
C for Mathematicians

James A. Carlson & Jennifer M. Johnson

(c) January 5, 2001

Contents

1. The Basics of Programming in C	1
1. Algorithms	1
2. C Programs	7
3. Loops	15
4. Control structures: review and comments	21
5. Computer Arithmetic and Error Analysis	28
6. Files	33

Chapter 1. The Basics of Programming in C

Computer programs are based on *algorithms*. Like the set of rules we learned in grade school for long division, these are step-by-step recipes that specify a sequence of actions which result in the computation of some quantity. In section one we will learn how to read, design, and write algorithms. Typical problems that can be solved algorithmically are computing the greatest common divisor of two integers, factoring an integer into primes, or computing the natural logarithm of 2, which we do by adding up enough terms in the series

$$1 - 1/2 + 1/3 - 1/4 + \dots + (-1)^n/n + \dots,$$

In the succeeding parts of this chapter we will learn a core part of the C language which is sufficient for solving many mathematical problems. Small though this core is, we can use it to do some amazing computations, such as modeling the flow of heat in a long thin rod.

§ 1. Algorithms

Computations ranging from long division to extraction of square roots to numerical solution of partial differential equations can be carried out under the direction of an algorithm. Yet no matter how complicated the underlying mathematics, algorithms are based on a small number of elements artfully combined:

- *variables* to remember data,
- *loops* to perform a sequence of actions over and over again,
- *conditional* or *if-then-else* statements to make decisions

Here is a simple example which computes the average of two numbers X and Y :

Algorithm A: *find average of X and Y , leave result in Z*

- | | |
|-------------------------|---|
| 1: $Z \leftarrow X + Y$ | <i>Add X and Y, store in Z</i> |
| 2: $Z \leftarrow Z/2$ | <i>Replace Z by $Z/2$</i> |
| 3: Stop | |

Algorithm A is written in *pseudocode*: a stripped-down shorthand that expresses the core idea of the algorithm in an informal way. It uses just three memory cells, the “registers” X , Y , and Z . It uses just three instructions, each of which is of the form “stop” or “replace the contents of one memory cell by some function of the contents of other memory cells.” The latter idea is captured in the notation $V \leftarrow E$, which means “compute the expression E , then store the result in the variable V ”. The instructions are carried out in order: first instruction 1, then 2, then 3: stop. Note that an obedient clerk who can read pseudocode could execute the algorithm without having any idea what he was doing. He just follows instructions, line by line.

Computing Square Roots

The kind of straight-line execution featured in algorithm A is not typical. Usually there are instructions which change the “flow of execution.” Let us illustrate this with an algorithm for computing the square root of two. The idea is to begin with an approximate square root X , then to make it better. To do this we compute the quotient $Y = 2/X$. If X is smaller than the square root of two, then Y will be bigger, so we take the average of the two, $Z = (X + Y)/2$, for an even better approximation. For example, if $X = 1$, then $Y = 2$, so $Z = 3/2$. Now replace X by Z and repeat this process: $Y = 2/X = 4/3$, so $Z = (3/2 + 4/3)/2 = 17/12$. The first two values of Z so obtained, written in decimal notation, are 1.5 and 1.4166... You should calculate the next two values before going on.

The result of your computation is a sequence of numbers $Z_1 = 1.5$, $Z_2 = 1.4166\dots$, $Z_3 = ?$, $Z_4 = ?$, ... which approaches $\sqrt{2}$ in the limit. We now describe the algorithm which generates this series more formally:

Algorithm S: *Compute an approximation to $\sqrt{2}$*

- | | |
|--|--|
| 1: $X \leftarrow 1$ | <i>Put 1 in the register X</i> |
| 2: $Y \leftarrow 2/X$ | <i>Put $2/X$ in the register Y</i> |
| 3: $Z \leftarrow (X + Y)/2$ | <i>Put the mean of X and Y in Z</i> |
| 4: $X \leftarrow Z$ | <i>Put Z in X</i> |
| 5: If $ X - Y > \epsilon$, then go to line 2 | <i>Go back if not done</i> |
| 6: Stop | <i>X and Y differ by at most ϵ</i> |

To see that the algorithm does its job, we trace its execution, writing down the content of each register at each line of the program. Let's do this with $\epsilon = 0.003$:

Line	X	Y	Z	Remarks
1	1	*	*	X ← 1
2	1	2	*	Y ← 2/X
3	1	2	3/2	Z ← (X+Y)/2
4	3/2	2	2	X ← Z
5	3/2	2	2	Go to 2
2	3/2	4/3	2	Y ← 2/X
3	3/2	4/3	17/12	Z ← (X+Y)/2
4	17/12	4/3	17/12	X ← Z
5	17/12	4/3	17/12	Go to 2
2	17/12	24/17	17/12	Y ← 2/X
3	17/12	24/17	577/408	Z ← (X+Y)/2
4	577/408	24/17	577/408	X ← Z
5	577/408	24/17	577/408	0 < X-Y < 0.003
6	577/408	24/17	577/408	stop

Note that lines 2,3,4,5 form a *loop*. All that is needed to form a loop is an instruction of the form

if **condition**, then **P**

If the condition holds, **P** is executed. If it does not hold, then **P** is ignored. In our case the condition is $|X - Y| < \epsilon$, and the **P** clause is an instruction to jump from the current line to line 2. Thus, if the condition holds, we jump from line 5 to line 2. If it does not hold, we continue execution at line 6, where the program halts.

Note: Algorithm *S* can be simplified slightly, eliminating the variable *Z*:

- 1: $X \leftarrow 1$
- 2: $Y \leftarrow 2/X$
- 3: $X \leftarrow (X + Y)/2$
- 4: If $|X - Y| > \epsilon$, then go to line 2
- 5: Stop

Summing a series

The algorithm used for computing the square root of two can be easily modified to solve other problems. A common task is summing a series, e.g., computing the quantity

$$H(n) = 1 + 1/2 + 1/3 + \cdots + 1/n.$$

We can find the sum $H(n)$ as follows:

Algorithm H: *compute the sum $1 + 1/2 + \dots + 1/n$*

1: $i \leftarrow 0$	<i>Set up loop counter i</i>
2: $S \leftarrow 0$	<i>Put 0 in the register S</i>
3: $i \leftarrow i + 1$	<i>Increment the counter</i>
4: $S \leftarrow S + 1/i$	<i>Add the next term</i>
5: If $i < n$, then go to line 3	<i>Go back if not done</i>
6: Stop	

You should trace the execution of the algorithm for, say, $n = 3$, to check its operation. Note that lines 3,4,5 form a loop which is traversed n times. The final result is accumulated in the variable S , which starts off with value zero. The variable i controls the progress of the loop. As in algorithm S , the conditional instruction in line five is what forms the loop and gives the algorithm its power.

Factoring integers

Conditional instructions like “if A then B ” have uses other than creating loops. We illustrate this through the design of an algorithm which accepts a positive integer n and produces its factorization, say, by writing it on a paper tape. The basic idea is “trial division.” Set the trial divisor d to its lowest possible value. If d divides n , print d and factor n/d to obtain the remaining factors of n . If d does not divide n , replace d by $d + 1$ and try again. Stop when $n = 1$. (Try this on a few integers before reading on). Here is an example, a trace of our informal algorithm:

Problem: Factor 220	Output tape
-----	-----
2 divides 220, quotient 110	2
2 divides 110, quotient 55	2 2
2 does not divide 110	2 2
3 does not divide 110	2 2
4 does not divide 110	2 2
5 divides 110, quotient 11	5 2 2
5 does not divide 11	5 2 2
6 does not divide 11	5 2 2
7 does not divide 11	5 2 2
8 does not divide 11	5 2 2
9 does not divide 11	5 2 2
10 does not divide 11	5 2 2
11 divides 11, quotient 1	11 5 2 2
Stop: quotient is 1	11 5 2 2

In more formal terms, the factoring algorithm is as follows.

- 1) Set d , the trial divisor, to 2.

- 2) Compute the remainder r of division of n by d . (We denote this quantity by $n\%d$, and we read it as “ n modulo d ”).
- 3) If $r = 0$, then (a) print d on the tape and move it one unit to the right, (b) replace n by n/d . Otherwise, replace d by $d + 1$.
- 4) If $n = 1$, then there are no more factors to be found, so stop. Otherwise, go back to (2)

While it is not immediately clear that computation will come to a halt after finitely many steps with the factorization of n printed on the tape, this is in fact the case. (See if you can prove this). We now state our algorithm even more formally:

Algorithm F : factor an integer n

- 1: $d \leftarrow 2$
- 2: $r \leftarrow n\%d$
- 3: if $r = 0$ then
 - print d ; $n \leftarrow n/d$;
 - else $d \leftarrow d + 1$
- 4: if $n = 1$
 - then stop
 - else go to line 2

Note the form of statement three:

if **condition** then **P** else **Q**

The condition is “ $n = 1$.” It is either true or false. In the condition is true, then the **P** clause is executed. If the condition is false, then the **Q** clause is executed. In statement three, the **P** clause is *compound*: it consists of more than one instruction. Conditional statements like the one above can be nested. Take, for instance, the form

if **condition**
 then **P**
 else **Q**

and replace **Q** by “if **condition2** then **Q** else **R**” to obtain

if **condition**
 then **P**
 else if **condition2**
 then **Q**
 else **R**

The conditions are “boolean expressions:” expressions whose value is either true or false.

Below is a piece of pseudocode which is part of an improved factoring algorithm. It makes use of the observation that if n has a factor other than 1 or n , then it has a factor d such that $d^2 \leq n$. (See if you can prove this). We can use the observation just made to stop the computation much earlier. In the worst case scenario, when n is prime, roughly \sqrt{n} divisors instead of roughly n divisors need to be checked. This can make a big difference for large numbers. If n is about one million, then the original algorithm has to check about one million divisors, while the improved algorithm only has to check one thousand.

```
if  $r = 0$ 
  then print  $d$ ;  $n \leftarrow n/d$ ;
  else if  $d^2 > n$ 
    then print  $n$ ;  $n = 1$ 
    else  $d = d + 1$ 
```

The nested if-then-else statement above is constructed so that it can be substituted for item (3) in algorithm F . Note that if $d^2 > n$, then the only remaining factor in n is n itself — that is, n is prime. Thus we print n . However, we need to break out of the loop, so then we set $n = 1$ to “send a signal” to line four, where the decision to stop is made.

Exercises.

Design algorithms to compute the quantities listed below. Test your algorithm by making a trace.

1. $n! = 1 \cdot 2 \cdot 3 \cdots n$.
2. The sum $1 + 2 + 3 + \cdots + n$.
3. The sum $1 + 1/2 + 1/3 + \cdots + 1/n$.
4. The sum $1 - 1/2 + 1/3 - 1/4 + \cdots + (-1)^n/n$. This sum is an approximation to the natural logarithm of 2.
5. The sum $1 + 1/1! + 1/2! + 1/3! + \cdots + 1/n!$. This sum is an approximation to $e = \log 1$, where the logarithm is the natural one.
6. Rewrite the algorithm of the previous problem so that it stops when the difference between the n -th partial sum and the $n + 1$ -st partial sum is less than 0.001.

Additional exercises:

7. The writer of the algorithm below wanted to compute the prime factorization of an integer n . (a) Trace its execution for $n = 6, 7, 8$ and 12. The trace should show what is printed as well as the state of d . (b) Discuss the flaws in the algorithm. (c) What does the algorithm actually produce?

```
1:  $d \leftarrow 2$ 
```


- 2: if d divides n (without remainder) then print d
 3: $d \leftarrow d + 1$
 4: if $d \leq n$, then to to line 2
8. Check that algorithm F is correct by tracing its execution for some small values of n . Be sure to test all branches of the logic of the algorithm. Are other improvements in the algorithm possible?
 9. Use the remarks at the end of this section to write an improved algorithm F . Then trace its execution for various values of n to check that it works properly. Try $n = 2, 3, 4, 6, 7, 12$, and some others.
 10. Every positive integer n can be written uniquely as the product of powers of 2:

$$n = a_0 \cdot 1 + a_1 \cdot 2 + a_2 \cdot 2^2 + a_3 \cdot 2^3 + \dots$$

where a_i is either 0 or 1. The sequence

$$\dots a_3 a_2 a_1 a_0$$

is the *binary expansion* of n . To find the binary expansion of, say, $n = 55$, we compute as follows.

$$\begin{aligned} 55 &= 1 + 54 = 1 + 2(27) \\ &= 1 + 2(1 + 26) = 1 + 2 + 2(2 \cdot 13) \\ &= 1 + 2 + 4(1 + 12) = 1 + 2 + 4 + 4(2 \cdot 6) \\ &= 1 + 2 + 4 + 8(2 \cdot 3) = 1 + 2 + 4 + 16(3) \\ &= 1 + 2 + 4 + 16(1 + 2) \\ &= 1 + 2 + 4 + 16 + 32 \\ &= 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \end{aligned}$$

Thus the binary expansion of 55 is 110111. We can summarize what we did as follows. Divide n by 2 and record the remainder (0 or 1). Then do the same to the quotient. Repeat until the quotient is 1. Thus we have the following trace:

$55 = 2 \times 27 + 1$	$a_0 = 1$
$27 = 2 \times 13 + 1$	$a_1 = 1$
$13 = 2(6) + 1$	$a_2 = 1$
$6 = 2(3) + 0$	$a_3 = 0$
$3 = 2(1) + 1$	$a_4 = 1$
$1 = 2(0) + 1$	$a_5 = 1$

Use the above procedure to compute the binary expansion of $n = 85$, $n = 127$, and $n = 138$. Then write an algorithm in the style of A and F that does the same thing. Trace its execution to test that it is correct.

11. Modify the design of the algorithm in the previous exercise so that the “paper tape” output represents n as a sum of powers of two. Thus, for the input $n = 55$, the output should be

$$55 = 1 + 2 + 4 + 16 + 32$$

Trace for $n = 85, 127, 138$ to test the algorithm.

§ 2. C Programs

In the last section we learned to read, design, and write algorithms in a kind of pseudocode that is very close to machine language. We'll now learn to express computational ideas in pseudocode that is closer to C, and then to transform the results into C programs. Let us begin by reconsidering the simplified version of algorithm S for computing the square root of two:

Algorithm S :

- 1: $X \leftarrow 1$
- 2: $Y \leftarrow 2/X$
- 3: $X \leftarrow (X + Y)/2$
- 4: If $|X - Y| > \epsilon$, then go to line 2
- 5: Stop

We can express the same computational idea as follows:

Algorithm S' :

```
 $X \leftarrow 1$ 
do
   $Y \leftarrow 2/X$ 
   $X \leftarrow (X + Y)/2$ 
while  $|X - Y| > \epsilon$ 
```

The first statement in S' is the same as in S . The part between “do” and “while” is the same as lines two and three of S . This is the *body* of a *do-while* loop: each statement in the body is executed, then the condition $|X - Y| > 0$ is checked. If it is true, the loop body is executed again. And so on, until the condition $|X - Y| > 0$ is false. The general form of a do-while loop is as follows, where the A_i state some action to be performed:

```
do
   $A_1$ 
   $A_2$ 
  ...
   $A_n$ 
while condition
```

As advertised, the kind of pseudocode used in algorithm S' is very close to code written in C. Indeed, compare it to the following:

Algorithm S'' :

```
 $X = 1;$ 
do {
   $Y = 2/X;$ 
```

```

    X = (X + Y)/2;
} while ( fabs(X - Y) > epsilon );

```

We have written *fabs* instead of *abs* because in C the former is the correct name of the function which computes the absolute value of so-called *floating point* numbers — the numbers used to represent real numbers. Algorithm *S''* expresses the same computational idea as does algorithm *S'*. It is also a grammatically correct fragment of a valid C program. Such a program, `root.c`, is listed below. It is something you can compile and run on your computer.

```

/* root.c: compute an approximate square root of two */

#include <stdio.h>
#include <math.h>

main()
{
    float X, Y, epsilon;
    printf("Enter epsilon: "); scanf("%f", &epsilon);

    X = 1;
    do {
        Y = 2/X;
        X = (X+Y)/2;
    } while ( fabs(X-Y) > epsilon );

    printf("Approximate square root of 2 = %f\n", X);
}

```

Let us examine the above program line by line. This close reading will give us a basis for writing our own programs. The first line, enclosed by `/*` and `*/`, is a comment. It contains the name and purpose of the program. The next line, `#include <stdio.h>`, tells the C compiler to use the standard input and output library, used for reading and writing data. The line, `#include <math.h>`, plays a similar role: the C math library, which contains the function `fabs` must be used. (When using `math.h` on a unix system, compile with `cc root.c -lm`. The option `lm` stands for “load math.”) Next comes the the function `main`, an obligatory part of every C program. The left brace “{” is the start of the body of the function `main`. It will be matched by a “}” at the end of the program.

The body of `main` consists of three paragraphs. In the first paragraph the variables `X`, `Y`, and `epsilon` are declared. All variables in C have a *type*, e.g., `int` for smallish integers like ± 123456 and `float` for “floating point” numbers like ± 123.456 . There are limitations on the size of `int`’s and `float`’s and also on the precision of `float`’s; these can vary from machine to machine. Note that

program statements end with a semicolon. The `printf` function displays text on the screen. The `scanf` function reads a `float` into the variable `epsilon`. The text `"%f"` tells `scanf` that it should expect a floating point number, not something else. The prefix `&` is a C-ism used for variables into which we store data.

The second paragraph of `main` is our algorithm, consisting almost entirely of a `do-while` loop. Note the use of `fabs`, the floating point absolute value function. There is also `abs`, but it is for `int`'s (and is part of `stdlib.h`, not `math.h`). The third paragraph consists of a single line, which displays the result of the computation. The `"\n"` is a "carriage return:" whatever is printed next will appear on a new line. This terminology is of ancient origin, and refers to the forgotten technology of typewriters.

Note that the C program has about three times as many lines as does the pseudoprogram. Such expansion is typical in designing a program that fully embodies an algorithm. Note also the general structure of our program:

```
/* comments */
< include statements: stdio.h, etc. >
main()
{
    < Declaration of variables used >
    < Program body >
}
```

You can use it as a template the programs you will write in the exercises below.

Summing a series

For more practice turning algorithms into C programs, let us consider the problem of summing the so-called harmonic series

$$1 + 1/2 + 1/3 + 1/4 + \dots$$

treated in problem four of the last section. This series is known to *diverge*. Recall what this means. If

$$H(n) = 1 + 1/2 + 1/3 + 1/4 + \dots + 1/n$$

is the n -th partial sum, then for any $c > 0$ there is an n such that $H(n) > c$. In other words, we can make $H(n)$ as large as we want by making n large enough. (Try find an elementary proof of this fact; it was known to Bishop Nicole Oresme in the fourteenth century).

Here is an algorithm for computing the n -th partial sum:

```
1:  $i \leftarrow 0$ 
2:  $S \leftarrow 0$ 
3:  $i \leftarrow i + 1$ 
4:  $S \leftarrow S + 1/i$ 
5: If  $i < n$ , then go to line 3
5: Stop
```

Written as a do-while loop it is

```
 $i \leftarrow 0$ 
 $S \leftarrow 0$ 
do
     $i \leftarrow i + 1$ 
     $S \leftarrow S + 1/i$ 
while  $i < n$ 
```

The corresponding C fragment is

```
i = 0;
S = 0;
do {
    i = i + 1;
    S = S + 1.0/i;
} while ( i < n );
```

We house it inside a C program:

```
/* h.c: compute the n-th partial sum of the harmonic series */

#include <stdio.h>

main() {

    int i, n;
    float S;

    printf( "Enter n: ");
    scanf( "%d", &n );

    i = 0;
    S = 0;
    do {
        i = i + 1;
        S = S + 1.0/i;
    } while ( i < n );

    printf( "H(%d) = %f\n", n, S );
}
```

Several points are worthy of note:

1. To read an `int` and store it in the variable `n`, we use `scanf("%d", &n)`. The `"%d"` is the “format string” for a (decimal) integer like 12345. Recall that we must reference `n` as `&n` in `scanf`. (See Kernighan and Ritchie for an explanation).
2. We say “`1.0/i`,” not “`1/i`.” When $i = 2$ the first expression has 0.5, a `float` as its value, while the value of the second is 0, an `int`. The first expression is (approximately) the quotient of real numbers. The second is the quotient of integers, i.e., the quotient as computed by long division. Thus $7.0/2$ is 3.5 whereas $7/2$ is 3.
3. The `printf` statement prints an `int` and a `float` using `%d` and `%f` respectively.

Factoring integers

For still more practice turning algorithms into C programs, let us consider the problem of factoring an integer into primes. Our original algorithm was as follows:

```

1:  $d \leftarrow 2$ 
2:  $r \leftarrow n \% d$ 
3: if  $r = 0$  then
    print  $d$ ;  $n \leftarrow n/d$ ;
    else  $d \leftarrow d + 1$ 
4: if  $n = 1$ 
    then stop
    else go to line 2

```

We reformulate it using a do-while loop:

```

 $d \leftarrow 2$ 
do
     $r \leftarrow n \% d$ 
    if  $r = 0$  then
        print  $d$ ;  $n \leftarrow n/d$ ;
        else  $d \leftarrow d + 1$ 
while  $n > 1$ 

```

Then we write the corresponding C program:

```

/* factor.c: print factorization of n */

#include <stdio.h>

main()
{
    int n, d, r;
    printf("Enter n: "); scanf("%d", &n);

    d = 2;
    do {
        r = n % d;
        if ( r == 0 )
            { printf("%d ", d) ; n = n/d; }
        else d = d + 1;
    } while ( n > 1 );

    printf( "\n" );
}

```

Note that we test for equality in C by an expression of the form `a == b`. To test the condition $a \geq b$, use `a >= b`, and to test $a \neq b$ use `a != b`. To understand the role of the last `printf` statement, try removing it and running the program.

Notes and hints for the exercises

1. As noted above, C has various *data types*, e.g., integer (`int`) and floating point (`float`). Use

```
int i, j;
```

to declare integer variables `i`, `j`. Use

```
float x, y;
```

to declare floating point variables `i`, `j`. Subsequently you can say, for example, `i = 7, j = -4, x = 3.1416`.

2. The type of an expression in C is determined by the type of the numbers appearing in it. Thus `1.0/2` has the value `0.5`, a `float`, while `1/2` has the value `0`, an `int`. Thus `1/2` is the *integer quotient* of 1 by 2: the quotient of 2 by 1 under long division is 0, with remainder 1. Likewise, `7.0/2` is `3.5`, while `7/2` is `3`.
3. Examples of how to print an `int`:

```
printf("i = %d, j = %6d\n", i, j );
```

Thus `i` is printed as a decimal integer, and `j` is printed in a space at least 6 characters wide.

4. Examples of how to print a `float`:

```
printf("x = %f, y = %6.2f\n", x, y );
```

Thus `x` is printed as a floating point number, and `y` is printed in as a floating point number in a space at least 6 characters wide, with two characters after the decimal point. The statement

```
printf("x = %6f, y = %.2f\n", x, y );
```

prints `x` in a space 6 characters wide, and `y` is printed with at least two characters after the decimal point.

5. The statements for reading values into variables follow the same syntax, e.g.,

```
scanf( "%d", &i ); scanf( "%f", &x );
```

6. To test whether a number `i` is even, use the expression `(i%2 == 0)`. The expression `i%2`, read as “`i mod 2`,” is the remainder of `i` upon division by 2, and so is 0 or 1. Thus `4%11` is 4 while `70%11` is 4. Modular arithmetic was invented and studied by Carl Friedrich Gauss (1777–1855).

Exercises.

1. Write and run the shortest possible C program (it does nothing). How many characters does it contain? Write and run a C program which displays a one-line message, “Hello world.” Write and run a C program which displays the message “Hello world” one hundred times.
2. Write a C program which implements each of the algorithms in exercises 1 – 6 of the previous section. Use the models established in `root.c` and `factor.c`, and refer to the template for overall structure. The notes and hints above give some information about C that you will need.
3. Write a C program which implements the factoring algorithm F of the previous section. Try factoring the numbers 12, 123, 1234, etc. Try factoring numbers such as 123456, 123457, 123458, etc. At some point you will have trouble with large numbers because `int`'s have a certain maximum size. A partial remedy for this difficulty is to use the `long` data type. These are just bigger `int`'s.
4. Write a C program which displays the binary expansion of a positive integer given by the user. Then write a second version which prints out the representation of an integer as a sum of powers of two, e.g., $55 = 1+2+3+4+16+32$.
5. The Babylonian scribes of 1700 BC apparently knew algorithm *S* for computing square roots or one similar to it. One clay tablet (YBC 7289) bearing an inscription of an isosceles right triangle with unit base and altitude gives the following value for $\sqrt{2}$: 1;26,51,10. Since the Babylonians used a sexagesimal system (base 60), this value should be interpreted as

$$1 + \frac{26}{60} + \frac{51}{60^2} + \frac{10}{60^3}$$

The sexagesimal system still survives in our system for measuring time and angles. This is discussed in (Aaboe, ch 1), (Gericke, p 12).

6. Modify the previous program so that it displays the number of times it executes the do-while loop.
7. Develop algorithms and programs for converting between Babylonian and decimal notation. Examples: (a) $1;30,10 = 1 + 30 \cdot 60^{-1} + 10 \cdot 60^{-2}$, (b) $13,9,23 = 13 \cdot 60^2 + 9 \cdot 60^1 + 23 \cdot 60^0$. The semicolon is the “sexagesimal decimal point.” Note that we would write 3 hours, 11 minutes, and 25 seconds as 3;11,25. What number of hours is this (decimal notation)?

§ 3. Loops

Loops are such an important part of computation that we will look at them in more detail. So far we have expressed them in the *do-while* form. There are two others that have a counterpart in C. The first is the *while-do* loop: the condition is tested before the loop body is executed instead of after, as in the do-while loop. We illustrate this by considering again the problem of computing the quantity

$$H(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$$

The algorithm built from a do-while loop was as follows:

```

i ← 0
S ← 0
do
    i ← i + 1
    S ← S + 1/i
while i < n

```

The while-do loop reads like this:

```

i ← 1
S ← 0
while i ≤ n do
    S ← S + 1/i
    i ← i + 1

```

Trace the execution of each algorithm for $n = 3$ to compare their operation. The corresponding algorithms in C are

```

i = 0;
S = 0;
do {
    i = i + 1;
    S = S + 1.0/i; }
while ( i < n );

```

and

```

i = 1;
S = 0;
while ( i <= n ) do {
    S = S + 1.0/i;
    i = i + 1;
};

```

For loops

The other kind of loop is called a *for loop*. It is used only when the number of times the loop body will be executed is known in advance, and it has the structure

```

for i from 1 to n do
    S1
    S2
    ...
    Sk

```

The statements S_1 through S_k are executed in order with $i = 1$, then with $i = 2$, and so on, up through $i = n$. In our low level pseudocode we could write this as

```
0:  $i \leftarrow 1$ 
1:  $S_1$ 
2:  $S_2$ 
...
 $k$ :  $S_k$ 
 $k + 1$ : if  $i \leq n$  then go to line 0
```

Sums like $H(n)$ are easy to compute with a for loop:

```
for i from 1 to  $n$  do
     $S \leftarrow S + 1/i$ 
```

The corresponding C code looks like this:

```
for ( i = 1; i <= n; i++ )
    S = S + 1.0/i;
```

The expression $i=1$ sets the loop counter, $i <= n$ checks it against the upper limit, and $i++$ increases its value by one. If we wanted to print out the partial sums as they are built up, we could write this:

```
for ( i = 1; i <= n; i++ ) {
    S = S + 1.0/i;
    printf( "H(%d) = %6.4f\n", i, S);
}
```

In this example the loop body is a so-called *compound statement*: a sequence of statements surrounded by curly braces.

A real C programmer would write the above loop more tersely (and more cryptically) as

```
for( i=1; i <= n; i++ )
    S += 1.0/i ;
```

The operator $+=$ means “replace the left-hand-side by the left-hand-side plus the right-hand-side.” Related operators are $-=$, $*=$, $/=$, and $%=$.

To remind ourselves of how easy it to write the actual C program once the algorithm is written, we write a C program for computing $H(100)$. This program uses all the little C-isms we have learned:

```

/* sum.c: sum n terms of the harmonic series */

#include <stdio.h>

main()
{
    int i, n;
    float S;

    n = 100;
    S = 0.0;

    for( i=1; i <= n; i++ )
        S += 1.0/i;

    printf( "H(%d) = %f\n", n, S );
}

```

Note. A for loop can always be replaced by a do-while loop. For instance, “for i from 1 to n do P ” can be replaced by

```

i ← 0
do
    i ← i + 1
    P
while i < n

```

Can a do-while loop always be replaced by a while-do loop? Or vice versa?

Studying the harmonic series

Let us consider the problem “given $c > 0$, what is the least value of n such that $H(n) > c$?” This is a problem that we can solve with a while-do loop:

```

Algorithm HH
i ← 0
S ← 0
while S ≤ c do
    i ← i + 1
    S ← S + 1/i
print i

```

This problem is *not* one we can solve with a for loop, since we do not know in advance how many times the loop body will be executed. Indeed, determining that number is the purpose of the computation. In the exercises below, you will be asked to turn this algorithm into a C program and to use it to study the harmonic series.

Computing the natural logarithm of 2

Another problem we can solve with the while-do loop is to compute

$$\log 2 = 1 - 1/2 + 1/3 - 1/4 + \cdots + (-1)^n/n + \cdots$$

to given tolerance ϵ . Here is the algorithm:

```
Algorithm L
oddSum  $\leftarrow$  1
evenSum  $\leftarrow$  oddSum - 1/2
i  $\leftarrow$  2
while oddSum - evenSum >  $\epsilon$  do
    i  $\leftarrow$  i + 1
    oddSum  $\leftarrow$  evenSum + 1/i
    i  $\leftarrow$  i + 1
    evenSum  $\leftarrow$  oddSum - 1/i
```

In the exercises below, you will be asked to turn this algorithm into a C program.

Exercises.

1. Rewrite `sum.c` so that it accepts the value of n from the user. Then use your program to investigate how $H(n)$ grows as n grows.
2. Rewrite `sum.c` using a while-do loop.
3. Turn algorithm *HH* into a C program.
4. How many terms are needed for $H(n)$ to exceed 5? How many for it to exceed 10? Why does algorithm *HH* terminate after finitely many steps given any $c > 0$? Does a C program implementing algorithm *HH* have the same behavior?
5. Turn algorithm *L* into a program for computing $\log 2$. Why does the program give approximations of $\log 2$ which are guaranteed to have accuracy at least ϵ ?
6. The series

$$u - \frac{u^2}{2} + \frac{u^3}{3} - \frac{u^4}{4} \pm \cdots$$

converges to $\ln(1 + u)$ for u in the interval $-1 < u \leq 1$. Use this fact to design a program which computes $\ln x$ for x in the interval $0 < x \leq 2$. Your program should read in a real number x and an integer n . Then it should compute the n -th partial sum of the series and print results in the form

$$\ln(2) = 0.693095$$

corresponding to the input $x=2$ and $n = 10000$. Modify your program to compute a table of logarithms for a fixed choice of n . Your output should look like this:

x	ln(x)
0.2	-1.609438
0.4	
0.6	.
0.8	.
1.0	.
1.2	.
1.4	.
1.6	.
1.8	
2.0	0.693095

Make a table for $n = 10$, $n = 100$, $n = 1000$, and for $n = 10000$. Discuss the accuracy of your results.

7. Modify your program from the previous exercise so that it stops automatically when your results are accurate to four decimal places. Again make a table as above, but now add a column that tells how many terms were necessary to achieve this accuracy for the given value of x .
8. Using a for loop write a program to compute n factorial. Refer to the algorithm you wrote in exercise 1 of section 1. Recast it as a for loop, then write the C program. Finally, modify your program so that it will produce a table of factorials for $n = 0$ to $n = 20$. Write one version using `float` variables and another using `int`. Check your results carefully. How accurate are they? In both versions your output should look like:

n	n!
0	1
1	1
2	2
3	6
.	.
.	.
.	.

You may find it useful to use `\t` in your print statements — this is similar to `\n`, but produces a tab instead of a new line.

9. It often happens that to solve a problem one needs a loop inside a loop. The following example from number theory is such a problem.

The system of integers modulo 11, written $\mathbf{Z}/11$, is a finite algebraic system consisting of the symbols 0, 1, 2, ... 10. Addition of a and b is defined by the rule “add a and b , then divide by 11 and take the remainder.” Thus $5 + 9 \equiv 3 \pmod{11}$. Multiplication is defined by the rule “multiply a and b , then divide by 11 and take the remainder.” Thus $2 \cdot 9 \equiv 7 \pmod{11}$.

Recall that in C the expression $a \% b$ gives the remainder gotten by dividing b into a . For example, if $a=7$ and $b=2$ then $a \% b = 1$, since we get a remainder of 1 when we divide 7 by 2. **Note:** Here a and b must be integer variables — the operator $\%$ cannot be used with float variables.

Write a program to print out an addition table for $\mathbf{Z}/11$. Its output should look like:

```

+  0  1  2  3  4  5  6  7  8  9 10
0  0  1  2  3  ...
1  1  2  3  4  ...
2  2  3  4  5  ...
3  3  4  5  6  ...
.
.
.
10 10  0  1  2  ...

```

For your program you will need several loops, including a loop inside a loop. Use the rough outline below as a guide.

```

loop to print the first row, a list of the integers mod 11

for i from 0 to 10      /* Loop through the rows */
  for j from 0 to 10    /* Loop across the columns */
    print i+j mod 11
  move to next line

```

10. Modify your work in the previous exercise to print out a multiplication table modulo 11.
11. A number b is a multiplicative inverse of a modulo 11 if $ab = 1 \pmod{11}$. Devise a program which find the inverse modulo 11 of a number a provided by the user. Then modify your program to print a table of inverses mod 11.
12. A number a is called a *quadratic residue mod 11* if there is a number b such that $b^2 = a \pmod{11}$. In other words, quadratic residues are squares mod 11. Write a program which lists the quadratic residues modulo 11.
13. Modify the previous program so that it lists the quadratic residues mod p for any prime p . Study the results for $p = 23$. For large values of p it is not practical to list tables. Modify your program so that it accepts an integer a , a prime p , and which determines whether a is a square modulo p .

§ 4. Control structures: review and comments

In the last three sections we learned to express computational ideas using statements which assign values to variables, which make decisions (if-then-else statements), and which carry out repeated actions (loops). All our statements were of a few simple forms:

1. $v \leftarrow E$
- 2a. if B then P
- b. if B then P else Q
- 3a. while B do P
- b. do P while A
- c. for i from a to b do P

Statements of type (2) and (3) change the order in which the execution of other statements is carried out and so are called *control structures*.

It turns out that these forms (together with the statement “stop”) are sufficient to express any computational idea. In fact, we could make do with the forms 1, 2b, and 3a. We have already seen a hint of this: a for loop can be written as a while-do loop. Moreover, the “substitution rule” allows us to build very elaborate structures. This rule states that any clause P or Q can be any kind of statement — of type 1 through 3, or a *compound statement* $\{S_1, S_2, \dots, S_n\}$ obtained by stringing together other statements. To see how the substitution rule works, start with the form

```
for i from 1 to n do P
```

For P substitute the compound statement for j from 1 to n do Q to get

```
for i from 1 to n do
  for j from 1 to n do Q
```

for Q substitute for k from 1 to n do $\{A; B\}$ to get

```
for i from 1 to n do
  for j from 1 to n do
    for k from 1 to n do  $\{A; B\}$ 
```


For A substitute `if (x > 0) then P else Q` and for B substitute `if (y > 0) then R else S` to get

```
for i from 1 to n do
  for j from 1 to n do
    for j from 1 to n do {
      if ( x > 0 ) then P else Q;
      if ( y > 0 ) then R else S;
    }
}
```

When you build complicated structures, start from a simple form and construct it as we did above, using the substitution rule. Structures at least this complicated are routine, as we will see later. (For examples, see the programs `search0.c`, `search1.c`, and `search2.c` in the section on functions).

Switch

Although the above forms have enough power to express any computational idea, there is one other form which is convenient. This is the `switch` statement, which makes decisions based on a list of conditions. Its form is

```
switch( expression ) {
  case A: statements
  case B: statements
  etc.
  default: statements
}
```

The `default` clause is optional; A, B, etc. are integer or character constants. A character constant is a single letter or symbol, like `a`, `z`, `8`, `#`, etc. The `expression` must take integer or character values.

To illustrate how `switch` is used, we will design a program that computes a letter grade A, B, C, D, or E from a sequence of scores entered by the user. The heart of the program is the C fragment listed below. We assume that `average` is a `float` which holds the grade average, assumed to be a number between 0 and 100. We divide the average by 10 and chop off the fractional part, converting the result into an `int`, which we store in `g`. This change of data type is done by an expression of the form `y = (int) x`, where `x` is a `float` and `y` is an `int`. With `g` taking a value between 0 and 10, the `switch` statement selects the statement to be executed. Thus, if `average = 88.3`, then `g = 8`, and so case 8 is executed,

setting `g` to `'B'`. The `printf` statement produces the message `Average = 88.30, grade = B.`"

```
g = (int) (average/10);
switch( g ){
    case 10: grade = 'A'; break;
    case 9: grade = 'A'; break;
    case 8: grade = 'B'; break;
    case 7: grade = 'C'; break;
    case 6: grade = 'D'; break;
    default: grade = 'E'; break;
}
printf( "Average = %.2f, grade = %c\n", average, grade);
```

This C fragment is part of the program below. Note that we use `char grade` to declare `grade` as a character variable, and that we use a `do-while` loop to read in the data. The user enters a negative number — an impossible grade — to halt the input process and compute the grade. Note that a program, like any piece of text, can (and should) be divided into paragraphs that express a common theme.

```

/* grade.c: compute letter grade from scores entered */

#include<stdio.h>

main()
{
    int score, count, g;
    float sum, average;
    char grade;

    printf("Enter scores,\n");
    printf("Enter a negative number to get total, average, and grade\n\n");

    /* Read in data and compute average */
    sum = 0.0;
    count = 0;
    do {
        printf( "score(%d) = ", count + 1);
        scanf( "%d", &score );
        if ( score >= 0 ) {
            count++;
            sum += score;
        }
    } while ( score >= 0 );

    average = sum/count;

    /* Compute and display letter grade */
    g = (int) (average/10);
    switch( g ){
        case 10: grade = 'A'; break;
        case 9: grade = 'A'; break;
        case 8: grade = 'B'; break;
        case 7: grade = 'C'; break;
        case 6: grade = 'D'; break;
        default: grade = 'E'; break;
    }
    printf( "Average = %.2f, grade = %c\n", average, grade);
}

```

In the program above it is crucial that T be declared as an integer variable. For example, if the student's average is 78.34, then `average/10` will be 7.834 and when we assign this value to the integer variable T, the fractional part is thrown

away and T is assigned the value 7. Thus any average between 70.00 and 79.99 will give T=7 and a letter grade of C.

As a second application of the switch statement we design a rudimentary four-function calculator, `calc.c`. The program presents the user with a prompt `>` to which he or she can respond by statements like `12 + 6`, `6*9.3`, etc.

```
/* calc.c: rudimentary calculator */

#include<stdio.h>

main()
{
    float a, b, c;
    char op;          /* the operation to be performed */

    for( ;; ){

        printf( "> ");
        scanf( "%f %c %f", &a, &op, &b );

        switch( op ) {
            case '+': c = a + b; break;
            case '-': c = a - b; break;
            case '*': c = a * b; break;
            case '/': c = a / b; break;
        }
        printf( "    %f\n", c );
    }
}
```

All legitimate inputs to `calc.c` have the form `a op b`, where `a` and `b` are floating point numbers, and where `op` is one of the symbols `+`, `-`, `*`, `/`. The program is built around a loop that begins with the phrase `for(;;)`. This is a convenient way of writing an infinite loop — one that never terminates. Each time the loop body is executed, the prompt is displayed to the user, who types some (legal?) input which is captured by `scanf`. Note once again the mandatory symbol `&` which precedes the variables into which data is to be stored. The variable `op` has the character data type, which is really a short integer. This is why it can be used to control the `switch` statement, which selects the computation to be performed. Finally, the result of this computation is displayed to the user.

Computing sines

As a final example of the switch statement, we compute the sine of x using the series

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \mp \dots$$

We give just the algorithm at the core of the program:

```
S = 0;
p = 1;
d = 1;
for( i=1; i<=n; i++ ) {
    p = p*x;
    d = d*i;
    ii = i%4;
    switch( ii ) {
        case 0: break;
        case 1: S = S + p/d; break;
        case 2: break;
        case 3: S = S - p/d; break;
    }
}
```

Here is a close reading of the algorithm. First, a variable S is set to zero. In this variable we accumulate the sum which computes the sine. Next we set up variables p , in which we accumulate the product x^i , and d , in which we accumulate the denominator $i!$. The variable ii will hold $i\%4$, the remainder of i upon division by 4. Recall that this expression is pronounced “ i mod 4.” These remainders take on the values 0, 1, 2, 3. (Stop to compute $i\%4$ for $i = 0, 1, 2, 3, 4, 5, 6, 7, 8$).

The heart of the algorithm is the for loop. First p , d , and ii are updated, so that $p = x^i$, $d = i!$. Then S is updated. If ii is 0 or 4, nothing is done. If ii is 1, then $p/d = x^i/i!$ is added to S . If ii is 3, then $p/d = x^i/i!$ is subtracted from S .

Below you will be asked to write a C program based on the above algorithm.

Exercises.

1. Write a program using `switch` that will print the season corresponding to a given month, where the month is represented by an integer in the range 1–12. For example, if the user types 12, corresponding to the month of December, then the program should give the output `Winter`. (**Hint:** Notice that if n is any integer in the range 1–12, then $n \% 12$ will be in the range 0–11, since 12 is converted to 0 by this operator and all the other integers are left alone. Then dividing by 3 will give the corresponding season.)
2. (Continuation) Rewrite the program of the previous exercise using nested `if`'s instead of `switch`.
3. Write a program to compute $\sin x$ for given x . The user should supply x and a positive integer n , and the computation should use all terms in the series up through the term involving x^n . Test your program by computing $\sin 1$, where the unit of angular measure is the radian.
4. (Continuation) Rewrite the previous program so that it computes the cosine of x . Use the series

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} \pm \dots$$

§ 5. Computer Arithmetic and Error Analysis

Computer computations with real or complex numbers are subject to two sources of error: *truncation error* and *roundoff error*. To understand these, consider the problem of computing the natural logarithm of two, given by the series

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} \mp \dots$$

We can only add up finitely many terms of the series, ignoring all the rest. Thus we could add just four terms, obtaining the approximation

$$\ln 2 \approx 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} = \frac{12 - 6 + 4 - 3}{12} = \frac{7}{12}$$

The error in $7/12$ as an approximation to $\ln 2$ is the tail of the infinite series:

$$\frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} \pm \dots$$

A theorem tells us that for such a series with alternating signs and decreasing terms, the error committed is no greater than the magnitude of the first term omitted. Thus $\ln 2$ is $7/12$ to within an error of $1/6$. Obviously we can improve our estimate for $\ln 2$ by taking more terms. Thus

$$\ln 2 \approx 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} + \frac{1}{9} - \frac{1}{10} = \frac{1627}{2520}$$

is accurate to within one part in eleven. This error, stemming from the fact that we can only do a finite amount of computation, is truncation error. It is unavoidable in a problem such as this.

The second source of error stems from the fact that only a finite amount of information can be stored in computer memory: a fixed number of bits is used to represent each `float` variable, and likewise for any other data type (`doubles`, `int`'s, etc.). Neither of the approximations computed above can be expressed exactly using finitely many digits:

$$\frac{7}{12} = 0.58\bar{3} \approx 0.583333 \quad \text{and} \quad \frac{1627}{2520} = 0.645\overline{634920} \approx 0.645635.$$

Thus, when we represent numbers as `float` variables, we introduce *round-off error*, the error introduced by truncating decimal expansions of real numbers.

We have minimized round-off error in the computation above by using exact values for all intermediate steps. Although we could write a program to sum the series algebraically (and you are asked to do so in Exercise 2 below), it is standard practice to use `float` (or `double`) variables throughout. This introduces additional round-off error. For instance, if we compute the first three terms $1 - 1/2 + 1/3$ using only 6 digits after the decimal point, then our values for 1 and for $1/2$ will be completely accurate, but we will use the *approximation* 0.333333 in place of the infinite decimal expansion of $1/3$. Taking more terms in the series approximation to $\ln 2$, we encounter more fractions with infinite decimal expansions. The error arising from finite-precision approximations to fractions like $1/3$ or $1/7$ can, in the long run, produce significant error in our estimate for $\ln 2$.

To get some idea how round-off error can be important and difficult to estimate, suppose we add 10,000 real numbers that are reliable in the first 5 digits, but are questionable in the 6th digit. More precisely, suppose the numbers we add have a possible error of $\pm 10^{-6}$. In the worst case, assuming that all the numbers are too big in the sixth digit, the total error will be $10^4 \cdot 10^{-6} = 10^{-2}$. That is, errors in the *sixth* digit produce an incorrect *second* digit. This is the worst possible case. It is more realistic to expect that the numbers added are sometimes too small in the 6th digit and sometimes too big. If these possibilities are equally likely, then the accumulated error will be smaller. See Exercises 3-4 in §6 below for some numerical experiments. For a theoretical treatment, we call upon some statistics: if the error in the sixth digit is uniformly distributed, then the error in the sum of N numbers will be on the order of $\sigma\sqrt{N}$, where $\sigma \approx 5.5$ is the standard deviation of a uniformly distributed random variable which takes the integer values in the range from -9 to $+9$. Thus the expected error in summing 10^4 numbers will be about $5.5 \times 10^2 \times 10^{-6} = 5.5 \times 10^{-4}$. Thus the error in the case of randomly distributed case is indeed smaller, though still significant.

In general, round-off error is difficult to analyze and we will rarely attempt to do so. However, it can introduce unexpected and hard-to-detect errors in your programs, as we will see in computing arc length in Chapter 2. Thus, we must keep it in mind when evaluating the performance of a program that computes a numerical approximation.

In any implementation of C, there is a machine-dependent limit on the size of the numbers we can work with. If a variable is declared to be of integer type, then there is a finite range of allowed integers. (This range can be determined experimentally — see Exercise 5 in Section 2 of Chapter 1.) Similarly, since we keep track of only a finite number of digits in a float variable, there are again serious limitations on the numbers we work with. In particular, there is a smallest difference that we can detect. If we keep track of only six digits and compare two quantities that differ only in the seventh decimal place, then to us they look equal.

The upshot is that if we work with numbers that are too big or too small, depending on the version of C and the machine at hand, serious inaccuracies may appear. Intermediate values that are out of the allowed range may produce an error message, or they may produce nonsensical results.

Such considerations will affect the design of computer programs. For instance, to compute a quantity like $x^n/n!$, where x is a real number larger than 1, we might use a loop in which we compute x^n and $n!$, taking the quotient when the loop is finished.

```

Get  $x$  and  $n$  from the user
 $p \leftarrow 1$            Initialize  $p$ 
 $f \leftarrow 1$        Initialize  $f$ 
for  $i$  from 1 to  $n$  do
     $p \leftarrow p * x$     $p$  stores  $x^i$ 
     $f \leftarrow f * i$     $f$  stores  $i!$ 
 $t \leftarrow p/f$         $t$  stores  $x^n/n!$ 

```

In above loop $f = i!$ grows rapidly as i increases. If $x > 1$, then p grows rapidly as well, and the computation will fail. On the other hand, if $x < 1$, then the quotient $x^n/n!$ tends toward zero as n grows without bound. A better program design, one that avoids intermediate results that are possibly too large, is

```

Get  $x$  and  $n$  from the user
 $t \leftarrow 1$            Initialize  $t$ 
for  $i$  from 1 to  $n$ 
     $t \leftarrow t * x/i$ 

```


At each step in the loop we multiply by x . If $x > 1$, this makes t grow, but we also divide by i , which makes t small. Thus we keep some control over the size of intermediate results.

Double Precision Arithmetic

So far we have worked with the variable type `int` for integer arithmetic and we have used `float` variables to work with real numbers. There is another option available to us, namely `double` variables. These are similar to `float`, but we keep track of twice as many digits. Computations are more accurate, but they require more time and memory. A double precision version of `sum.c` would look like:

```
/* sum.c: with double precision */

#include <stdio.h>

main()
{
    int i, n;
    double sum;

    /* Read in an integer and store it as n */

    printf("Enter n: ");
    scanf("%d", &n);

    sum = 0.0;

    for( i=1; i <= n; i++ )
        sum = sum + 1.0/i;

    printf( "H(%d) = %15.12lf\n", n, sum );
}
```

Note that we use the placeholder `%lf` (for “long float”) when we read in a value for a double precision variable and when we print such values.

Exercises.

1. Modify your factorial program from Exercise 6 in §2 to use double precision variables. How accurate are the values you obtain compared to those obtained using integer variables? using single precision or float variables?
2. Write a program that sums the series

$$\ln 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} \mp \cdots \pm \frac{1}{n}$$

using integer variables throughout. Your program should give the output 14/24 for the input $n = 4$. What is the largest value of n for which you get accurate results?

3. (For Maple users) To see the effect of round-off error in computing the partial sums of the series approximation to $\ln 2$, try the following:

```
[> n := 500;
```

Sum the first 1000 terms, adding fractions ALGEBRAICALLY instead of numerically

```
[> s := 1;
    for i from 1 to n do
        s := s + 1/(2*i + 1) - 1/(2*i);
    od;
[> s;                                # the exact answer, in
                                        fraction form
[> evalf(s, 3);                       # convert the fraction
                                        to decimal form, 3 digit precision
```

Recompute, using decimal approximations, rounded to three places, for the fractions in the sum.

```
[> Digits := 3;                       # just keep track of 3 digits
[> s := 1.0;
```

Since we type $1.0/(2*i+1)$ etc the fractions are converted to decimal form, rounded to 3 digits after the decimal point

```
[> for i from 1 to n do
        s := s + 1.0/(2*i + 1) - 1.0/(2*i);
    od;
[> s;
```

4. Write a program that takes an integer n as input and then computes a floating point number $dx = 1.0/n$. Use a loop to compute

$$\sum_{i=1}^n \frac{1}{n} = \underbrace{\frac{1}{n} + \frac{1}{n} + \cdots + \frac{1}{n}}_{n \text{ times}}$$

Modify your program to compute the sum for $n = 2, 3, \dots, 20$. Display your results in a table.

What is the actual value of the sum? Would it help to use `double` variables instead of `float`?

- Using the two pseudoprograms given in the text above as a guide, write two programs to compute $x^n/n!$. In both programs use `double` variables. Compare their performance — with each version what is the largest value of n for which you can compute $x^n/n!$ accurately if $x = 2$? if $x = 3$? if $x = 10$?

§ 6. Files

When a problem requires the manipulation of more than a small amount of information, it is generally best to read it from and write to a file — a “permanent” store of data on disk. To illustrate how this is done, we will first learn how to create a file of random numbers, then how to read and analyze it.

Writing Data to a File

Our first example is a C program which creates a file of “random” integers. The number of integers generated is under the control of the user, and the user can select the name of the file in which these integers are stored.

```
/* randfile.c: creates a file of N random numbers */

#include <stdlib.h>
#include <stdio.h>

main(){

    int i, N;
    FILE *fp;
    char *fname;

    printf("Number of random integers to generate: ");
    scanf("%d", &N);
    printf("File name: ");
    scanf( "%s", fname );

    fp = fopen( fname, "w" );
    for( i = 0; i < N; i++ )
        fprintf( fp, "%5d\n", rand() );

    fclose( fp );
}
```

We see several new elements in this program. First is the declaration of the *file pointer* `fp`. It is through `fp` that we access a file for reading and writing. These operations require that we declare the library `<stdio.h>`. Second is the declaration of the pointer `fname`, which is used for the name the user gives to the file. In C we use pointers to manipulate character strings like "John", "Mary", etc. For now we will set aside the question of what a pointer is (see Kernighan and Ritchie) and instead talk about how they are used. Notice that when we declare a pointer, the character `*` appears before its name. Observe that this is the case with `fp`, `fname`, etc.

We also see the function `rand`, which is defined in `<stdlib.h>`. Each evaluation of the expression `rand()` results in the production of a random integer in the range 0 to `RAND_MAX`. The latter is a system constant guaranteed to be at least 32767. (It is always at least $2^{15} - 1$, or 32,767. To determine `RAND_MAX` experimentally, see the exercises below.)

Below is a sample session with `randfile`, which was compiled on a unix system by the command `cc -o randfile randfile.c`. The user gives the name `random.data` to the file created, and the unix command `cat random.data` displays the file contents.

```
> randfile
Number of random integers to generate: 10
File name: random.data
> cat random.data
16838
5758
10113
17515
31051
5627
23010
7419
16212
4086
```

Let us examine how the file was created. The user responded to the questions posed by `randfile`, and so `N` takes the value 10 and `fname` takes the value `random.data`. Note that we wrote `scanf("%s", fname)`, not `scanf("%s", &fname)`. This is because `scanf` requires a pointer, which is what `fname` is. Contrast this with the statement `scanf("%d", &N)`. The `int` variable `N` is not a pointer, but `&N` points to `N`. Note that `"%s"` is the format for reading a string of characters.

The function `fopen` opens a file with that name and associates it to the file pointer `fp`. The `"w"` in `fopen` informs our computer that the file is on into which

we will write data. Data is written using `fprintf` and the function `rand` under the control of a `for` loop. The function `fprintf` is like `printf`, but writes to files, not the screen. The function `rand` returns a “random” integer each time it is called (used). Note that in `fprintf` we must specify the file pointer used as well as the format string and the data to be written.

When calling the function `rand`, we write `rand()`. This is because the function takes no arguments (inputs). This is unlike a function such as `sqrt`, declared in the library `<math.h>`, which takes a number — `int`, `float`, etc. — as its argument.

The last statement, `fclose(fp)` closes the file associated to `fp`. You should always close files after using them.

Reading Data from a File

We now devise a program `avfile.c` which can read the data created by `randfile.c` and compute various statistics of that file, e.g., the average of the numbers it contains.

```

/* avfile.c: computes statistics on a file of numbers */

#include <stdio.h>

main()
{
int k, count;
float item, sum;

FILE *fp;
char *fname;

printf("File to open: ");
scanf( "%s", fname );
fp = fopen( fname, "r" );
count = 0;
sum = 0.0;

do {
    k = fscanf( fp, "%f", &item );
    if ( k == 1 ) {
        count++;
        sum += item;
    }
} while ( k != EOF );

printf( "  %d items\n  sum = %f\n", count, sum );

if ( count > 0 )
    printf( "  average = %f\n", sum/count );

fclose( fp );
}

```

Here is a session with avfile, which we compiled with the unix command `cc -o avfile avfile.c`:

```

> avfile
File to open: random.data
10 items
sum = 137629.000000
average = 13762.900391

```

The variable `fname` is set to `random.data`, the pointer `fp` is set up to read data from the file `random.data`, and the the data is processed under the control of a

loop which uses `fscanf`, the file counterpart of `scanf`. The loop determines the number of items in the file and the sum of these items, storing this information in `count` and `sum`, respectively. Note that `item` is not a pointer, so we must write `&item` to turn it into one, as `fscanf` requires.

To understand how the `do-while` loop terminates when all the data is read, we need to understand `fscanf` better. In addition to reading data into `item`, `fscanf` returns an integer value. This value is the number of values read and assigned, or a special integer, `EOF` (usually `-1`) if no data could be read (because the end of the file was reached). The constant `EOF` is defined in `<stdio.h>` and stands for “end of file.” The symbol `!=` is C for “not equal to.” Compare with `==`, “equal to.”

Here is another version of the above loop:

```
while ( fscanf( fp, "%f", &item ) != EOF do {
    count++;
    sum += item;
}
```

it is more concise, but perhaps more obscure, since the main action occurs in the boolean expression. To verify that `fscanf` works as advertised, we run the following program:

```
/* test.c Look at the hidden value of fscanf */

#include <stdio.h>

main()
{
    int k, item;

    FILE *fp;

    fp = fopen( "my.data", "r" );

    do {
        k = fscanf( fp, "%f", &item );
        printf("%d ", k);
    } while ( k > 0 );

    fclose( fp );

}
```

Try it, and think about what you see.

Exercises.

1. Devise a program which computes a two-column table of factorials and which stores the output in a file.
2. Devise a program to find the maximum and minimum entries in a file like `random.data`. Use the pseudoprogram below as a guide, and use `avfile.c` as a template to be modified. Then run your program on the output of `randfile.c` for $N = 1000$, $N=5000$, $N=10000$, and $N=50000$. Can you determine `RAND_MAX` from this experiment?

Algorithm MM: find the max and min integer in a file

Open the data file (for reading)

Read the first number in the file and store it as `item`.

```
max ← item
```

```
min ← item
```

```
do
```

```
    Read in the next number and store it as item.
```

```
    if max < item
```

```
        then max ← item
```

```
    if min > item
```

```
        then min ← item
```

```
while there is data to read.
```

```
Close the data file.
```

3. Generate N random numbers between -1 and 1 . Store them in a file. Then write a program that reads from this file and computes the sum of these N numbers. Intuitively, one expects the sum to be close to zero, since we expect that randomly chosen integers in $[0, 1]$ will tend to cancel randomly chosen integers in $[-1, 0]$. Compute the sum for $N = 1000$, for $N = 10,000$ and for $N = 100,000$.
4. In §4 above, we discussed the effect of round-off error. Given N integers that have a possible error of $\pm 10^{-6}$, randomly distributed, what sort of accumulated error do we expect to see? Use your results in the previous exercise to give an experimentally-based speculation.
5. In Exercise 1 of §3 above, a program to produce the prime factorization of a given integer n is outlined. Write a program that takes a large integer M as its input and produces as its output a file of all the primes less than or equal to M . For example, if $M = 10$, your file would look like:

```
2
3
5
7
```

Then write a second program that counts how many primes are in that file. Use your program to determine how many primes there are ≤ 100 , ≤ 1000 , $\leq 10,000$.

6. In the factoring program described in Exercise 1 of §3, integers were factored by trial division. That program was quite inefficient. For example to factor 308, the program we wrote there would go through the following steps. First we would factor out as many 2's as possible to get $308 = 2 \cdot 2 \cdot 77$. Next we check if 3 divides 77, but it does not. The next divisor we try is 4, but checking this divisor is, of course, a waste of time. We already know that 4 won't work because we already accounted for all factors of 2 in the original number. Next we try 5, and it fails as well. Our next trial divisor is 6, again a wasted effort since we already checked both 2 and 3. At the next step we finish since 77 factors as 7 times 11.

We could make the factoring program more efficient by checking only prime divisors. In the previous exercise we wrote a program that makes a file of all prime numbers less than M and we can use that file as a source of possible divisors in our factoring program.

Write a factoring program that does this. Use the pseudoprogram below as a guide:

Factor an integer n using a file of primes as a source for trial divisors.

Open the file that contains your list of primes (for reading)

Read the first prime from the file and store it as p .

while $n > 1$

 while p divides n

$n \leftarrow n/p$

 print p

 Read in the next prime and store as p .

Close the file of primes.