal processing integrated circuits. These devices are usually programmed in assembly language in such a way that it is often impossible for changes to be made by anyone but the original pro- grammer, and after a few months even the original programmer may have to rewrite the program to add additional functions. Often a compiler is not available for the processor or the processor's architecture is not well suited to efficient generation of code from a com- piled language. The current trend in programmable signal processors appears to be to- ward high-level languages. In fact, many of the DSP chip manufacturers are supplying C compilers for their more advanced products.

2.11.1 Software Quality

The four measures of software quality (reliability, maintainability, extensibility, and effi- ciency) are rather difficult to quantify. One almost has to try to modify a program to find out if it is maintainable or extensible. A program is usually tested in a finite number of ways much smaller than the millions of input data conditions. This means that a program can be considered reliable only after years of bug-free use in many different environ- ments.

Programs do not acquire these qualities by accident. It is unlikely that good pro- grams will be intuitively created just because the programmer is clever, experienced, or uses lots of comments. Even the use of structured-programming techniques (described briefly in the next section) will not assure that a program is easier to maintain or extend. It is the author's experience that the following five coding situations will often lessen the software quality of DSP programs:

(1) Functions that are too big or have several purposes
(2) A main program that does not use functions
(3) Functions that are tightly bound to the main program
(4) Programming "tricks" that are always poorly documented
(5) Lack of meaningful variable names and comments

An *oversized function* (item 1) might be defined as one that exceeds two pages of source listing. A function with more than one purpose lacks strength. A function with one clearly defined purpose can be used by other programs and other programmers. Functions with many purposes will find limited utility and limited acceptance by others. All of the func- tions described in this book and contained on the included disk were designed with this important consideration in mind. Functions that have only one purpose should rarely ex- ceed one page. This is not to say that all functions will be smaller than this. In time- critical DSP applications, the use of in-line code can easily make a function quite long but can sometimes save precious execution time. It is generally true, however, that big programs are more difficult to understand and maintain than small ones.

A *main program* that does not use functions (item 2) will often result in an ex- tremely long and hard-to-understand program. Also, because complicated operations often can be independently tested when placed in short functions, the program may be easier to debug. However, taking this rule to the extreme can result in functions that are *tightly bound* to the main program, violating item 3. A function that is tightly bound to the rest of the program (by using too many global variables, for example) weakens the entire program. If there are lots of tightly coupled functions in a program, maintenance becomes impossible. A change in one function can cause an undesired, unexpected change in the rest of the functions.

Clever programming tricks (item 4) should be avoided at all costs as they will often not be reliable and will almost always be difficult for someone else to understand (even with lots of comments). Usually, if the program timing is so close that a trick must be used, then the wrong processor was chosen for the application. Even if the programm trick solves a particular timing problem, as soon as the system requirements change they almost always do), a new timing problem without a solution may soon develop.

A program that does not use *meaningful variables and comments* (item 5) is gu anteed to be very difficult to maintain. Consider the following valid C program:

```
main(){int _o_oo_,_ooo;for(_o_oo__=2;;__o__o__++)
{for(__ooo_=2;_o__oo__%__ooo_!=0;__ooo_++;
if(__ooo_==_o_oo__)printf("\n%d",_o_oo__);}}
```

Even the most experienced C programmer would have difficulty determining what th three-line program does. Even after running such a poorly documented program, it m be hard to determine how the results were obtained. The following program does exac the same operations as the above three lines but is easy to follow and modify:

```
main()
{
  int prime_test,divisor;
/* The outer for loop trys all numbers >1 and the inner
for loop checks the number tested for any divisors
less than itself. */
  for(prime_test = 2 ; ; prime_test++) {
    for(divisor = 2 ; prime_test % divisor != 0 ; divisor++);
    if(divisor == prime_test) printf("\n%d",prime_test);
  }
}
```

It is easy for anyone to discover that the above well-documented program prints a list prime numbers, because the following three documentation rules were followed:

(1) Variable names that are meaningful in the context of the program were used. Av variable names such as **x**,**y**,**z** or **i**,**j**,**k**, unless they are simple indexes used in very obvious way, such as initializing an entire array to a constant.

(2) Comments preceded each major section of the program (the above program on has one section). Although the meaning of this short program is fairly clear witho the comments, it rarely hurts to have too many comments. Adding a blank line b tween different parts of a program also sometimes improves the readability of program because the different sections of code appear separated from each other.

(3) Statements at different levels of nesting were indented to show which control stru ture controls the execution of the statements at a particular level. The author pref to place the right brace (**{**) with the control structure (**for**, **while**, **if**, etc.) and place the left brace (**}**) on a separate line starting in the same column as the beg ning of the corresponding control structure. The exception to this practice is function declarations where the right brace is placed on a separate line after the gument declarations.

2.11.2 Structured Programming

Structured programming has developed from the notion that any algorithm, no matter how complex, can be expressed by using the programming-control structures *if-else*, *while*, and *sequence*. All programming languages must contain some representation of these three fundamental control structures. The development of structured programming revealed that if a program uses these three control structures, then the logic of the program can be read and understood by beginning at the first statement and continuing downward to the last. Also, all programs could be written without goto statements. Generally, structured-programming practices lead to code that is easier to read, easier to maintain, and even easier to write.

The C language has the three basic control structures as well as three additional structured-programming constructs called *do-while*, *for*, and *case*. The additional three control structures have been added to C and most other modern languages because they are convenient, they retain the original goals of structured programming, and their use often makes a program easier to comprehend.

The *sequence* control structure is used for operations that will be executed once in a function or program in a fixed sequence. This structure is often used where speed is most important and is often referred to as in-line code when the sequence of operations are identical and could be coded using one of the other structures. Extensive use of in-line code can obscure the purpose of the code segment.

The *if-else* control structure in C is the most common way of providing conditional execution of a sequence of operations based on the result of a logical operation. Indenting of different levels of **if** and **else** statements (as shown in the example in section 2.4.1) is not required; it is an expression of C programming style that helps the readability of the if-else control structure. Nested **while** and **for** loops should also be indented for improved readability (as illustrated in section 2.7.3).

The *case* control structure is a convenient way to execute one of a series of operations based upon the value of an expression (see the example in section 2.4.2). It is often used instead of a series of if-else structures when a large number of conditions are tested based upon a common expression. In C, the **switch** statement gives the expression to test and a series of **case** statements give the conditions to match the expression. A **default** statement can be optionally added to execute a sequence of operations if none of the listed conditions are met.

The last three control structures (while, do-while, and for) all provide for repeating a sequence of operations a fixed or variable number of times. These loop statements can make a program easy to read and maintain. The **while** loop provides for the iterative execution of a series of statements as long as a tested condition is true; when the condition is false, execution continues to the next statement in the program. The **do-while** control structure is similar to the **while** loop, except that the sequence of statements is executed at least once. The **for** control structure provides for the iteration of statements with automatic modification of a variable and a condition that terminates the iterations. **For** loops are more powerful in C than most languages. C allows for any initializing statement, any iterating statement and any terminating statement. The three statements do

not need to be related and any of them can be a null statement or multiple statements
following three examples of **while**, **do-while**, and **for** loops all calculate the po
of two of an integer **i** (assumed to be greater than 0) and set the result to **k**. The wh
loop is as follows:

```
k = 2;  /* while loop k=2**i */
while(i > 0) {
    k = 2*k;
    i--;
}
```

The **do-while** loop is as follows:

```
k = 1;  /* do-while loop k = 2**i */
do {
    k = 2*k;
    i--;
} while(i > 0);
```

The **for** loop is as follows:

```
for(k = 2 ; i > 1 ; i--)
    k = 2*k; /* for loop k=2**i */
```

Which form of loop to use is a personal matter. Of the three equivalent code segmen
shown above, the **for** loop and the **while** loop both seem easy to understand and woul
probably be preferred over the **do-while** construction.

The C language also offers several extensions to the six structured programmin
control structures. Among these are **break**, **continue**, and **goto** (see section 2.4.5
Break and **continue** statements allow the orderly interruption of events that are exe
cuting inside of loops. They can often make a complicated loop very difficult to follow
because more than one condition may cause the iterations to stop. The infamous **got**
statement is also included in C. Nearly every language designer includes a **goto** state
ment with the advice that it should never be used along with an example of where
might be useful.

The program examples in the following chapters and the programs contained on th
enclosed disk were developed by using structured-programming practices. The code ca
be read from top to bottom, there are no **goto** statements, and the six accepted contr
structures are used. One requirement of structured programming that was not adopte
throughout the software in this book is that each program and function have only on
entry and exit point. Although every function and program does have only one entr
point (as is required in C), many of the programs and functions have multiple exit point
Typically, this is done in order to improve the readability of the program. For exampl
error conditions in a main program often require terminating the program prematurel

after displaying an error message. Such an error-related exit is performed by calling the C library function **exit(n)** with a suitable error code, if desired. Similarly, many of the functions have more than one return statement as this can make the logic in a function much easier to program and in some cases more efficient.

REFERENCES

FEUER, A.R. (1982). *The C Puzzle Book.* Englewood Cliffs, NJ: Prentice Hall.

KERNIGHAM, B. and PLAUGER, P. (1978). *The Elements of Programming Style.* New York: McGraw-Hill.

KERNIGHAN, B.W. and RITCHIE, D.M. (1988). *The C Programming Language* (2nd ed.). Englewood Cliffs, NJ: Prentice Hall.

PRATA, S. (1986). *Advanced C Primer++.* Indianapolis, IN: Howard W. Sams and Co.

PURDUM, J. and LESLIE, T.C. (1987). *C Standard Library.* Indianapolis, IN: Que Co.

ROCHKIND, M.J. (1988). *Advanced C Programming for Displays.* Englewood Cliffs, NJ: Prentice Hall.

STEVENS, A. (1986). *C Development Tools for the IBM PC.* Englewood Cliffs, NJ: Prentice Hall.

STROUSTRUP, B. (1986). *The C++ Programming Language.* Reading, MA: Addison-Wesley.

WAITE, M., PRATA, S. and MARTIN, D. (1987). *C Primer Plus.* Indianapolis, IN: Howard W. Sams and Co.