Kernighan, B., and Ritchie, D. 1978, *The C Programming Language* (Englewood Cliffs, NJ: Prentice-Hall). [2] [Reference for K&R "traditional" C. Later editions of this book conform to the ANSI C standard.]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [3]

1.1 Program Organization and Control Structures

We sometimes like to point out the close analogies between computer programs, on the one hand, and written poetry or written musical scores, on the other. All three present themselves as visual media, symbols on a two-dimensional page or computer screen. Yet, in all three cases, the visual, two-dimensional, *frozen-in-time* representation communicates (or is supposed to communicate) something rather different, namely a process that *unfolds in time*. A poem is meant to be read; music, played; a program, executed as a sequential series of computer instructions.

In all three cases, the target of the communication, in its visual form, is a human being. The goal is to transfer to him/her, as efficiently as can be accomplished, the greatest degree of understanding, in advance, of how the process *will* unfold in time. In poetry, this human target is the reader. In music, it is the performer. In programming, it is the program user.

Now, you may object that the target of communication of a program is not a human but a computer, that the program user is only an irrelevant intermediary, a lackey who feeds the machine. This is perhaps the case in the situation where the business executive pops a diskette into a desktop computer and feeds that computer a black-box program in binary executable form. The computer, in this case, doesn't much care whether that program was written with "good programming practice" or not.

We envision, however, that you, the readers of this book, are in quite a different situation. You need, or want, to know not just *what* a program does, but also *how* it does it, so that you can tinker with it and modify it to your particular application. You need others to be able to see what you have done, so that they can criticize or admire. In such cases, where the desired goal is *maintainable* or *reusable* code, the targets of a program's communication are surely human, not machine.

One key to achieving good programming practice is to recognize that programming, music, and poetry — all three being symbolic constructs of the human brain — are naturally structured into hierarchies that have many different nested levels. Sounds (phonemes) form small meaningful units (morphemes) which in turn form words; words group into phrases, which group into sentences; sentences make paragraphs, and these are organized into higher levels of meaning. Notes form musical phrases, which form themes, counterpoints, harmonies, etc.; which form movements, which form concertos, symphonies, and so on.

The structure in programs is equally hierarchical. Appropriately, good programming practice brings different techniques to bear on the different levels [1-3]. At a low level is the `ascii` character set. Then, constants, identifiers, operands, operators. Then program statements, like `a[j+1]=b+c/3.0;`. Here, the best programming advice is simply *be clear*, or (correspondingly) *don't be too tricky*. You might momentarily be proud of yourself at writing the single line

```
k=(2-j)*(1+3*j)/2;
```

if you want to permute cyclically one of the values `j` $= (0, 1, 2)$ into respectively `k` $= (1, 2, 0)$. You will regret it later, however, when you try to understand that line. Better, and likely also faster, is

```
k=j+1;
if (k == 3) k=0;
```

Many programming stylists would even argue for the ploddingly literal

```
switch (j) {
    case 0: k=1; break;
    case 1: k=2; break;
    case 2: k=0; break;
    default: {
        fprintf(stderr,"unexpected value for j");
        exit(1);
    }
}
```

on the grounds that it is both clear and additionally safeguarded from wrong assumptions about the possible values of `j`. Our preference among the implementations is for the middle one.

In this simple example, we have in fact traversed several levels of hierarchy: Statements frequently come in "groups" or "blocks" which make sense only taken as a whole. The middle fragment above is one example. Another is

```
swap=a[j];
a[j]=b[j];
b[j]=swap;
```

which makes immediate sense to any programmer as the exchange of two variables, while

```
ans=sum=0.0;
n=1;
```

is very likely to be an initialization of variables prior to some iterative process. This level of hierarchy in a program is usually evident to the eye. It is good programming practice to put in comments at this level, e.g., "initialize" or "exchange variables."

The next level is that of *control structures*. These are things like the `switch` construction in the example above, `for` loops, and so on. This level is sufficiently important, and relevant to the hierarchical level of the routines in this book, that we will come back to it just below.

At still higher levels in the hierarchy, we have functions and modules, and the whole "global" organization of the computational task to be done. In the musical analogy, we are now at the level of movements and complete works. At these levels, *modularization* and *encapsulation* become important programming concepts, the general idea being that program units should interact with one another only through clearly defined and narrowly circumscribed interfaces. Good modularization practice is an essential prerequisite to the success of large, complicated software projects, especially those employing the efforts of more than one programmer. It is also good practice (if not quite as essential) in the less massive programming tasks that an individual scientist, or reader of this book, encounters.

Some computer languages, such as Modula-2 and `C++`, promote good modularization with higher-level language constructs absent in `C`. In Modula-2, for example, functions, type definitions, and data structures can be encapsulated into "modules" that communicate through declared public interfaces and whose internal workings are hidden from the rest of the program [4]. In the `C++` language, the key concept is "class," a user-definable generalization of data type that provides for data hiding, automatic initialization of data, memory management, dynamic typing, and operator overloading (i.e., the user-definable extension of operators like `+` and `*` so as to be appropriate to operands in any particular class) [5]. Properly used in defining the data structures that are passed between program units, classes can clarify and circumscribe these units' public interfaces, reducing the chances of programming error and also allowing a considerable degree of compile-time and run-time error checking.

Beyond modularization, though depending on it, lie the concepts of *object-oriented programming*. Here a programming language, such as `C++` or Turbo Pascal 5.5 [6], allows a module's public interface to accept redefinitions of types or actions, and these redefinitions become shared all the way down through the module's hierarchy (so-called *polymorphism*). For example, a routine written to invert a matrix of real numbers could — dynamically, at run time — be made able to handle complex numbers by overloading complex data types and corresponding definitions of the arithmetic operations. Additional concepts of *inheritance* (the ability to define a data type that "inherits" all the structure of another type, plus additional structure of its own), and *object extensibility* (the ability to add functionality to a module without access to its source code, e.g., at run time), also come into play.

We have not attempted to modularize, or make objects out of, the routines in this book, for at least two reasons. First, the chosen language, `C`, does not really make this possible. Second, we envision that you, the reader, might want to incorporate the algorithms in this book, a few at a time, into modules or objects with a structure of your own choosing. There does not exist, at present, a standard or accepted set of "classes" for scientific object-oriented computing. While we might have tried to invent such a set, doing so would have inevitably tied the algorithmic content of the book (which is its *raison d'être*) to some rather specific, and perhaps haphazard, set of choices regarding class definitions.

On the other hand, we are not unfriendly to the goals of modular and object-oriented programming. Within the limits of `C`, we have therefore tried to structure our programs to be "object friendly." That is one reason we have adopted ANSI C with its function prototyping as our default `C` dialect (see §1.2). Also, within our implementation sections, we have paid particular attention to the practices of *structured programming*, as we now discuss.

Control Structures

An executing program unfolds in time, but not strictly in the linear order in which the statements are written. Program statements that affect the order in which statements are executed, or that affect whether statements are executed, are called *control statements*. Control statements never make useful sense by themselves. They make sense only in the context of the groups or blocks of statements that they in turn control. If you think of those blocks as paragraphs containing sentences, then the control statements are perhaps best thought of as the indentation of the paragraph and the punctuation between the sentences, not the words within the sentences.

We can now say what the goal of structured programming is. It is *to make program control manifestly apparent in the visual presentation of the program*. You see that this goal has nothing at all to do with how the computer sees the program. As already remarked, computers don't care whether you use structured programming or not. Human readers, however, *do* care. You yourself will also care, once you discover how much easier it is to perfect and debug a well-structured program than one whose control structure is obscure.

You accomplish the goals of structured programming in two complementary ways. First, you acquaint yourself with the small number of essential control structures that occur over and over again in programming, and that are therefore given convenient representations in most programming languages. You should learn to think about your programming tasks, insofar as possible, exclusively in terms of these standard control structures. In writing programs, you should get into the habit of representing these standard control structures in consistent, conventional ways.

"Doesn't this inhibit *creativity*?" our students sometimes ask. Yes, just as Mozart's creativity was inhibited by the sonata form, or Shakespeare's by the metrical requirements of the sonnet. The point is that creativity, when it is meant to communicate, does *well* under the inhibitions of appropriate restrictions on format.

Second, you *avoid*, insofar as possible, control statements whose controlled blocks or objects are difficult to discern at a glance. This means, in practice, that *you must try to avoid named labels on statements and* `goto`*'s*. It is not the `goto`'s that are dangerous (although they do interrupt one's reading of a program); the named statement labels are the hazard. In fact, whenever you encounter a named statement label while reading a program, you will soon become conditioned to get a sinking feeling in the pit of your stomach. Why? Because the following questions will, by habit, immediately spring to mind: Where did control come *from* in a branch to this label? It could be anywhere in the routine! What circumstances resulted in a branch to this label? They could be anything! Certainty becomes uncertainty, understanding dissolves into a morass of possibilities.

Some examples are now in order to make these considerations more concrete (see Figure 1.1.1).

Catalog of Standard Structures

Iteration. In `C`, simple iteration is performed with a `for` loop, for example

```
for (j=2;j<=1000;j++) {
    b[j]=a[j-1];
    a[j-1]=j;
}
```

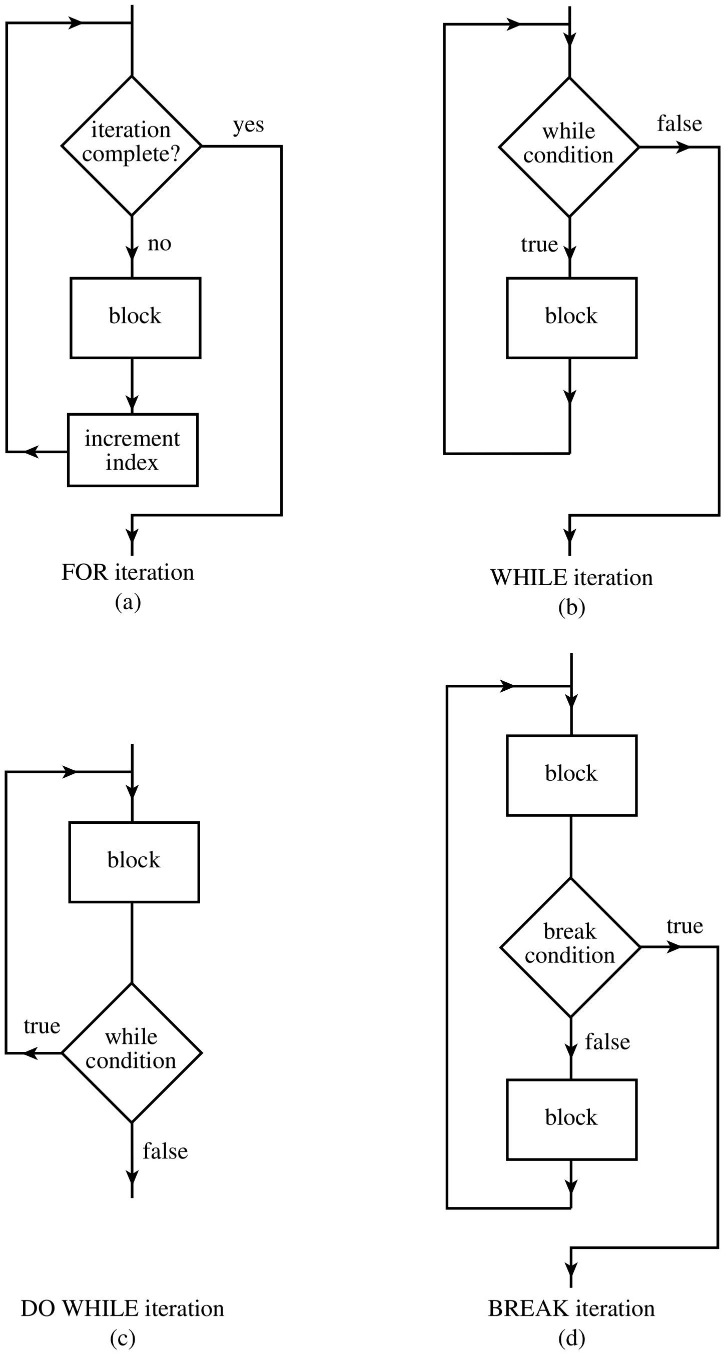
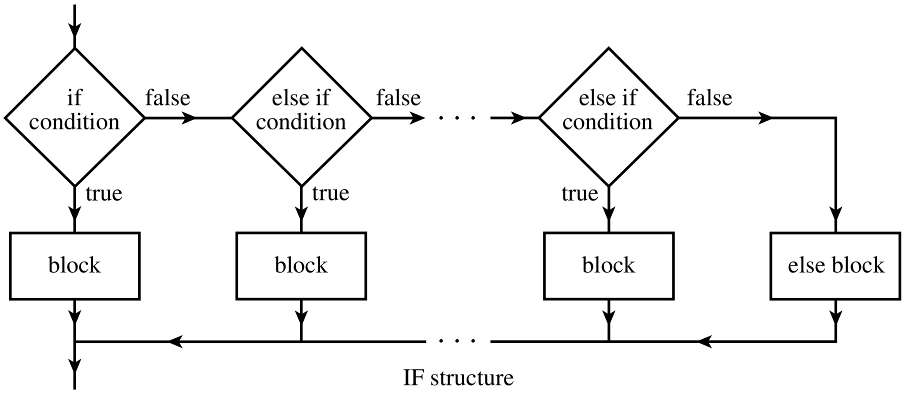
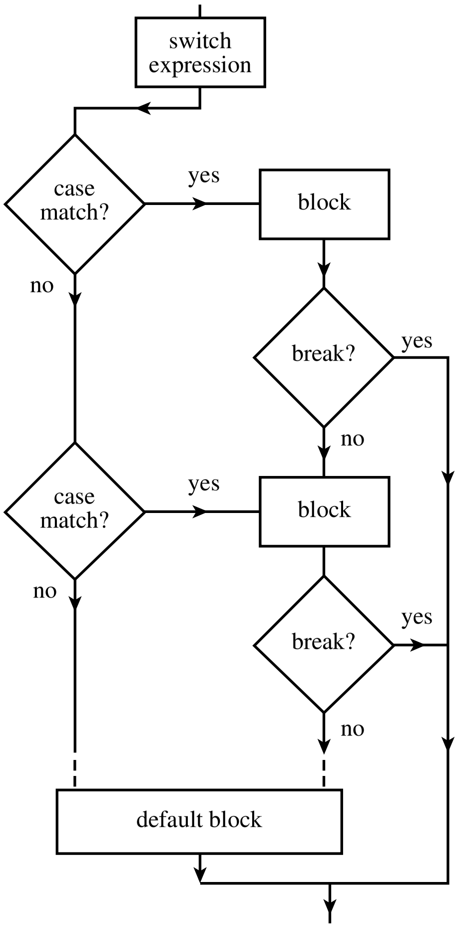


Figure 1.1.1. Standard control structures used in structured programming: (a) `for` iteration; (b) `while` iteration; (c) `do while` iteration; (d) `break` iteration; (e) `if` structure; (f) `switch` structure



IF structure
(e)



SWITCH structure
(f)

Figure 1.1.1. Standard control structures used in structured programming (see caption on previous page).

Notice how we always indent the block of code that is acted upon by the control structure, leaving the structure itself unindented. Notice also our habit of putting the initial curly brace on the same line as the `for` statement, instead of on the next line. This saves a full line of white space, and our publisher loves us for it.

IF structure. This structure in C is similar to that found in Pascal, Algol, FORTRAN and other languages, and typically looks like

```
if (...) {
    ...
}
else if (...) {
    ...
}
else {
    ...
}
```

Since compound-statement curly braces are required only when there is more than one statement in a block, however, C's `if` construction can be somewhat less explicit than the corresponding structure in FORTRAN or Pascal. Some care must be exercised in constructing nested `if` clauses. For example, consider the following:

```
if (b > 3)
    if (a > 3) b += 1;
else b -= 1;                  /* questionable! */
```

As judged by the indentation used on successive lines, the intent of the writer of this code is the following: 'If b is greater than 3 and a is greater than 3, then increment b. If b is not greater than 3, then decrement b.' According to the rules of C, however, the actual meaning is 'If b is greater than 3, then evaluate a. If a is greater than 3, then increment b, and if a is less than or equal to 3, decrement b.' The point is that an `else` clause is associated with the most recent open `if` statement, no matter how you lay it out on the page. Such confusions in meaning are easily resolved by the inclusion of braces. They may in some instances be technically superfluous; nevertheless, they clarify your intent and improve the program. The above fragment should be written as

```
if (b > 3) {
    if (a > 3) b += 1;
} else {
    b -= 1;
}
```

Here is a working program that consists dominantly of `if` control statements:

```
#include <math.h>
#define IGREG (15+31L*(10+12L*1582))        Gregorian Calendar adopted Oct. 15, 1582.

long julday(int mm, int id, int iyyy)
```
In this routine `julday` returns the Julian Day Number that begins at noon of the calendar date specified by month `mm`, day `id`, and year `iyyy`, all integer variables. Positive year signifies A.D.; negative, B.C. Remember that the year after 1 B.C. was 1 A.D.
```
{
    void nrerror(char error_text[]);
```

```
    long jul;
    int ja,jy=iyyy,jm;

    if (jy == 0) nrerror("julday: there is no year zero.");
    if (jy < 0) ++jy;
    if (mm > 2) {                              Here is an example of a block IF-structure.
        jm=mm+1;
    } else {
        --jy;
        jm=mm+13;
    }
    jul = (long) (floor(365.25*jy)+floor(30.6001*jm)+id+1720995);
    if (id+31L*(mm+12L*iyyy) >= IGREG) {       Test whether to change to Gregorian Cal-
        ja=(int)(0.01*jy);                        endar.
        jul += 2-ja+(int) (0.25*ja);
    }
    return jul;
}
```

(Astronomers number each 24-hour period, starting and ending at *noon*, with a unique integer, the Julian Day Number [7]. Julian Day Zero was a very long time ago; a convenient reference point is that Julian Day 2440000 began at noon of May 23, 1968. If you know the Julian Day Number that begins at noon of a given calendar date, then the day of the week of that date is obtained by adding 1 and taking the result modulo base 7; a zero answer corresponds to Sunday, 1 to Monday, . . . , 6 to Saturday.)

While iteration. Most languages (though not `FORTRAN`, incidentally) provide for structures like the following `C` example:

```
while (n < 1000) {
    n *= 2;
    j += 1;
}
```

It is the particular feature of this structure that the control-clause (in this case `n < 1000`) is evaluated *before* each iteration. If the clause is not true, the enclosed statements will not be executed. In particular, if this code is encountered at a time when `n` is greater than or equal to 1000, the statements will not even be executed once.

Do-While iteration. Companion to the `while` iteration is a related control-structure that tests its control-clause at the *end* of each iteration. In `C`, it looks like this:

```
do {
    n *= 2;
    j += 1;
} while (n < 1000);
```

In this case, the enclosed statements will be executed at least once, independent of the initial value of `n`.

Break. In this case, you have a loop that is repeated indefinitely until some condition *tested somewhere in the middle of the loop* (and possibly tested in more than one place) becomes true. At that point you wish to exit the loop and proceed with what comes after it. In C the structure is implemented with the simple `break` statement, which terminates execution of the innermost `for`, `while`, `do`, or `switch` construction and proceeds to the next sequential instruction. (In `Pascal` and standard `FORTRAN`, this structure requires the use of statement labels, to the detriment of clear programming.) A typical usage of the break statement is:

```
for(;;) {
    [statements before the test]
    if (...) break;
    [statements after the test]
}
[next sequential instruction]
```

Here is a program that uses several different iteration structures. One of us was once asked, for a scavenger hunt, to find the date of a Friday the 13th on which the moon was full. This is a program which accomplishes that task, giving incidentally all other Fridays the 13th as a by-product.

```
#include <stdio.h>
#include <math.h>
#define ZON -5.0                          Time zone −5 is Eastern Standard Time.
#define IYBEG 1900                        The range of dates to be searched.
#define IYEND 2000

int main(void) /* Program badluk */
{
    void flmoon(int n, int nph, long *jd, float *frac);
    long julday(int mm, int id, int iyyy);
    int ic,icon,idwk,im,iyyy,n;
    float timzon = ZON/24.0,frac;
    long jd,jday;

    printf("\nFull moons on Friday the 13th from %5d to %5d\n",IYBEG,IYEND);
    for (iyyy=IYBEG;iyyy<=IYEND;iyyy++) {     Loop over each year,
        for (im=1;im<=12;im++) {              and each month.
            jday=julday(im,13,iyyy);          Is the 13th a Friday?
            idwk=(int) ((jday+1) % 7);
            if (idwk == 5) {
                n=(int)(12.37*(iyyy-1900+(im-0.5)/12.0));
                This value n is a first approximation to how many full moons have occurred
                since 1900. We will feed it into the phase routine and adjust it up or down
                until we determine that our desired 13th was or was not a full moon. The
                variable icon signals the direction of adjustment.
                icon=0;
                for (;;) {
                    flmoon(n,2,&jd,&frac);    Get date of full moon n.
                    frac=24.0*(frac+timzon);  Convert to hours in correct time zone.
                    if (frac < 0.0) {         Convert from Julian Days beginning at
                        --jd;                     noon to civil days beginning at mid-
                        frac += 24.0;             night.
                    }
                    if (frac > 12.0) {
                        ++jd;
                        frac -= 12.0;
                    } else
                        frac += 12.0;
                    if (jd == jday) {         Did we hit our target day?
                        printf("\n%2d/13/%4d\n",im,iyyy);
                        printf("%s %5.1f %s\n","Full moon",frac,
```

```
                        " hrs after midnight (EST)");
                    break;                   Part of the break-structure, a match.
                } else {                     Didn't hit it.
                    ic=(jday >= jd ? 1 : -1);
                    if (ic == (-icon)) break;      Another break, case of no match.
                    icon=ic;
                    n += ic;
                }
            }
        }
    }
    }
    return 0;
}
```

If you are merely curious, there were (or will be) occurrences of a full moon on Friday the 13th (time zone GMT-5) on: 3/13/1903, 10/13/1905, 6/13/1919, 1/13/1922, 11/13/1970, 2/13/1987, 10/13/2000, 9/13/2019, and 8/13/2049.

Other "standard" structures. Our advice is to avoid them. Every programming language has some number of "goodies" that the designer just couldn't resist throwing in. They seemed like a good idea at the time. Unfortunately they don't stand the *test* of time! Your program becomes difficult to translate into other languages, and difficult to read (because rarely used structures are unfamiliar to the reader). You can almost always accomplish the supposed conveniences of these structures in other ways.

In C, the most problematic control structure is the `switch...case...default` construction (see Figure 1.1.1), which has historically been burdened by uncertainty, from compiler to compiler, about what data types are allowed in its control expression. Data types `char` and `int` are universally supported. For other data types, e.g., `float` or `double`, the structure should be replaced by a more recognizable and translatable `if...else` construction. ANSI C allows the control expression to be of type `long`, but many older compilers do not.

The `continue;` construction, while benign, can generally be replaced by an `if` construction with no loss of clarity.

About "Advanced Topics"

Material set in smaller type, like this, signals an "advanced topic," either one outside of the main argument of the chapter, or else one requiring of you more than the usual assumed mathematical background, or else (in a few cases) a discussion that is more speculative or an algorithm that is less well-tested. Nothing important will be lost if you skip the advanced topics on a first reading of the book.

You may have noticed that, by its looping over the months and years, the program `badluk` avoids using any algorithm for converting a Julian Day Number back into a calendar date. A routine for doing just this is not very interesting structurally, but it is occasionally useful:

```
#include <math.h>
#define IGREG 2299161

void caldat(long julian, int *mm, int *id, int *iyyy)
Inverse of the function julday given above. Here julian is input as a Julian Day Number,
and the routine outputs mm,id, and iyyy as the month, day, and year on which the specified
Julian Day started at noon.
{
    long ja,jalpha,jb,jc,jd,je;
```

```
    if (julian >= IGREG) {        Cross-over to Gregorian Calendar produces this correc-
        jalpha=(long)(((float) (julian-1867216)-0.25)/36524.25);       tion.
        ja=julian+1+jalpha-(long) (0.25*jalpha);
    } else if (julian < 0) {     Make day number positive by adding integer number of
        ja=julian+36525*(1-julian/36525);      Julian centuries, then subtract them off
    } else                                     at the end.
        ja=julian;
    jb=ja+1524;
    jc=(long)(6680.0+((float) (jb-2439870)-122.1)/365.25);
    jd=(long)(365*jc+(0.25*jc));
    je=(long)((jb-jd)/30.6001);
    *id=jb-jd-(long) (30.6001*je);
    *mm=je-1;
    if (*mm > 12) *mm -= 12;
    *iyyy=jc-4715;
    if (*mm > 2) --(*iyyy);
    if (*iyyy <= 0) --(*iyyy);
    if (julian < 0) iyyy -= 100*(1-julian/36525);
}
```

(For additional calendrical algorithms, applicable to various historical calendars, see [8].)

CITED REFERENCES AND FURTHER READING:

Harbison, S.P., and Steele, G.L., Jr. 1991, *C: A Reference Manual*, 3rd ed. (Englewood Cliffs, NJ: Prentice-Hall).

Kernighan, B.W. 1978, *The Elements of Programming Style* (New York: McGraw-Hill). [1]

Yourdon, E. 1975, *Techniques of Program Structure and Design* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Jones, R., and Stewart, I. 1987, *The Art of C Programming* (New York: Springer-Verlag). [3]

Hoare, C.A.R. 1981, *Communications of the ACM*, vol. 24, pp. 75–83.

Wirth, N. 1983, *Programming in Modula-2*, 3rd ed. (New York: Springer-Verlag). [4]

Stroustrup, B. 1986, *The C++ Programming Language* (Reading, MA: Addison-Wesley). [5]

Borland International, Inc. 1989, *Turbo Pascal 5.5 Object-Oriented Programming Guide* (Scotts Valley, CA: Borland International). [6]

Meeus, J. 1982, *Astronomical Formulae for Calculators*, 2nd ed., revised and enlarged (Richmond, VA: Willmann-Bell). [7]

Hatcher, D.A. 1984, *Quarterly Journal of the Royal Astronomical Society*, vol. 25, pp. 53–55; see also *op. cit.* 1985, vol. 26, pp. 151–155, and 1986, vol. 27, pp. 506–507. [8]

1.2 Some C Conventions for Scientific Computing

The C language was devised originally for systems programming work, not for scientific computing. Relative to other high-level programming languages, C puts the programmer "very close to the machine" in several respects. It is operator-rich, giving direct access to most capabilities of a machine-language instruction set. It has a large variety of intrinsic data types (short and long, signed and unsigned integers; floating and double-precision reals; pointer types; etc.), and a concise syntax for effecting conversions and indirections. It defines an arithmetic on pointers (addresses) that relates gracefully to array addressing and is highly compatible with the index register structure of many computers.