An Introduction to C
Through Annotated Examples

by

Henry M. Walker

Grinnell College, Grinnell, IA

Program 1: quarts-1.c

The first program converts a number of quarts to liters, from the keyboard using the `stdio.h` .

```
/* A simple program to convert a number of quarts to liters
   Version 1:  global variables only
*/

#include <stdio.h>                        /* reference to standard I/O library */

int quarts;                                 /* declarations */
double liters;                              /* double = real */

main ()                                     /* beginning of main program */
{  printf ("Enter volume in quarts: ");    /* input prompt */

   scanf ("%d", &quarts);                   /* read */

   liters = quarts / 1.056710 ;             /* arithemtic, assignment */

   printf ("%d quarts = %lf liters\n", quarts, liters);
                                            /* write text and new line */
}
```

Annotations for `quarts-1.c`*:*

- Comments in C begin anywhere with the symbols `/*`, continuing until the symbols `*/`, on the same or later lines.
- C makes use of libraries for many common operations. The statement
  `#include <stdio.h>`
  instructs the machine how to use the stdio operations.
- Each C program contains a driver function/program, called `main`. Here, main uses no input parameters (hence the parentheses with nothing between them following `main`).
- Variables may be declared globally or at the start of a C function. Here, `quarts` and `liters` are declared globally as integer and real variables, respectively. The term `double` specifies a real number, stored using double precision. (The term `float` may be used for single precision, real numbers.)
- Braces { and } are used in C to mark the beginning and ending of blocks. In this case, the braces indicate the statements for the main program.
- Semicolons (;) are used to terminate every statement in C.
- The equal sign (=) is used for assignment. (We will see later that == is used for the comparison operator.)
- Arithmetic operations include +, −, *, and / for addition, subtraction, multiplication, and division. For integers, the division operation / yields the integer quotient, while the modulus operation % gives the remainder.

- `scanf` and `printf` are used for input and output, respectively. In each case, the first parameter is a string, which indicates how input or output will be formatted.
  - § When printing text only, as in the first output line in this program, the text of the string is enclosed in double quotes `" "`, and the characters are printed exactly as given.
  - § Within a format string, some symbols are used for special symbols. For example,
    `'\n'` stands for a new-line character,
    `'\t'` stands for a tab character,
    `'\"'` stands for a double quote character, and
    `'\\'` stands for the backslash character itself.
  - § When printing the value of a variable, the format string for `printf` gives the type of data to be read:
    `"%d"` stands for a (decimal) integer,
    `"%f"` stands for a (floating point) real number,
    `"%lf"` stands for a double precision (long) real number,
    `"%c"` stands for a (single) character,
    `"%s"` stands for a character string.
  - § When reading a value, the `scanf` procedure uses the same formatting conventions as `printf`.
  - § When using `scanf`, one must pass the address of variables, rather than their values. That is, `scanf` must know the address in memory into which a value is to be placed. The ampersand & before `quarts` in the `scanf` statement specifies "the address of `quarts`".

Sample Run of `quarts-1.c`*:*

```
babbage% gcc -o quarts-1 quarts-1.c
babbage% quarts-1
Enter volume in quarts: 4
4 quarts = 3.785334 liters
babbage% quarts-1
Enter volume in quarts: 1
1 quarts = 0.946333 liters
babbage% quarts-1
Enter volume in quarts: 3.5
3 quarts = 2.839000 liters
```

In this sample run (run on the machine *babbage*), the program (called `quarts-1.c`) is compiled and run using the gcc compiler.

Since the program reads quarts as an integer, the input 3.5 is read as the integer 3 in the third run of the program. If additional input were requested within the program, the next characters read would be .5 .

Program 2: quarts-2.c

Another program to convert a number of quarts to liters.

```
/* A simple program to convert a number of quarts to liters    *
 * Version 2:  using local variables and simple error checking */

#include <stdio.h>
#include <assert.h>

int main (void)            /* explicitly state main has no parameters */
{  int quarts;                                /* variables declared   */
   double liters;                             /* at the start of main */

   printf ("Enter volume in quarts: ");
   scanf ("%d", &quarts);

   assert (quarts >= 0);                       /* abort if quarts < 0 */

   liters = quarts / 1.056710 ;
   printf ("%d quarts = %lf liters\n", quarts, liters);

   return 0;                         /* tell OS no errors occurred */
}
```

Annotations for `quarts-2.c`*:*

- In declaring the `main` header, the addition of `void` states explicitly that `main` is a procedure of no parameters.
- `main` may return an integer error code to the operating system by declaring; thus, the declaration `ing main` and the concluding statement `return 0` to indicate no errors.
- Variables can be declared within a function at the start of a block (i.e., immediately after a left brace {. Here, variables are declared at the beginning of the function `main`, so they can be used any time within that function.
- The `assert` statement checks the validity of a Boolean expression. If the statement is true, program execution continues without interruption. If the condition is false, however, the program is terminated, with an appropriate error message printed.
- The `assert` statement provides a simple mechanism to verify pre-conditions and post-conditions, although more sophisticated approaches are needed if the program is to resolve the problem and resume processing. (This latter approach is accomplished by exception handlers, to be discussed at a later time.)

Sample Run of `quarts-2.c`*:*

```
babbage% gcc -o quarts-2 quarts-2.c
babbage% quarts-2
Enter volume in quarts: 4
4 quarts = 3.785334 liters
babbage% quarts-2
Enter volume in quarts: -3
quarts-2.c:15: failed assertion ‘quarts >= 0’
IOT trap (core dumped)
```

Program 3: quarts-3.c

Program illustrating variable declaration and initialization, together with simple error checking.

```
/* A simple program to convert a number of quarts to liters   *
 * Version 3:  declaring local variables just as needed       */

#include <stdio.h>

int main (void)
{  int quarts;
   printf ("Enter volume in quarts: ");
   scanf ("%d", &quarts);

   /* if value of quarts is invalid, ask user again */
   while (quarts < 0) {
     printf ("Value of quarts must be nonnegative; enter value again: ");
     scanf ("%d", &quarts);
   }

   {  double liters = quarts / 1.056710 ;/* declaring and initializing liters
                                            in a new block, just as needed */

      printf ("%d quarts = %lf liters\n", quarts, liters);
   }                                     /*end of block with liters */

   return 0;
}
```

Annotations for `quarts-3.c`*:*

- C allows variables to be declared at the start of any block. Thus, in this program, `liters` is declared in a new block, after the value of `quarts` has been entered successfully.
- C also allows a variable to be initialized when it is declared by specifying the variable's initial value as part of the declaration. If this initial value were not given as part of the declaration, then the declaration line for `liters` would be replaced by the two lines:
  `double liters;`
  `liters = quarts / 1.056710;`
- Experienced C programmers disagree upon whether variable declarations should normally be placed at the start of a function or whether variables should be declared just before they are needed at the start of interior blocks.
  - § Variables declared at the start of a function often are easier for a programmer to find and check.
  - § When variables are declared as needed, memory allocation may not be needed if a section of code is skipped during specific runs of a program.
- For clarity in the following examples, variables normally are declared at the start of functions, unless there are special reasons to proceed otherwise.
- The `while` loop continues as long as the condition specified is maintained. The general syntax is:
  `while (condition) statement ;` where condition is any Boolean condition and the statement is any single statement. When several statements are to be executed in the body of a loop, they are enclosed in braces { }, as shown in the example.

Sample Run of `quarts-3.c`*:*

```
babbage% gcc -o quarts-3 quarts-3.c && quarts-3
Enter volume in quarts: 4
4 quarts = 3.785334 liters
babbage% quarts-3
Enter volume in quarts: -5
Value of quarts must be nonnegative; enter value again: -2
Value of quarts must be nonnegative; enter value again: 1
1 quarts = 0.946333 liters
```

In this work, the steps for compiling and running this program were combined on one line with the conjunction &&. This conditional conjunction && states that the first command (`gcc -o quarts-3 quarts-3.c`) should be done first. However, the second command (running `quarts-3`) will occur only if the first step is successful.

Program 4: smallest3-1.c

Simplistic program to find the smallest of three integers.

```
/* A simple program to find the smallest of three numbers   *
 * Version 1:  using simple if-then statements              */

#include <stdio.h>

int main (void)
{  int i1, i2, i3;

   printf ("Program to determine the smallest of three integers\n");
   printf ("Enter three integer values:  ");
   scanf ("%d %d %d", &i1, &i2, &i3);

   if ((i1 <= i2) && (i1 <= i3))
      printf ("The smallest value is %d\n", i1);

   if ((i2 < i1) && (i2 <= i3))
      printf ("The smallest value is %d\n", i2);

   if ((i3 < i1) && (i3 < i2))
      printf ("The smallest value is %d\n", i3);

   return 0;
}
```

Annotations for `smallest3-1.c`*:*

- The simple `if` statement has the form:
  `if (condition) statement`
  Here, the "condition" is evaluated, and the "statement" is executed if the "condition" is true. Otherwise, the "statement" is skipped.
- Simple comparisons may use the relations: `<`, `<=`, `>`, `>=`, `==` and `!=`. Here `!` means "not", so `!=` means "not equal".
- Technically, C has no separate Boolean type, and Boolean results are considered integers. Further, in C, a zero value is considered as false, while any nonzero value is true.
- *Caution:* Programmers who may have programmed in other languages sometimes mistakenly write `=` for `==`. In C, the compiler will interpret the `=` as an assignment. Thus, the statement
  `if (i = j) cout << "Equal" ;`
  will assign the value of `j` to `i`. If the value of `j` was 0, then the result of the assignment is considered false, and nothing will be printed. On the other hand, if `j` had a nonzero value, then the assignment to `i` returns a true (i.e., nonzero) value, and output is generated.
  Be careful in writing comparisons to use `==` *rather than* `=`.
- Comparisons may be combined with `&&` and `||` for "and" and "or", respectively. The precedence of such operations follows the familiar rules of Boolean algebra. However, when in doubt, it is always safer to use parentheses to clarify meaning.

- In reading multiple values with `scanf`, the format string must contain a specification for each variable to be read. Thus, this program contains
  `scanf ("%d %d %d", &i1, &i2, &i3);`
  In addition, `scanf` interprets the characters between the variable specifications as follows:
  - § Blanks or tabs are ignored (as is the case here).
  - § Ordinary characers (not %) are expected to match the next non-white space characters of the input stream exactly.
  - § Conversion specifications begin with %, as shown here for (decimal) integers.

Sample Run of `smallest3-1.c`*:*

```
babbage% gcc -o smallest3-1 smallest3-1.c && smallest3-1
Program to determine the smallest of three integers
Enter three integer values:  1 2 3
The smallest value is 1
babbage% smallest3-1
Program to determine the smallest of three integers
Enter three integer values:  1 1 1
The smallest value is 1
babbage% smallest3-1
Program to determine the smallest of three integers
Enter three integer values:  2 1 0
The smallest value is 0
babbage% smallest3-1
Program to determine the smallest of three integers
Enter three integer values:  0 2 0
The smallest value is 0
```

Program 5: smallest3-2.c

Program to find the smallest of three integers, using two steps.

```
/* A simple program to find the smallest of three numbers            *
 * Version 2:  using if-then-else statements with intermediate steps */

#include <stdio.h>

int main (void)
{  int i1, i2, i3;
   int smaller, smallest;

   printf ("Program to determine the smallest of three integers\n");
   printf ("Enter three integer values:  ");
   scanf ("%d %d %d", &i1, &i2, &i3);

   if (i1 <= i2)
        smaller = i1;
   else smaller = i2;

   if (smaller <= i3)
        smallest = smaller;
   else smallest = i3;

   printf ("The smallest value is %d\n", smallest);

   return 0;
}
```

Annotations for `smallest3-2.c`*:*

- The compound `if` statement has the form:
  `if (condition) statement1`
  `else statement 2`
  As with the simple `if` statement, the "condition" is evaluated and "statement1" is executed if the "condition" is true. In this compound statement, however, "statement2" is executed if the "condition" is false.
- Note that both "statement1" and "statement2" end with semicolons (as with all C statements.

This program produces the same output as `smallest3-1.c`.

Program 6: smallest3-3.c

Program to find the smallest of three integers, using nested if statements.

```
/* A simple program to find the smallest of three numbers   *
 * Version 3:  using nested if-then-else statements         */

#include <stdio.h>

int main (void)
{  int i1, i2, i3;
   int smallest;

   printf ("Program to determine the smallest of three integers\n");
   printf ("Enter three integer values:  ");
   scanf ("%d %d %d", &i1, &i2, &i3);

   if (i1 <= i2)
        /* compare i1 and i3; i2 cannot be smallest */
        if (i1 <= i3)
           smallest = i1;
           else smallest = i3;
     else
        /* compare i2 and i3; i1 cannot be smallest */
        if (i2 <= i3)
           smallest = i2;
           else smallest = i3;

   printf ("The smallest value is %d\n", smallest);

   return 0;
}
```

Annotations for `smallest3-3.c`*:*

- `if` statements can be nested as desired. Since each `if` statement is considered a single entity, such statements may be used in either the "then" or "else" clauses of other `if` statements.

This program also produces the same output as `smallest3-1.c`.

Program 7: smallest3-4.c

Program to find the smallest of three integers, using nested if statements with brackets for clarity.

```
/* A simple program to find the smallest of three numbers   *
 * Version 4:  using nested if-then-else statements         */

#include <stdio.h>

int main (void)
{  int i1, i2, i3;
   int smallest;

   printf ("Program to determine the smallest of three integers\n");
   printf ("Enter three integer values:  ");
   scanf ("%d %d %d", &i1, &i2, &i3);

   if (i1 <= i2)
        /* compare i1 and i3; i2 cannot be smallest */
        {if (i1 <= i3)
           smallest = i1;
           else smallest = i3;
        }
     else
        /* compare i2 and i3; i1 cannot be smallest */
        {if (i2 <= i3)
           smallest = i2;
           else smallest = i3;
        }

   printf ("The smallest value is %d\n", smallest);

   return 0;
}
```

Annotations for `smallest3-4.c`*:*

- When several statements are to be used within "then" or "else" clauses, braces { } are used to group the statements. Such braces also can be used around single statements for clarity.

This program again produces the same output as `smallest3-1.c`.

Program 8: quarts-for-1.c

A program to compute the number of liters for several values of quarts.

```
/* A program to compute the number of liters for 1, 2, ..., 12 quarts *
 * Version 1:  simple table without formatting                         */

#include <stdio.h>

int main (void)
{   int quarts;
    double liters;

    printf (" Table of quart and liter equivalents\n\n");
                                /* two \n's to skip an extra line */

    printf ("Quarts        Liters\n");

    for (quarts = 1; quarts <= 12; quarts++)
      { liters  = quarts / 1.056710 ;
        printf ("  %d          %f\n", quarts, liters);
      }

    return 0;
}
```

Annotations for `quarts-for-1.c`*:*

- The `for` statement follows the syntax:
  `for (initialization; condition; updates) statement;`
  Here, "initialization" (if any) is performed at the start, before any Boolean expressions are evaluated. The Boolean "condition" is evaluated at the top of each loop iteration. If the condition is true, the "statement" is executed. Otherwise, execution of the `for` statement terminates, and processing continues with the statement following the `for`. The "updates" allow any variables or other work to be done after a loop iteration, before the "condition" is evaluated again.
- The ++ operation associated with a variable increments the variable by 1. Thus, `i++` is the same as the statement `i = i + 1`. (Technically, `i++` is a post-increment operation, while `++i` is a pre-increment operation. Here, the incrementing of `i` takes place in a statement by itself, and either operation has the same effect. Further consideration of such subtleties is left to a later time.)
- Note that the `for` statement is more general than in some other languages, as any initialization and updating are possible. For example, the `while` loop is a special case of the `for`, with empty initialization and updating sections: `for (;condition;) statement`; That is, `while (condition) statement;` is equivalent to
  `for (; condition; ) statement;`
- In order to arrange the values approximately in columns, the `printf` format string contains several spaces: `"  %d          %f\n"`. The amount of space allocated for each number depends upon the size and accuracy of the number.

Sample Run of `quarts-for-1.c`*:*

```
babbage% gcc -o quarts-for-1 quarts-for-1.c && quarts-for-1
 Table of quart and liter equivalents

Quarts        Liters
  1          0.946333
  2          1.892667
  3          2.839000
  4          3.785334
  5          4.731667
  6          5.678001
  7          6.624334
  8          7.570667
  9          8.517001
  10          9.463334
  11          10.409668
  12          11.356001
```

Program 9: quarts-for-2.c

A table of quart and liter values, using formatted output.

```
/* A program to compute the number of liters for 1, 2, ..., 12 quarts *
 * Version 2:  simple table with formatting of integers and reals     */

#include <stdio.h>

int main (void)
{   int quarts;
    double liters;

    printf (" Table of quart and liter equivalents\n\n");
    printf ("Quarts        Liters\n");


    for (quarts = 1; quarts <= 12; quarts++)
      { liters  = quarts / 1.056710 ;
        printf ("%4d%16.4f\n", quarts, liters);
        /* use 4-character width for printing the integer quarts,
           use 16-character width, with 4 places after the decimal point,
           for printing floating point number liters */
      }

    return 0;
}
```

Annotations for `quarts-for-2.c`*:*

- Formatting of integers in C is relatively simple: Only a width must be given in the output specification, and by default integers are right justified in the width given. (If the integer requires more space than given by the width, then the width in expanded as needed.) The width is specified as part of the number format, as shown in the program.
- Similarly, formatting of real numbers in C requires a specification of both the overall width for the number and the number of decimal places to be printed. Both of these values are illustrated in the program; the specification `%16.4f` indicates that 16 characters will be allocated to the overall number, and the number will be printed with four digits to the right of the decimal point.
- Note that space characters in a `printf` format string are printed as specified. Thus, the following lines would yield the same results shown in this output:

```
printf ("%4d%16.4f\n", quarts, liters);
printf ("%4d %15.4f\n", quarts, liters);
printf ("%4d  %14.4f\n", quarts, liters);
```

Sample Run of `quarts-for-2.c`*:*

```
babbage% gcc -o quarts-for-2 quarts-for-2.c && quarts-for-2
 Table of quart and liter equivalents

Quarts        Liters
   1          0.9463
   2          1.8927
   3          2.8390
   4          3.7853
   5          4.7317
   6          5.6780
   7          6.6243
   8          7.5707
   9          8.5170
  10          9.4633
  11         10.4097
  12         11.3560
```

Program 10: quarts-for-3.c

Another table of quart and liter equivalents – this time including half quarts as well.

```
/* A program to compute the number of liters for 0.5, 1, ..., 5.5, 6.0 quarts
   Version 3:  table expanded to half values for quarts                      */

#include <stdio.h>

int main (void)
{   const double conversion_factor = 1.056710;  /* declaration of constant */
    double quarts, liters;

    printf (" Table of quart and liter equivalents\n\n");
    printf ("Quarts        Liters\n");

    for (quarts = 0.5; quarts <= 6.0; quarts += 0.5)
      { liters  = quarts / conversion_factor ;
        printf ("%5.1f%15.4f\n", quarts, liters);
      }

    return 0;
}
```

Annotations for `quarts-for-3.c`*:*

- This program illustrates that the `for` statement is quite general. Here the control variable `quarts` is real, rather than integer. While this seems reasonable in this context, note that roundoff error for real numbers sometimes can provide unanticipated or incorrect results within such loops.
- In C, the assignment `quarts += 0.5` is shorthand for `quarts = quarts + 0.5`. More generally, the operator `+=` adds the value on the right-hand side of the operator to the variable on the left-hand side.
- Constants are declared within a program by specifying the type of the variable, the initial value, and designating the variable as being a constant, `const`. Thus, the declaration `const double conversion_factor = 1.056710;` indicates that the variable `conversion_factor` will be a double precision, real number with value 1.056710, and this value is not allowed to change within the `main` program where this constant is defined.

Sample Run of `quarts-for-3.c`*:*

```
babbage% gcc -o quarts-for-3 quarts-for-3.c && quarts-for-3
 Table of quart and liter equivalents

Quarts        Liters
  0.5         0.4732
  1.0         0.9463
  1.5         1.4195
  2.0         1.8927
  2.5         2.3658
  3.0         2.8390
  3.5         3.3122
  4.0         3.7853
  4.5         4.2585
  5.0         4.7317
  5.5         5.2048
  6.0         5.6780
```

Program 11: quarts-for-4.c

This program illustrates the nesting of `for` loops to produce a table of liter equivalents.

```
/*******************************************************************
 *  A program to compute the number of liters for gallons and quarts *
 *  Version 4:  table of liters for quarts and gallons             *
 *******************************************************************/
#define conversion_factor 1.05671

#include <stdio.h>

int main (void)
{  int gals, quarts;
   double liters;
   printf ("Table of liter equivalents for gallons and quarts\n\n");
   printf ("                           Quarts\n");
   printf ("Gallons         0          1          2           3\n");
   for (gals = 0; gals <= 5; gals++)
     { printf ("%4d     ", gals);
       for (quarts = 0; quarts <= 3; quarts++)
         { liters  = (4.0*gals + quarts) / conversion_factor ;
           printf ("%12.4f", liters);
         }
     printf ("\n");
      }

   return 0;
}
```

Annotations for `quarts-for-4.c`*:*

- This program also illustrates another common approach to defining constants within C. Historically, `#define` statements were used to indicate constants well before the `const` modifier was introduced. Thus, most old programs and most experienced C programmers are likely to use the `#define` construct. (In contrast, experienced C++ programmers are more likely to use `const`.)

Sample Run of `quarts-for-4.c`*:*

```
babbage% gcc -o quarts-for-4 quarts-for-4.c && quarts-for-4
Table of liter equivalents for gallons and quarts

                           Quarts
Gallons         0          1          2           3
   0          0.0000     0.9463     1.8927      2.8390
   1          3.7853     4.7317     5.6780      6.6243
   2          7.5707     8.5170     9.4633     10.4097
   3         11.3560    12.3023    13.2487     14.1950
   4         15.1413    16.0877    17.0340     17.9803
   5         18.9267    19.8730    20.8193     21.7657
```

Program 12: darts.c

A Monte Carlo simulation to approximate Pi. More precisely, consider a circle C of radius 1 in the plane, centered at the origin. Also, consider the square S of side 2, centered at the origin, so that the corners of the square are at $(\pm 1, \pm 1)$. If we pick a point P at random in S or in the first quadrant of S. Then the probability that P is also in C is $\pi/4$.

```
/*  This program approximates Pi by picking a points in a sqaure and   *
 *  and determining how often they are also in an appropriate circle.  */

#include <stdio.h>

/* libraries for the random number generator */
#include <stdlib.h>
#include <time.h>

/* Within the stdlib.h library,
 *    time returns a value based upon the time of day
 *    on some machines, rand returns a random integer between 0 and 2^31 - 1
 *        although on some machines rand gives values between 0 and 2^32 - 1
 *        and on other machines rand gives values between 0 and 2^15 - 1
 *    MaxRandInt is this maximum integer minus 1
 *        (Note:  2^32 = 2147483648, 2^31 = 1073741824 and 2^15 = 32768)
 *    Use 2^32-1 for SparkStations
 *    Use RAND_MAX for Linux machines,
 *    Use 2^15-1 for IBM Xstation 140s and HP 712/60s
 */
const int MaxRandInt = RAND_MAX;       /* declaration of program constants*/
const int NumberOfTrials = 5000;

int main (void)
{  int i;
   int counter = 0;                     /* declare and initialize counter */
   double x, y;
   double MaxRandReal = (double) MaxRandInt; /* make MaxRandInt a real */

   printf ("This program approximates Pi by picking %d points in a square\n",
            NumberOfTrials);
   printf ("and counting the number in an appropriate circle.\n\n");

   // initialize random number generator
   // change the seed to the random number generator, based on the time of day
   srand (time ((time_t *) 0) );

   // pick points in first quadrant with coordinates between 0 and 1
   // determine how many are in the circle of radius 1
   for (i = 1; i <= NumberOfTrials; i++) {
       x = rand() / MaxRandReal;
       y = rand() / MaxRandReal;
       if (x*x + y*y <= 1) counter++;
   }

   printf ("%d points were inside the circle, and\n", counter);
   printf ("the approximate value of Pi is %.5f .\n",
               4.0 * counter / NumberOfTrials);
   return 0;
}
```

Annotations for `darts.c`*:*

- This program illustrates the use of the random number generator `rand`, which is based upon a seed value kept internally. `rand` returns pseudo-random positive integers. `srand` is used to set the initial seed value, based upon the built-in `time` function. The range of the integer value returned by `rand` depends upon the local machine executing the code. Some choices are noted in the program.
- The `const` statements before the `main` specify global constants (and other expressions) for use throughout all functions.
- The expression `(double) MaxRandInt` performs type conversions. In the definitions from this program, `MaxRandInt` represents an integer value. The prefix `(double)` converts this to a real value for `MaxRandReal`.
- The function `rand ()` returns a random integer value from the uniform distribution from 0 to `MaxRandInt`. Since the program divides this by a real value (`MaxRandReal`), C converts the integer to a real before the division takes place. If we had not declared `MaxRandReal` as a real, but used `MaxRandInt` instead, then we would need to convert both `rand ()` and `MaxRandInt` to integers before performing the division (otherwise we would obtain an integer result, in contrast to what was intended). This alternative conversion could be performed with the revised statements
  ```
  x = (double) rand () / (double) MaxRandReal;
  y = (double) rand () / (double) MaxRandReal;
  ```

Sample Run of `darts.c`*:*

```
babbage% gcc -o darts darts.c && darts
This program approximates Pi by picking 5000 points in a square
and counting the number in an appropriate circle.

3942 points were inside the circle, and
the approximate value of Pi is 3.15360 .
babbage% darts
This program approximates Pi by picking 5000 points in a square
and counting the number in an appropriate circle.

3924 points were inside the circle, and
the approximate value of Pi is 3.13920 .
babbage% darts
This program approximates Pi by picking 5000 points in a square
and counting the number in an appropriate circle.

3854 points were inside the circle, and
the approximate value of Pi is 3.08320 .
babbage% darts
This program approximates Pi by picking 5000 points in a square
and counting the number in an appropriate circle.

3905 points were inside the circle, and
the approximate value of Pi is 3.12400 .
```

Program 13: max-min.c

This following program computes the maximum, minimum, and average of n real numbers.

```
/* A program to read n numbers, compute their maximum, minimum, and average,
 * and to print a specified jth item.
 * This code illustrates built-in C-style arrays.
 */
#include <stdio.h>

int main (void)
{    int j, n;
     double max, min, sum;

     printf ("Program to process real numbers.\n");
     printf ("Enter number of reals: ");
     scanf ("%d", &n);

     {  /* start new block, where new array may be declared */
        double a[n];  /* declare array of n values, with subscripts 0, ..., n-1
                         While ANSI C allows only constant array sizes,
                         some compilers allow variables in the specification
                         of array size, as shown here                        */

        printf ("Enter %d numbers: ", n);
        for (j = 0; j < n; j++)
             scanf ("%lf", &a[j]); /* subscripts given in brackets [ ]  */

        sum = max = min = a[0];    /* right to left assignment operator */

        for (j = 1; j < n; j++)
           { if (a[j] > max)
                max = a[j];
             if (a[j] < min)
                min = a[j];
             sum += a[j];
           }
        printf ("Maximum:  %5.2f\n", max);
        printf ("Minimum:  %5.2f\n", min);
        printf ("Average:  %5.2f\n\n", sum/n);

        printf ("Enter the index (1..n) of the number to be printed: ");
        scanf  ("%d", &j);
        printf ("The %d th number is %.2f\n", j, a[j-1]);
     } /* end of array block */

     return 0;
}
```

Annotations for `max-min.c`*:*

- Arrays within C are declared by indicating the number of array elements in square brackets [ ], as in the statement `double a[n]`. The ANSI C Standard indicates that, unlike this example, the size of the array must be an integer *constant*. However, this example also shows that some compilers, such as the Gnu compilers from the Free Software Foundation, allow variables to be used in specifying array size. In such cases, an array size could be read first and then used within a new block as part of an array declaration.

- Individual array elements are accessed by placing a subscript in brackets, such as `a[0]` or `a[j]`.
- Reading of individual array elements requires the "address of" operator &, just as is used for reading other values.
- As noted previously, reading of a double-precision real number is accomplished using the `scanf` format `"%lf"`.

Sample Run of `max-min.c`*:*

```
babbage% gcc -o max-min max-min.c && max-min
Program to process real numbers.
Enter number of reals: 6
Enter 6 numbers: 1 2 4 6 8 9
Maximum:   9.00
Minimum:   1.00
Average:   5.00

Enter the index (1..n) of the number to be printed: 5
The 5 th number is 8.00
babbage% max-min
Program to process real numbers.
Enter number of reals: 5
Enter 5 numbers: 4 2 1 3 5
Maximum:   5.00
Minimum:   1.00
Average:   3.00

Enter the index (1..n) of the number to be printed: -3
The -3 th number is 3657243390666508700000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000.00
```

- C does not perform bounds checking when accessing array elements. Thus, nonsense is printed for `a[-3]` when only `a[0]` through `a[5]` are declared and given values. Note especially that no *Subscript Out Of Range* error message is generated for the attempt to access `a[-3]`.

Program 14: trap-1.c

This program approximates Pi as the area of one-fourth of a circle of radius 2.

```
/* A program for approximating the area under a function y = f(x)          *
 * between a and b using the Trapezoidal Rule.                             *
 *    The Trapezoidal Rule divides [a, b] into n evenly spaced intervals   *
 *    of width W = (b-a)/n.  The approximate area is                       *
 *    W*(f(a)/2+f(a+W)+f(a+2W)+ ... +f(a+(n-2)W)+f(a+(n-1)W)+f(b)/2)       *
 * Version 1:  Approximating area under sqrt(4-x^2) on [0, 2].             *
 *    As this is 1/4 of circle of radius 2, result should be Pi.           */

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

#define a 0.0         /* limits for the area under y = f(x) */
#define b 2.0         /* area will be computed under f(x) on [a,b] */

double f(double x)
/* function to be used in the area approximation */
{  return (sqrt(4.0 - x*x));
}

int main (void)
{  int  n;
   double width, sum, xvalue, area;

   printf ("Program approximates the area under a function using the ");
   printf ("Trapezoidal Rule.\n");
   printf ("Enter number of subintervals to be used: ");
   scanf ("%d", &n);

   width = (b - a) / n;

   /* compute the sum in the area approximation */
   sum = (f(a) + f(b)) / 2.0;           /* first and last terms in sum */
   for (xvalue = a + width; xvalue < b; xvalue += width)
      sum += f(xvalue);

   area = sum * width;

   printf ("The approximate area is %7.4f\n", area);

   return 0;
}
```

Annotations for `trap-1.c`*:*

- The main program uses the Trapezoidal Rule to approximate the area under a function $y = f(x)$ on the interval $[a, b]$.
- The function $f(x)$ is defined as a separate function. Specifically, `double f` indicates that the function f will return a double-precisions, floating-point number, and the header `(double x)` specifies that f will have a single, double-precision, floating-point parameter.
- The format of function f is similar to that of main. Both contain a header (with possible parameters), followed by a function body in braces, { }.

- Here, the entire function is declared first, before the main program.
- A function computes and returns a value, and this value is designated using a `return` statement. Here, the value returned is `(sqrt(4.0 - x*x))`.
- The *math.h* library of C contains many common mathematics functions, including $\sin(x)$, $\cos(x)$, $\tan(x)$, $exp(x)$, $log(x)$, $log10(x)$, $pow(x, y)$, and $sqrt(x)$. Here, $exp(x)$ computes e^x, $log(x)$ computes the natural log, $log10(x)$ computes the common log, and $pow(x, y)$ computes x^y.
- In the computation of the Trapezoidal Rule, the constants a and b are specified as constants using `#define` statements in this program.

Sample Run of `trap-1.c`*:*

```
babbage% gcc -o trap-1 trap-1.c -lm
babbage% trap-1
Program approximates the area under a function using the Trapezoidal Rule.
Enter number of subintervals to be used: 100
The approximate area is  3.1404
```

- In compiling programs involving functions in *math.h*, some compilers require the use of the `-lm` option, as shown.

Program 15: trap-2.c

This program uses a function for computations involving the Trapezoidal Rule.

```
/* Approximating area under sqrt(4-x^2) on [0, 2] using the Trapezoidal Rule.*
 * Version 2:  A function performs the area computation.                      */

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

double f(double x);
/* function to be used in the area approximation */

double area(double a, double b, int n);
/* Approximation of area under f(x) on [a, b] using the Trapezoidal Rule */

int main (void)
{  int n;
   printf ("Program approximates the area under a function using the ");
   printf ("Trapezoidal Rule.\n");
   printf ("Enter number of subintervals to be used: ");
   scanf ("%d", &n);
   printf ("The approximate area is %7.4f\n", area (0.0, 2.0, n));
   return 0;
}

double f(double x)
/* function to be used in the area approximation */
{  return (sqrt(4.0 - x*x));
}

double area (double a, double b, int n)
/* Finding area via the Trapezoidal Rule */
{  double width = (b - a) / n;
   double sum = (f(a) + f(b)) / 2.0;   /* first and last terms in sum */
   double xvalue;

   for (xvalue = a + width; xvalue < b; xvalue += width)
      sum += f(xvalue);
   return (sum * width);
}
```

Annotations for `trap-2.c`*:*

- Functions either may be declared in full before a main program or a header may be defined at the start with details given later. This program illustrates the second approach. At the start, functions `f` and `area` are identified with a header, which is terminated by a semicolon to indicate that details are forthcoming later. This header, called a *function prototype*, contains information about the type of value returned by the function (`double` in each case here) and about the number and type of the function parameters.
- When function details are given later, the header information is repeated, followed by the specific code elements.
- Within a function, variables may be declared and initialized, following the same rules illustrated in previous examples for the main program.
- This program produces the same output as the previous program.

Program 16: trap-3.c

This program computes the Trapezoidal Rule using a procedure.

```
/* Approximating the area under sqrt(4-x^2) on [0, 2]        *
 *     using the Trapezoidal Rule.                            *
 * Version 3:  A procedure performs the area computation.    */

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

double f(double x);
/* function to be used in the area approximation */

void compute_area(double a, double b, int n, double *area);
/* Approximation of area under f(x) on [a, b] using the Trapezoidal Rule */

int main (void)
{  int n;
   double new_area;
   printf ("Program approximates the area under a function using the ");
   printf ("Trapezoidal Rule.\n");
   printf ("Enter number of subintervals to be used: ");
   scanf ("%d", &n);

   compute_area(0.0, 2.0, n, &new_area);
   printf ("The approximate area is %7.4f\n", new_area);

   return 0;
}

double f(double x)
/* function to be used in the area approximation */
{  return (sqrt(4.0 - x*x));
}

void compute_area (double a, double b, int n, double *area)
/* Finding area via the Trapezoidal Rule */
{  double width = (b - a) / (double) n;
   double sum = (f(a) + f(b)) / 2.0;   /* first and last terms in sum */
   double xvalue;

   for (xvalue = a + width; xvalue < b; xvalue += width)
      sum += f(xvalue);

   *area = sum * width;
}
```

Annotations for `trap-3.c`*:*

- A function returning no value (a void result) is a procedure.
- In C, all parameters are passed by value (at least for simple variables). Thus, in declaring `double f(double x)` and calling `f(a)`, the value of a is copied to x before the computation of f proceeds.

- If a value is to be returned in a parameter (e.g., as an area), then the address of the actual parameter must be given in the function call. As with the use of `scanf`, this is accomplished by adding an ampersand `&` before the actual parameter (`new_area`) in the function call.

  This use of an address as parameter is prescribed in the function header by adding a star `*` before the actual parameter. Then, within the function, the `variable` itself refers to the address of the actual parameter, while `* variable` refers to the value at that address. Thus, in this example, the assignment, `*area = sum * width;`, indicates that the value of `sum * width` will be stored in the memory location whose address is given by the `area` parameter.

- Once again, this program produces the same output as *trap-1.c*.

Program 17: trap-4.c

Another area computation, involving a home-grown square-root function which tests that square roots are taken of non-negative numbers only.

```
/* Approximating area under y = f(x) on [a, b] using the Trapezoidal Rule.   *
 * Version 4:  Approximating area under sqrt(4-x^2) on [0, 2].               *
 *    Here sqrt is computed via Newton's Method, rather than through math.h. */

#include <stdio.h>
#include <assert.h>

const double a = 0.0;                  /* alternative constant definitions */
const double b = 2.0;                    /* [a, b] gives interval for area */
const double root_accuracy = 0.00005;   /* sqrt computes to this accuracy */

double sqrt(double r)
/* function to compute square root of r using Newton's Method */
{ double change = 1.0;
  double x;              /* current approximation for sqrt(r)           */
  if (r == 0.0)
     return 0.0;         /* the square root of 0.0 is a special case      */
  assert (r > 0.0);      /* negative square roots are not defined         */
  x = r;                 /* r is used as a first approximation to sqrt(r) */
  while ((change > root_accuracy)||(change < - root_accuracy))
    { change = (x*x - r) / (2*x);
      x -= change; }
  return x;
}

double f(double x) /* function to be used in the area approximation */
{  return sqrt(4.0 - x*x); }

int main (void)
{  int  n;
   double width, sum, xvalue, area;

   printf ("Program approximates the area under a function using the ");
   printf ("Trapezoidal Rule.\n");
   printf ("Enter number of subintervals to be used: ");
   scanf ("%d", &n);

   width = (b - a) / n;

   /* compute the sum in the area approximation */
   sum = (f(a) + f(b)) / 2.0;            /* first and last terms in sum */
   for (xvalue = a + width; xvalue < b; xvalue += width)
      sum += f(xvalue);

   area = sum * width;
   printf ("The approximate area is %7.4f\n", area);
   return 0;
}
```

Annotations for `trap-4.c`:

- This program again produces the same output as *trap-1.c*.

Program 18: trap-5.c

Using the same area function for the computation of areas under two functions.

```
/* Approximating the area under several functions                    *
 *      using the Trapezoidal Rule.                                  *
 * Version 5:  An area function has numeric and functional parameters. */

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

double circle(double x);
/* function for a circle of radius 2, centered at the origin */

double parabola(double x);
/* function for the standard parabola y = x^2 */

double area(double a, double b, int n, double f (double));
/* Approximation of area under f(x) on [a, b] using the Trapezoidal Rule */

int main (void)
{  int number;
   printf ("Program approximates the area under a function using the ");
   printf ("Trapezoidal Rule.\n");
   printf ("Enter number of subintervals to be used: ");
   scanf ("%d", &number);

   printf ("\n");;
   printf ("Approximation of 1/4 area of circle of radius 2 is %7.5f .\n\n",
                    area (0.0, 2.0, number, circle));
   printf ("Approximation of area under y = x^2 between 1 and 3 is%8.5f .\n\n",
                    area (1.0, 3.0, number, parabola));
   return 0;
}

double circle(double x)
/* function for a circle of radius 2, centered at the origin */
{  return (sqrt(4.0 - x*x));
}

double parabola(double x)
/* function for the standard parabola y = x^2 */
{ return x*x;
}

double area (double a, double b, int n, double f (double))
/* Finding area via the Trapezoidal Rule */
{  double width = (b - a) / n;
   double sum = (f(a) + f(b)) / 2.0;   /* first and last terms in sum */
   double xvalue;

   for (xvalue = a + width; xvalue < b; xvalue += width)
      sum += f(xvalue);
   return (sum * width);
}
```

Annotations for `trap-5.c`*:*

- This program represents a variation of program *trap-2.c*. Here, `area` computes the area under $y = f(x)$ on $[a, b]$, using n trapezoids, and all of these elements (`a, b, n, f`) are passed as parameters.
- To declare a function parameter, the function is given a formal name (e.g., `f`), and information concerning its parameters and return type are specified. In the example, the header of `area` contains the information
  `double f (double)`
  which indicates that a function will be passed to `area`. This function will take one double precision, real parameter and will return a double precision, real number. Further, when the details of `area` are defined, this function will be referred to as `f` (just as the numbers `a, b` and `n` will be used within `area`).
- When `area` is called, an appropriate function name is specified for `f`, just as numbers are specified for `a, b` and `n`. Thus, in the call
  `area (0.0, 2.0, number, circle)`
  `a` is given the value 0.0, `b` is given the value 2.0, `n` is given the value *number*, and `f` will refer to the function *circle*. Whenever `f` is mentioned within `area` during the execution of this call, the function *circle* will be used. Similarly, for the call
  `area (1.0, 3.0, number, parabola)`
  the function *parabola* will be used whenever `f` appears.
- In using function parameters, the actual functions (`circle` or `parabola`) must be of the type as the formal parameter (`f`). In this case, the functions use one double-precision parameter, and they returned a double.

Sample Run of `trap-5.c`*:*

```
Program approximates the area under a function using the Trapezoidal Rule.
Enter number of subintervals to be used: 100

Approximation of 1/4 area of circle of radius 2 is 3.14042 .

Approximation of area under y = x^2 between 1 and 3 is 8.66680 .
```

Program 19: genfile.c

This program illustrates the creation and printing of files.

```
/* program to write 10 random integers to "integer.file"          *
 *             and 10 random real numbers to "real.file"          */


#include <stdio.h>  /* library for both keyboard and keyboard I/O */

#include <stdlib.h> /* libraries for the random number generator  */
#include <time.h>
/*   time, from time.h,  returns a value based upon the time of day         *
 *   rand , from stdlib.h, returns a random integer between 0 and MaxRandInt*
 *      the value of MaxRandInt is machine dependent:                       *
 *          2^32-1 = 2147483647 for SparkStations,                          *
 *          2^15-1 = 32767 for IBM Xstation 140s and HP 712/60s             */
#define MaxRandInt  32767

int main (void)
{  FILE  *file1, *file2; /* output files */
   int i;
   int ival;
   double rval;
   double MaxRandReal = (double) MaxRandInt;  /* make MaxRandInt a real */

   /* initialize random number generator                                *
    * change seed to the random number generator, based on time of day  */
   printf ("initializing random number generator\n");
   srand (time ((time_t *) 0) );

   /* place integer values on first file */
   printf ("generating file of integers\n");
   file1 = fopen ("integer.file", "w"); /* open file for writing */

   for (i = 1; i <= 10; i++)
       { ival = rand ();    /* two lines could be abbreviated */
         fprintf (file1, "%d\n", ival);}
   fclose (file1);

   /* place real values on second file */
   printf ("generating file of reals\n");
   file2 = fopen ("real.file", "w");

   for (i = 1; i <= 10; i++)
       { fprintf (file2, "%f\n", rand() / MaxRandReal);}
   fclose (file2);

   printf ("writing of files completed\n");
   return 0;
}
```

Annotations for `genfile.c`*:*

- File I/O is analogous to keyboard I/O, except for the initial steps of opening and closing the file.

- Output files are declared as pointer variables to elements of type `FILE`.
- Once a file `f` is declared, it is opened or closed with the statements
  `<variable> = fopen (<file-name>, <mode>)` and `fclose (<file-variable>)`, respectively. The `<file-name>` is any string enclosed in double quotes; the `<mode>` is
  - § "w" for writing to a file,
  - § "a" for appending additional information to an existing file, and
  - § "r" for reading from a file (as in the next example).
- Output to a file uses `fprintf` operator, which works the same as `printf` with a file variable added.

Sample Run of `genfile.c`*:*

```
babbage% gcc -o genfile genfile.c
babbage% genfile
initializing random number generator
generating file of integers
generating file of reals
writing of files completed
babbage% cat integer.file
16434
28836
18114
7586
11166
5920
6577
7990
17474
5539
babbage% cat real.file
0.432630
0.043611
0.737999
0.732139
0.254158
0.067476
0.830164
0.044099
0.038179
0.009583
```

Program 20: readfiles.c

This program shows how to read numbers from files.

```
/* program to read 10 integers from "integer.file"       *
 *            and 10 real numbers from "real.file"       */

#include <stdio.h>

int main (void)
{  FILE *intfile, *realfile;

   int i;
   double r;

   /* read integer values from "integer.file" */
   printf ("The integers from 'integer.file' follow:\n");
   intfile = fopen ("integer.file", "r");  /* open file for reading */

   while (fscanf (intfile, "%d", &i) != EOF)
      /* continue until scanf returns EOF */
      { printf ("%10d\n", i);
      }

   fclose (intfile);

   printf ("Integer file processed\n");

   /* read real values from "real.file" */
   printf ("The real numbers from 'real.file' follow:\n");

   realfile = fopen ("real.file", "r");

   while(fscanf (intfile, "%lf", &r) != EOF) {
     printf ("%10.5f\n", r);
   }

   fclose (realfile);

   printf ("Real file processed\n");

   return 0;
}
```

Annotations for `readfiles.c`*:*

- Input files are declared of type `FILE *`.
- As with files for output, input files `f` are opened and closed with the statements `<variable> = fopen (<file-name>, "r")` and `fclose (<file-variable>)`, respectively. (This format is exactly the same as for output files, except that the mode used in the `fopen` statement is `"r"`).
- Similarly, input from a file may use the `fscanf` procedure, which is analogous to reading keyboard input with `scanf`.

- When using `fscanf` for reading, the procedure returns an integer as follows:
  - § When values are read successfully according to the format string, `fscanf` returns the number of variables actually read.
  - § When `fscanf` encounters the end of a file, `fscanf` returns the special value `EOF`, which is defined in `stdio`.

  Thus, testing `(fscanf ( ...  ) != EOF)` provides a natural way to continue processing until the end of a file is encountered.

Sample Run of `readfiles.c`*:*

```
babbage% gcc -o readfiles readfiles.c && readfiles
The integers from 'integer.file' follow:
     16434
     28836
     18114
      7586
     11166
      5920
      6577
      7990
     17474
      5539
Integer file processed
The real numbers from 'real.file' follow:
   0.43263
   0.04361
   0.73800
   0.73214
   0.25416
   0.06748
   0.83016
   0.04410
   0.03818
   0.00958
Real file processed
```

Code Segment 21: fileletters-1.c

A program that reads a file name and counts the letters and other characters in that file.

```
/* Program to count the number of times each letter in a file occurs *
 * Version 1:  reading file name within program, all work in main    */

#define MAX 20                /* maximum length of file name            */
#include <stdio.h>            /* library for file and keyboard I/O      */
#include <ctype.h>            /* library for testing character data     */

int main (void)
{ FILE *file_var;
  char file_name[MAX];        /* string array for file name */
  int counts[26];             /* counts[0]..counts[25] for individual letters. */
  int non_letters = 0;        /* counter for non-letters  */
  int control_chars = 0;      /* counter for control characters (e.g., eol) */
  int i;                      /* loop index */
  char ch;                    /* character read from file */

  printf ("This program counts letter frequencies in a file\n");

  /* initialization */
  for (i=0; i<26; i++)
     counts[i] = 0;

  printf ("Enter file name: ");
  scanf ("%s", file_name);
  file_var = fopen (file_name, "r");

  /* processing the file */
  while ((ch = getc (file_var)) != EOF)
    { if (islower(ch))
         counts[ch - 'a']++;
      else if (isupper(ch))
         counts[ch - 'A']++;
      else if (iscntrl(ch))
         control_chars++;
      else non_letters++;
    }
  fclose (file_var);  /* finish up file */

  /* printing results */
  printf ("Frequency counts of letters in %s.\n\n", file_name);

  for (i='a'; i<= 'z'; i++)   /* integer i takes character values */
    printf ("  %c", i);      /* print integer value in integer format */
  printf ("\n");

  for (i=0; i < 26; i++)
    printf ("%3d", counts[i]);

  printf ("\nNon-letters:  %d\n", non_letters);
  printf ("Control characters:  %d\n", control_chars);
}
```

Annotations for `fileletters-1.c`*:*

- Technically, a string is an array of characters, and the declaration of any array requires a specification of a maximum length. Within C, however, strings are treated as a sequence of characters ending with a null character `'\0'`. Thus, the string is considered to stop with the first null, regardless of how large the array is declared. To be safe, one must be sure to declare the array to be sufficiently large.
- Printing and reading of strings utilizes "%s" format within `scanf` and `printf`.
- Whereas all simple types (e.g., `int`, `double`, and `char`) are passed by value in C, all structured types (e.g., arrays) are passed by reference. That is, the base address of the structure is passed as the parameter.
  - § At a technical level, the passing of the base address requires little copying, whereas passage by value would require the considerable overhead of copying an entire structure. For efficiency, C uses just the address of the structure, so structured variables are passed by reference.
  - § At a practical level, use of the *address operator &* is rarely needed in passing structures as variables. Thus, here `file_name` is used as parameter to `scanf`, without the &.
- While `fscanf` could be used to read character data using "format, white space would be omitted. With this option for this program, processing would give correct results for letter frequencies, but counts of non-letters and control characters would not include spaces, tabs, returns, and other white space characters.
- The function `getc` from *stdio.h* returns the next character in a file. In this reading, the end-of-file character `EOF` is considered to be just another characer. `putc` is the corresponding function to print a character to a file.
- The condition for the `while` statement combines reading, assignment, and comparison within a single step. First, `getc(file_var` returns a character, and that character is assigned to variable `ch`. This value then is compared with `EOF` within the `while` condition.
- The library *ctype.h* contains several functions that are useful in working with characters in C. In the following table, parameter `c` is considered to be a character (type `char`):

"boolean" functions	
`isupper(c)`	upper case letter
`islower(c)`	lower case letter
`isalpha(c)`	upper or lower case letter
`isdigit(c)`	decimal digit
`isalnum(c)`	upper or lower case letter or a digit
`isspace(c)`	space, formfeed, newline, carriage return, tab, vertical tab
`ispunct(c)`	printing character except space, letter, or digit
"conversion functions	
`tolower(c)`	convert upper case to lower case; no effect on other chars
`toupper(c)`	convert lower case to upper case; no effect on other chars

- In C, characters are stored as coded integers. Thus, characters may be used within integer expressions, and the expression `ch - 'a'` subtracts the code for the letter `'a'` from the letter code read into `ch`.
- When an integer appears with `printf`or `fprintf` using "format, the character with that code is printed.

Sample Run of `fileletters-1.c`*:*

```
babbage% cat character.file
&ABCDEFGHIJKLMNOPQRSTUVWXYZ
A&BCDEFGHIJKLMNOPQRSTUVWXY
AB&CDEFGHIJKLMNOPQRSTUVWX
ABC&DEFGHIJKLMNOPQRSTUVW
ABCD&EFGHIJKLMNOPQRSTUV
ABCDE&FGHIJKLMNOPQRSTU
ABCDEF;GHIJKLMNOPQRST
ABCDEFG;HIJKLMNOPQRS
ABCDEFGH;IJKLMNOPQR
ABCDEFGHI;JKLMNOPQ
ABCDEFGHIJ-KLMNOP
ABCDEFGHIJK-LMNO
ABCDEFGHIJKL-MN
ABCDEFGHIJKLM-
ABCDEFGHIJKL
ABCDEFGHIJK
ABCDEFGHIJ
ABCDEFGHI
ABCDEFGH
ABCDEFG
ABCDEF
ABCDE
ABCD
ABC
AB
A
?
&abcdefghijklmnopqrstuvwxyz
a&bcdefghijklmnopqrstuvwxy
ab&cdefghijklmnopqrstuvwx
abc&defghijklmnopqrstuvw
abcd&efghijklmnopqrstuv
abcde&fghijklmnopqrstu
abcdef;ghijklmnopqrst
abcdefg;hijklmnopqrs
abcdefgh;ijklmnopqr
abcdefghi;jklmnopq
abcdefghij-klmnop
abcdefghijk-lmno
abcdefghijkl-mn
abcdefghijklm-
abcdefghijkl
abcdefghijk?
babbage% gcc -o fileletters-1 fileletters-1.c && fileletters-1
This program counts letter frequencies in a file
Enter file name: character.file
Frequency counts of letters in character.file.

  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
 42 41 40 39 38 37 36 35 34 33 32 30 28 26 24 22 20 18 16 14 12 10  8  6  4  2
Non-letters:  30
Control characters:  43
```

- Processing counts the letters in file `character.file`. This file was created to make testing easy: subsequent lines remove one line from the alphabetic listing, both upper and lower case letters are included, the lower case listing contains 10 fewer lines than the upper case listing, and exactly 30 non-letters are included (plus <return> characters at the end of each of the 43 lines).

Program 22: fileletters-2.c

A program, organized by procedures, to count letter frequencies in a file.

```
/* Program to count the number of times each letter in a file occurs *
 * Version 2:  with main organized into procedures                   */

#define MAX 20              /* maximum length of file name           */
#include <stdio.h>          /* library for file and keyboard I/O     */
#include <ctype.h>          /* library for testing character data    */

void initialize (int counts[], char file_name[], FILE **filev);
/* procedure to initialize all program variables */

void process_file (FILE **filev, int counts[], int *others, int *control);
/* procedure to read file and determine character frequency counts */

void print_results (char name[], int counts [], int others, int control);
/* procedure to print the frequency counts in various categories */

int main (void)
{ FILE *file_var;
  char file_name[MAX];      /* string array for file name */
  int counts[26];           /* counts[0]..counts[25] for individual letters. */
  int non_letters = 0;      /* counter for non-letters  */
  int control_chars = 0;    /* counter for control characters (e.g., eol) */

  printf ("This program counts letter frequencies in a file\n");
  initialize (counts, file_name, &file_var);
  process_file (&file_var, counts, &non_letters, &control_chars);
  print_results (file_name, counts, non_letters, control_chars);
}

void initialize (int counts[], char file_name[], FILE **filev)
/* procedure to initialize all program variables */
{ int i;
  for (i=0; i<26; i++)
     counts[i] = 0;
  printf ("Enter file name: ");
  scanf ("%s", file_name);
  (*filev) = fopen (file_name, "r");
}

void process_file (FILE **filev, int counts[], int *others, int *control)
/* procedure to read file and determine character frequency counts */
{  char ch;
   while ((ch = getc (*filev)) != EOF)
    { if (islower(ch))
          counts[ch - 'a']++;
      else if (isupper(ch))
         counts[ch - 'A']++;
      else if (iscntrl(ch))
         (*control)++;
      else (*others)++;
    }
  fclose (*filev);  /* finish up file */
}
```

```
void print_results (char name[], int counts[], int others, int control)
/* procedure to print the frequency counts in various categories */
{ int i;
  printf ("Frequency counts of letters in %s.\n\n", name);

  for (i='a'; i<= 'z'; i++)    /* integer i takes character values */
    printf ("  %c", i);       /* print integer value in integer format */
  printf ("\n");

  for (i=0; i < 26; i++)
    printf ("%3d", counts[i]);

  printf ("\nNon-letters:  %d\n", others);
  printf ("Control characters:  %d\n", control);
}
```

Annotations for `fileletters-2.c`*:*

- This program organizes the file processing of *fileletters-1.c* into three stages, each coordinated by a separate procedure. The steps for each stage follows those in the earlier version.
- As in previous examples using procedures, no global variables are used here, parameters are used consistently to move values in and out of procedures, and function prototypes appear at the program's start.
- The `print_results` procedure takes values and prints them, without changing any values. Thus, the procedure uses default parameter passing throughtout.
  - § For arrays `name` and `counts`, C will pass the array's base address, while C will copy values of the individual values for `others` and `control`.
  - § When declaring array parameters, the size of the array is omitted from the procedure header. Since only the base address is passed, the procedure only knows the start of the array, not its size. (Further, the same procedure could be called for arrays of different size, although that possibility is not needed in this application.)
- The `initialize` procedure must place the correct first values into the `counts` array and the file pointer `file_var`. Since arrays are structures, C passes `counts` by reference, so any initialization made within the procedure will be stored directly within the array. In contrast, `file_var` is a pointer to a file, and initialization must provide this pointer with the address of a newly created file buffer. Thus, we must pass the pointer `file_var` by reference:
  - § Since `file_var` was declared as type `FILE *`, passage by reference must give the address of this pointer. Thus, the header specifies type `FILE **` as a pointer to the file pointer.
  - § The address of `file_var` is needed as the actual parameter, so an ampersand & is used in the call: `initialize (counts, file_name, &file_var);`.
  - § In opening the file, we store the buffer address in the pointer variable `(*filev)`.
- The `process_file` procedure changes the status of the file buffer (`FILE *file_var`) as well as the frequency array `counts` and frequencies `non_letters` and `control_chars`. Thus, parameters are passed for `process_file` in the same way as for `initialize`.
- This program again produces the same output as *fileletters-1.c*.

Code Segment 23: fileletters-3.c

The following variation allows the file name to be included on the command line. The program also utilizes C's *string.h* library for strings.

```
/* Program to count the number of times each letter in a file occurs *
 * Version 3:  with the possibility of command-line input           */

#define MAX 20              /* maximum length of file name            */
#include <stdio.h>          /* library for file and keyboard I/O      */
#include <ctype.h>          /* library for testing character data     */
#include <string.h>         /* library for common string functions    */

void process_file (FILE **filev, int counts[], int *others, int *control);
/* procedure to read file and determine character frequency counts */

void process_char(char ch, int counts[], int *others_var, int *cntl_var);
/* procedure to increment frequency counts for the given character */

void print_results (char name[], int counts [], int others, int control);
/* procedure to print the frequency counts in various categories */

int main (int argc, char *argv[])
{ FILE *file_var;
  char file_name[MAX];      /* string array for file name */
  int counts[26]           /* counts[0]..counts[25] for individual letters. */
    = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};  /*initialize array with 0's */
  int non_letters = 0;      /* counter for non-letters  */
  int control_chars = 0;    /* counter for control characters (e.g., eol) */

  printf ("This program counts letter frequencies in a file\n");

  /* determine file name from command line, if present, or ask user */
  if (argc > 1)
     strcpy (file_name, argv[1]);/* copy command-line string to file_name */
  else {
     printf ("Enter file name: ");
     scanf ("%s", file_name);}

  /* work with file and file data */
  file_var = fopen (file_name, "r");
  process_file (&file_var, counts, &non_letters, &control_chars);
  fclose (file_var);  /* finish up file */

  /* report frequencies */
  print_results (file_name, counts, non_letters, control_chars);
}

void process_file (FILE **filev, int counts[], int *others, int *control)
/* procedure to read file and determine character frequency counts */
{  char ch;
   while ((ch = getc (*filev)) != EOF)
     process_char (ch, counts, others, control);
}
```

```
void process_char (char ch, int counts[], int *others_var, int *cntl_var)
/* procedure to increment frequency counts for the given character */
{ if (islower(ch))
       counts[ch - 'a']++;
    else if (isupper(ch))
       counts[ch - 'A']++;
    else if (iscntrl(ch))
       (*cntl_var)++;
    else (*others_var)++;
}

void print_results (char name[], int counts[], int others, int control)
/* procedure to print the frequency counts in various categories */
{ int i;
  printf ("Frequency counts of letters in %s.\n\n", name);

  for (i='a'; i<= 'z'; i++)    /* integer i takes character values */
    printf ("  %c", i);       /* print integer value in integer format */
  printf ("\n");

  for (i=0; i < 26; i++)
    printf ("%3d", counts[i]);

  printf ("\nNon-letters:  %d\n", others);
  printf ("Control characters:  %d\n", control);
}
```

Annotations for `fileletters-3.c`*:*

- This program provides the user with the option of supplying the file name on the command line, when first running the program (see the sample output that follows).
- Arguments on the command line are passed to the `main` program from the operating system as parameters. When using parameters:
  - § The first parameter is an integer which indicates how many parameters are passed to the program. Hence, the integer `argc` will indicate how many parameters are supplied on the command line.
  - § The second parameter `argv` is a pointer to an array of command line arguments.
  - § Since the program name is always the first command-line parameter, `argc` always is at least 1, and `argv[0]` is the program name in string form.
  - § Any arguments after the program name appear as strings in `argv[1]` and subsequent array elements. Here, `argv[1]` contains the file name from the command line, if given by the user.
- Library *string.h* provides many useful procedures for working with strings. This program uses `strcpy` to copy the second string to the first.
- At declaration, arrays may be initialized by listing appropriate values in braces { }, with individual values separated by commas.
- In this program, procedure `process_file` calls `process_char` to handle processing of a specific character. Here, `ch` is supplied, and frequency counts are updated as needed.
  - § As frequency variables must be updated, all parameters must be passed by reference: `process_char`'s header shows that array `counts` is passed as a base address, while addresses are passed for both `otherss_var` and `cntl_var`.
  - § In the header of `process_file`, `others` and `control` are already declared as addresses. Thus, no additional address operator is needed in the call to `process_char`; `others` and `control` may be passed directly.

Sample Run of `fileletters-3.c`*:*

```
babbage% fileletters-3
This program counts letter frequencies in a file
Enter file name: character.file
Frequency counts of letters in character.file.

  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
 42 41 40 39 38 37 36 35 34 33 32 30 28 26 24 22 20 18 16 14 12 10  8  6  4  2
Non-letters:  30
Control characters:  43
babbage% fileletters-3 character.file
This program counts letter frequencies in a file
Frequency counts of letters in character.file.

  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
 42 41 40 39 38 37 36 35 34 33 32 30 28 26 24 22 20 18 16 14 12 10  8  6  4  2
Non-letters:  30
Control characters:  43
```

- When the program is run as before (with no file name on the command line), the program prompts for the file name as in past runs.
- When a file name is provided on the command line, that file is opened and processed. The program does not ask for an additional file name.