Toward Developing Good Programming Style

C version, August 1997
(McCann)

Every program you write that you intend to keep around for more than a couple of hours ought to have documentation in it. Don't talk yourself into putting off the documentation. A program that is perfectly clear today is clear only because you just wrote it. Put it away for a few months, and it will most like take you a while to figure out what it does and how it does it. If it takes you a while to figure it out, how long would it take someone else to figure it out?

Programming style is a term used to describe the effort a programmer should take to make his or her code easy to read and easy to understand. Good organization of the code and meaningful variable names help readability, and liberal use of comments can help the reader understand what the program does and why.

Probably the best way to demonstrate the value of good style is with a simple example. (Even if you don't know C very well yet, keep reading; you can benefit from this example even if you can't understand it.) Take a look at this program:

```
#include <stdio.h>
int main(void) {
int seg[10] = {6,2,5,5,4,5,6,3,7,6};
int d1, d2, d3, d4, m=0, td, ts;
for (d1=0; d1<2; d1++)
  for (d2=0; d2<10; d2++)
          for (d3=0; d3<6; d3++)
 for (d4=0; d4<10; d4++)
    if ((!((d1==0)&&(d2==0))) && (!((d1==1)&&(d2>2)))) {
           if (d1==0) {
ts = seg[d2] + seg[d3] + seg[d4];
       td = d2 + d3 + d4;
  if (ts == td) { m++;
printf(" %1d:%1d%1d\n",d2,d3,d4); }
       } else {
ts = seg[d1] + seg[d2] + seg[d3] + seg[d4];
 td = d1 + d2 + d3 + d4;
        if (ts == td) { m++;
          printf("%1d%1d:%1d%1d\n",d1,d2,d3,d4); }
} }
return 0; }
```

Do you have the faintest idea of what it accomplishes? How long would it take you to explain how it works? More to the point: If you were assigned to fix a problem with this code, how frustrating would the task be to you and how often would you curse the programmer who wrote it? Now imagine that you wrote it and some other programmer was cursing you! (Worse: Imagine you wrote this, it didn't work, and you had to ask your instructor for help in finding the problem. If I were your instructor, I'd flat-out refuse to look at this until you cleaned it up.)

Style Principle
Structure and document your program the way you wish other programmers would.

INDENTATION

(For a more detailed treatment of code indentation, please see the Indenting C Programs page.)

One immediate problem that this program has is that it does not adhere to a consistent indentation pattern. There are dozens of indentation styles that you could adopt, and some general ideas are common to most of them:

1. The style is applied consistently throughout the program.
2. Code within a block (e.g., inside a loop, or in the body of a subprogram) should be indented.
3. If a block is nested within another block the inner block's body should be indented relative to the enclosing block.
4. Avoid excessive "stairstep" indentation (such as you often see with groups of nested IF statements) because this will force you to attempt to squeeze code to fit on just the right half of the screen/page. If stairstepping becomes a problem, reduce the number of spaces per indentation (from 8 to 4, for example) or switch to a vertical style temporarily.

Here's the code from above with a consistent indentation applied. I also took the liberty of adding a little "white space" (blank lines) to help set off sections of the program. I think you'll agree that this is an improvement, but not yet acceptable:

```
#include <stdio.h>

int main(void)
{
        int seg[10] = {6,2,5,5,4,5,6,3,7,6};
        int d1, d2, d3, d4, m=0, td, ts;

        for (d1=0; d1<2; d1++)
                for (d2=0; d2<10; d2++)
                        for (d3=0; d3<6; d3++)
                                for (d4=0; d4<10; d4++)
                                        if ((!((d1==0)&&(d2==0))) &&
                                            (!((d1==1)&&(d2>2))))
                                        {
                                                if (d1==0)
                                                {
                                                        ts = seg[d2] + seg[d3]
                                                        + seg[d4];
                                                        td = d2 + d3 + d4;
                                                        if (ts == td)
                                                        {
                                                                m++;
                                                                printf(" %1d:"
                                                                "%1d%1d\n",
```

```
                                                        d2,d3,d4);
                                                }
                                        }
                                        else
                                        {
                                                ts = seg[d1] + seg[d2]
                                                + seg[d3] + seg[d4];
                                                td = d1 + d2 + d3 + d4;
                                                if (ts == td)
                                                {
                                                        m++;
                                                        printf("%1d"
                                                        "%1d:%1d%1d\n",
                                                        d1,d2,d3,d4);
                                                }
                                        }
                                }
        return 0;
}
```

Do you see the stairstep effect caused by the indentation of the nested FOR loops? (Heck, how could you miss it?) Add the IF statements to the code and there isn't much space left for code on each line if you're trying to stay within the 80 column limit of most screens. Here's the program with a vertical style applied to the loops (note that a similar procedure can be applied to closely nested IF statements as well):

```
#include <stdio.h>

int main(void)
{
        int seg[10] = {6,2,5,5,4,5,6,3,7,6};
        int d1, d2, d3, d4, m=0, td, ts;

        for (d1=0; d1<2; d1++)
        for (d2=0; d2<10; d2++)
        for (d3=0; d3<6; d3++)
        for (d4=0; d4<10; d4++)
            if ((!((d1==0)&&(d2==0))) && (!((d1==1)&&(d2>2))))
            {
                    if (d1==0)
                    {
                            ts = seg[d2] + seg[d3] + seg[d4];
                            td = d2 + d3 + d4;
                            if (ts == td)
                            {
                                    m++;
                                    printf(" %1d:%1d%1d\n",d2,d3,d4);
                            }
                    }
                    else
```

```
                        {
                                ts = seg[d1] + seg[d2] + seg[d3] + seg[d4];
                                td = d1 + d2 + d3 + d4;
                                if (ts == td)
                                {
                                        m++;
                                        printf("%1d%1d:%1d%1d\n",d1,d2,d3,d4);
                                }
                        }
                }
        return 0;
}
```

This looks much better because there's now room to indent the nested IF statements with enough room left over to do a decent job with the statements inside the IFs. Not everyone likes this approach, however. Some people prefer that programmers move the body of the loops to a subprogram instead. In my view this is acceptable only if that section of code can logically stand on its own. Even if this code-relocation idea doesn't sit right with you, play along; having a subprogram will be handy as an example later on. Here's the complete program with this philosophy applied:

```
#include <stdio.h>

void count(int, int, int, int, int*);

int main(void)
{
        int d1, d2, d3, d4, m=0;

        for (d1=0; d1<2; d1++)
                for (d2=0; d2<10; d2++)
                        for (d3=0; d3<6; d3++)
                                for (d4=0; d4<10; d4++)
                                        count(d1,d2,d3,d4,&m);

        return 0;
}

void count (int d1, int d2, int d3, int d4, int *m)
{
        int seg[10] = {6,2,5,5,4,5,6,3,7,6};
        int ts,td;

        if ((!((d1==0)&&(d2==0))) && (!((d1==1)&&(d2>2))))
        {
                if (d1==0)
                {
                        ts = seg[d2] + seg[d3] + seg[d4];
                        td = d2 + d3 + d4;
                        if (ts == td)
```

```
                        {
                                (*m)++;
                                printf(" %1d:%1d%1d\n",d2,d3,d4);
                        }
                }
                else
                {
                        ts = seg[d1] + seg[d2] + seg[d3] + seg[d4];
                        td = d1 + d2 + d3 + d4;
                        if (ts == td)
                        {
                                (*m)++;
                                printf("%1d%1d:%1d%1d\n",d1,d2,d3,d4);
                        }
                }
        }
}
```

Please note that the 'seg' variable was moved into the subprogram because it is not used by the main routine at all.

MEANINGFUL VARIABLE NAMES

Another impediment to program readability is that the program's identifiers (variable names, subprogram names, etc.) are mostly meaningless. You should strive to give each object a name that gives the reader a strong hint as to the object's purpose within the program. Many early languages limited the size of the allowable names, and that forced programmers to use short, cryptic names. Modern languages permit identifier names to be quite lengthy, so there's no excuse not to create good names. As with indentation, there are some principles that apply to naming:

1. Use good, meaningful names, but don't go overboard. If you have a variable in your program that holds the number of hours an employee works in a week, you might call it HOURS, although that name still leaves a lot of doubt as to the exact contents of the variable. On the other hand, a name such as HOURS_WORKED_IN_A_WEEK is much more descriptive but contains 22 symbols; a simple increment of the variable might fill an entire line! A compromise such as HOURS_PER_WEEK is a good solution, though there are others (see below).
2. Many languages now permit you to use underscores as part of names, as shown above. If you can't use them, you can still improve name readability by mixing the case of the letters in the name. For example, 'HoursPerWeek' is much easier to read than 'hoursperweek'.
3. Always place a comment with each variable name declaration. The comment should give a brief phrase or sentence that explains the purpose of the variable. If the variable name itself isn't enough to make the purpose of the variable clear to the reader, the comment should clear up any confusion. (At the same time, you still want to select good names; the reader doesn't want to have to keep referring to the declaration comment to refresh his or her memory of a variable's purpose.
4. Common abbreviations are often acceptable in variable names. For example: HRS_PER_WEEK. But don't get carried away; HRS_P_WK simply is not a good name.
5. Although this isn't really a point about naming, it is related: Don't reuse variables in a subprogram. For example, you may need a loop control integer variable at the top of the block, and you might also need a place to store a value for a while at the bottom of the block. Resist the temptation to reuse that loop control variable as the temporary holder. Such reuses can lead to a reduction in readability of the code as well as to confusion in program debugging.

Here's our program with better variable names and declaration comments:

```
#include <stdio.h>

void count_segments(int, int, int, int, int*);

int main(void)
{
        int hour1, /* the first (leftmost) digit in the hour two-digit pair */
            hour2, /* the second (rightmost) digit in the hour */
            tens,  /* the ten's digit in the minute two-digit pair */
            ones,  /* the one's digit in the minute */
            matches=0;  /* the count of the times the sums match */

        for (hour1=0; hour1<2; hour1++)
                for (hour2=0; hour2<10; hour2++)
                        for (tens=0; tens<6; tens++)
                                for (ones=0; ones<10; ones++)
                                        count_segments(hour1,hour2,
                                                       tens,ones,&matches);

        return 0;
}

void count_segments (int hour1, int hour2, int tens, int ones, int *matches)
{
        int segments[10] = {6,2,5,5,4,5,6,3,7,6};
                                /* array of the number of segments
                                   needed to display each digit */
        int total_segments,     /* number of segments used in the time */
            total_digits;       /* sum of the digits in the time value */

        if ((!((hour1==0)&&(hour2==0))) && (!((hour1==1)&&(hour2>2))))
        {
                if (hour1==0)
                {
                        total_segments = segments[hour2] + segments[tens]
                                + segments[ones];
                        total_digits = hour2 + tens + ones;
                        if (total_segments == total_digits)
                        {
                                (*matches)++;
                                printf(" %1d:%1d%1d\n",hour2,tens,ones);
                        }
                }
                else
                {
                        total_segments = segments[hour1] + segments[hour2]
                                + segments[tens] + segments[ones];
```

```
                        total_digits = hour1 + hour2 + tens + ones;
                        if (total_segments == total_digits)
                        {
                                (*matches)++;
                                printf("%1d%1d:%1d%1d\n",
                                        hour1,hour2,tens,ones);
                        }
                }
        }
}
```

Notice that the variable names are a little bit on the cryptic side, and the declarations are commented to provide clarification. In my view this is a reasonable compromise. As we'll see, some variables are difficult to name, and a short but slightly cryptic name with a good declaration comment is cleaner than a descriptive but lengthy and awkward name. Also notice that the longer variable names forced us to split some lines and reorganize others to get them to fit in the 80 column lines. Readability does have its price.

One place where short, cryptic variable names are often used: Loop control variables. These are often difficult to name, and most programmers will simply use single letter names such as I and J for them. (Why I and J? Those letters are frequently used for integer subscripts in mathematics. The early scientific language FORTRAN was designed so that variables whose names started with I and J (and a few others) were of type INTEGER by default, which made them easy to use as integer loop control variables. The habit may never fade.)

INTERNAL DOCUMENTATION

The variable declaration comments are one part of good internal documentation. Internal documentation is the set of comments that are included within the code to help clarify algorithms. Some students take internal documentation to mean that they should comment each line of code! This is obviously an example of overdoing a good idea. Any programmer knows how to increment a value in a variable; there's no reason to explain trivial operations such as that. The value of some good internal documentation should be clear by looking at the latest version of our sample program. Even with the good code organization and variable names, the function of this program is still not obvious.

Here's a list of items that should be included in your internal documentation:

1. "Block comments" (comments that are several lines long) should be placed at the head of every subprogram. These will include the subprogram name; the purpose of the subprogram; and a list of all parameters, including direction of information transfer (into this routine, out from the routine back to the calling routine, or both), and their purposes.
2. Meaningful variable names. In a nod to tradition, simple loop variables may have single letter variable names, but all others should be meaningful. Never use nonstandard abbreviations.
3. Each variable and constant must have a brief comment next to its declaration that explains its purpose. This applies to all variables, as well as to fields of struct declarations.
4. Complex sections of code and any other parts of the program that need some explanation should have comments just ahead of them or embedded in them.

A critical point: Documentation, and internal documentation in particular, should be written and included in the program as the code is being written. Students tend to get in the habit of writing the code and then tossing in some documentation only if they have time before the due date. This makes documenting seem even more boring and tedious that it already is, and students who rush the documentation at the last minute usually do a very mediocre job. Documentation should be written when the code is being written, and should be typed in as the code is typed in.

To demonstrate some of these points, here's yet another version of our program, this time containing some acceptable internal documentation:

```
#include <stdio.h>

void count_segments(int, int, int, int, int*);

int main(void)
{
        int hour1, /* the first (leftmost) digit in the hour two-digit pair */
            hour2, /* the second (rightmost) digit in the hour */
            tens,  /* the ten's digit in the minute two-digit pair */
            ones,  /* the one's digit in the minute */
            matches=0;  /* the count of the times the sums match */

        for (hour1=0; hour1<2; hour1++)
                for (hour2=0; hour2<10; hour2++)
                        for (tens=0; tens<6; tens++)
                                for (ones=0; ones<10; ones++)
                                        count_segments(hour1,hour2,
                                                       tens,ones,&matches);

        return 0;
}

        /*------------------------------------------------ COUNT_SEGMENTS -----
          |  Function COUNT_SEGMENTS
          |
          |  Purpose:  COUNT_SEGMENTS computes the number of segments a
          |      digital clock will need to display the time given by
          |      the parameters. It then computes the sum of the digits
          |      and compares the two totals.  If they match, the success
          |      is recorded by incrementing the sum 'matches' and by
          |      displaying the time.
          |              The number of segments a digital clock uses to
          |      display any of the ten numbers 0-9 is stored in the array
          |      'segments'.  The array is indexed by the digit; thus, the
          |      number of segments needed to display a '0' is in element [0].
          |
          |  Parameters:
          |      hour1 (IN) - In a two-digit hour value, this is the leftmost
          |              digit.  Ex:  In the time 12:34, hour1 would hold 1.
          |      hour2 (IN) - In a two-digit hour value, this is the rightmost
          |              digit.  Ex:  In the time 12:34, hour2 would hold 2.
          |      tens (IN) - In a two-digit minute value, this is the leftmost
          |              digit.  Ex:  In the time 12:34, tens would hold 3.
          |      ones (IN) - In a two-digit minute value, this is the rightmost
          |              digit.  Ex:  In the time 12:34, ones would hold 4.
          |      matches (IN/OUT) - The sum of the times the number of segments
```

```
         |              equals the sum of the digits.
         |
         |  Returns:  Nothing.  (This is a void function.)
         |
         *-----------------------------------------------------------------*/

void count_segments (int hour1, int hour2, int tens, int ones, int *matches)
{
        int segments[10] = {6,2,5,5,4,5,6,3,7,6};
                                /* array of the number of segments
                                   needed to display each digit */
        int total_segments,     /* number of segments used in the time */
            total_digits;       /* sum of the digits in the time value */

                /* We don't want to consider times that start with
                 * '00' (like 00:35) or anything with hours over
                 * '12' (like 16:14).
                 */

        if ((!((hour1==0)&&(hour2==0))) && (!((hour1==1)&&(hour2>2))))
        {
                        /* If the time is between 12:59 and 10:00, the
                         * leftmost hour digit is 0.  A clock doesn't
                         * display that 0, so we shouldn't count its
                         * segments.
                         */

                if (hour1==0)
                {
                        total_segments = segments[hour2] + segments[tens]
                                + segments[ones];
                        total_digits = hour2 + tens + ones;
                        if (total_segments == total_digits)
                        {
                                (*matches)++;
                                printf(" %1d:%1d%1d\n",hour2,tens,ones);
                        }
                }
                else    /* Here we do count the leftmost hour digit */
                {
                        total_segments = segments[hour1] + segments[hour2]
                                + segments[tens] + segments[ones];
                        total_digits = hour1 + hour2 + tens + ones;
                        if (total_segments == total_digits)
                        {
                                (*matches)++;
                                printf("%1d%1d:%1d%1d\n",
                                        hour1,hour2,tens,ones);
                        }
                }
        }
```

```
}
```

The trick with internal documentation is to make it easy to find while at the same time ensuring that it's not making the code hard to read. Block comments can be partially boxed (as shown) to separate them from the code. The use of the '*' at the start of each line of the shorter clarifying comments in the code serves a similar purpose. There's no one right way to do this, but it does need to be done. Experiment with some styles and pick one you like. One piece of advice: Don't fall in love with the "complete box" style. Lots of students like to completely enclose the block comments within a box. This looks great, but the right-hand wall of the box is very hard to keep lined up as you make adjustments and additions to the comments. The "three wall" style shown above is much easier to deal with and looks almost as good.

EXTERNAL DOCUMENTATION

In a professional programmer's shop, large projects are documented in great detail, not only with comments in the code but with descriptions that are maintained separately from the code. In such an environment, programmers are often asked to fix problems in code that they didn't write. Many times, the author of the code isn't even with the company any longer. The documentation may be all the programmer has as reference material to help him or her make the necessary modifications.

External documentation doesn't deal with details of the code. Instead, it serves as a general description of the project, including such information as what the code does, who wrote it and when, which common algorithms it uses, upon which other programs or libraries it is dependent, which systems it was designed to work with, what form and source of input it requires, the format of the output it produces, etc. Often the external documentation will include structure charts of the outline of the program that were produced when the program was being designed. All of this information is necessary to help other programmers understand the program. One seemingly innocent change in a program can have unpredictable consequences on other parts of the system. Good documentation can help prevent such problems.

In most programming classes, it is impractical for instructors to require large amounts of external documentation for programs that are only a few hundred lines long. Instead, it is common for instructors to require that a small amount of external documentation be included at the top of the program in the form of a large block comment. This condensed version should include at least the following pieces of information:

1. Your name, the course name, assignment name/number, instructor's name, and due date.
2. Description of the problem the program was written to solve.
3. Approach used to solve the problem. This should always include a brief description of the major algorithms used, or their names if they are common algorithms.
4. The program's operational requirements: Which language system you used, special compilation information, where the input can be located on disk, etc.
5. Required features of the assignment that you were not able to include.
6. Known bugs should be reported here as well. If a particular feature does not work correctly, it is in your best interest to be honest and complete about your program's shortcomings.

The final version of the program is given at the end of this document. Look it over carefully. Do you understand what the program does? More importantly for this discussion, do you understand how it does it? If the indentation, identifier names, and documentation helped, then they were well worth the time it took the programmer to put them in. Hopefully, you'll now see the value of putting such documentation in your programs as well.

Take the time to ask yourself if you think the design of the comments is a good one; are the comments easy to find and to read? Do they distract from the code excessively? Are there too many of them to suit you, or too few? By

asking and answering questions such as these, you will begin to develop a style of your own. When you see documentation styles that you like, consider adopting them into your own style. Soon you'll have one you like, and as a result you'll be more likely to use it.

There are plenty of decisions that were made in the design and documentation of this program that can be questioned and improved on. As you gain more experience in programming, consider revisiting this program and trying to rewrite it from scratch. Perhaps you can think of a better way to generate the times, for example. There isn't a program in existence that can't be improved, and this one is certainly no exception.

MISCELLANEOUS COMMENTS

In a programming class, instructors don't want you to write the most efficient programs; they'd much rather you learn the material well and learn good programming style at the same time. Never pursue efficiency at the expense of clarity. An efficient program is better than an inefficient one, of course, but it is also true that a slow, correct program is better than a fast, buggy one. Clear, well-designed programs are more likely to be correctly functioning programs. Get the program working before you worry too much about making it work quickly.

A style topic that this document didn't cover is Top-Down Design. In TDD, the idea is to design a program by first identifying the major tasks of the program. For each task, break it down into smaller subtasks. Continue this process until it is clear how each task is to be accomplished. Each task that you have identified is a candidate to be a subprogram, with the main program consisting mostly of calls to the top-level subtasks. This approach requires that you have the discipline to plan the structure of your program before you write any code. If you can do it, the process of planning the program will help minimize the number of logical errors in your program. Typically, you'll spend less time planning than you would have spent debugging the code you didn't plan. This is a lesson most programmers can only learn by experiencing a long, late-night debugging marathon firsthand. For examples of Top-Down Design, refer to an introductory programming text. Most of them cover it in detail.

Finally, no document on style would be complete without a mention of the GOTO problem. The unconditional branch (GOTO) operation is provided in nearly all languages, and its use is frequently discouraged, particularly by instructors. When you're learning to program, it's important that you learn to avoid using a GOTO. Programs with several GOTOs can quickly become hard to understand and thus hard to repair or modify. However, in some isolated situations, a nice unconditional branch can do the job of a lot of convoluted but well-structured logic. If you ever feel the need to use a GOTO, be sure to ask your instructor if he or she will sanction its use in that situation. They may be able to show you a more structured solution to the problem.

```
/*==========================================================================
 |   Assignment:  Program #0 -- Digital Clock Digit and Segment Sums
 |
 |       Author:  [Student's Name Here]
 |     Language:  ANSI C (tested using xlc on an IBM RS/6000 running AIX)
 |   To Compile:  xlc segment.c
 |
 |        Class:  COMSC 0000
 |   Instructor:  Dr. Staff
 |     Due Date:  December 32nd, 2024, at the beginning of class
 |
 +-------------------------------------------------------------------------
 |
 |  Description:  A common digital display clock, such as an alarm clock or
 |      a digital wristwatch, creates numbers by lighting segments in a
```

```
 |      standard 7-segment display.  This program assumes that the digits
 |      look like this:
 |
 |               _       _   _       _   _   _   _   _
 |              | |   |   _|  _|  |_|  |_  |_      |  |_|  |_|
 |              |_|   |  |_   _|    |   _|  |_|     |  |_|   _|
 |
 |      Thus, 6 segments are needed to display the digit 0, for example.
 |      The program creates all legal twelve-hour times (plus some illegal
 |      ones that are logically eliminated from consideration) and for each
 |      determines if the number of segments in the displayed time equals
 |      the sum of the digits in the display.  For example, 7:21 requires
 |      10 segments and the sum of the digits is also 10 (7+2+1).  All such
 |      times are output by the program and the total number of such
 |      times is also determined.
 |
 |        Input:  No input (either from the keyboard or from a file) is
 |                required by this program.
 |
 |       Output:  The times are displayed one per line to the standard output.
 |                The number of times displayed is output at the end.
 |
 |     Algorithm:  The times are generated by a set of 4 nested FOR loops.
 |                Times are composed of 4 digits, one loop per digit.  Legal
 |                times can start with a 0 or a 1, but the 0 is never shown
 |                on a clock.  Hours go from 01 through 12, so the second
 |                digit of the hour value can range from 0 through 9, as can
 |                the rightmost digit of the minute.  The left digit in the
 |                minute value ranges from 0 through 5 only.  For each time
 |                generated by the loops, the COUNT_SEGMENTS subprogram
 |                determines if the segment sum equals the digit sum;
 |                see the internal documentation of COUNT_SEGMENTS for
 |                details on its operation.
 |
 |   Required Features Not Included:  The program adheres to all
 |                requirements stated in the program assignment, and all
 |                required features are included.
 |
 |    Known Bugs:  There are no known bugs remaining in this program.
 |
 +=================================================================*/

#include <stdio.h>

void count_segments(int, int, int, int, int*);

int main(void)
{
        int hour1, /* the first (leftmost) digit in the hour two-digit pair */
            hour2, /* the second (rightmost) digit in the hour */
            tens,  /* the ten's digit in the minute two-digit pair */
            ones,  /* the one's digit in the minute */
```

```
            matches=0;  /* the count of the times the sums match */

        for (hour1=0; hour1<2; hour1++)
                for (hour2=0; hour2<10; hour2++)
                        for (tens=0; tens<6; tens++)
                                for (ones=0; ones<10; ones++)
                                        count_segments(hour1,hour2,
                                                tens,ones,&matches);

        printf("\nThe number of times displayed is %d.\n",matches);

        return 0;
}

        /*------------------------------------------------ COUNT_SEGMENTS -----
          |  Function COUNT_SEGMENTS
          |
          |  Purpose:  COUNT_SEGMENTS computes the number of segments a
          |      digital clock will need to display the time given by
          |      the parameters. It then computes the sum of the digits
          |      and compares the two totals.  If they match, the success
          |      is recorded by incrementing the sum 'matches' and by
          |      displaying the time.
          |              The number of segments a digital clock uses to
          |      display any of the ten numbers 0-9 is stored in the array
          |      'segments'.  The array is indexed by the digit; thus, the
          |      number of segments needed to display a '0' is in element [0].
          |
          |  Parameters:
          |      hour1 (IN) - In a two-digit hour value, this is the leftmost
          |              digit.  Ex:  In the time 12:34, hour1 would hold 1.
          |      hour2 (IN) - In a two-digit hour value, this is the rightmost
          |              digit.  Ex:  In the time 12:34, hour2 would hold 2.
          |      tens (IN) - In a two-digit minute value, this is the leftmost
          |              digit.  Ex:  In the time 12:34, tens would hold 3.
          |      ones (IN) - In a two-digit minute value, this is the rightmost
          |              digit.  Ex:  In the time 12:34, ones would hold 4.
          |      matches (IN/OUT) - The sum of the times the number of segments
          |              equals the sum of the digits.
          |
          |  Returns:  Nothing.  (This is a void function.)
          |
           *-------------------------------------------------------------------*/

void count_segments (int hour1, int hour2, int tens, int ones, int *matches)
{
        int segments[10] = {6,2,5,5,4,5,6,3,7,6};
                                /* array of the number of segments
                                   needed to display each digit */
        int total_segments,     /* number of segments used in the time */
            total_digits;       /* sum of the digits in the time value */
```

```
                /* We don't want to consider times that start with
                 * '00' (like 00:35) or anything with hours over
                 * '12' (like 16:14).
                 */

        if ((!((hour1==0)&&(hour2==0))) && (!((hour1==1)&&(hour2>2))))
        {
                        /* If the time is between 12:59 and 10:00, the
                         * leftmost hour digit is 0.  A clock doesn't
                         * display that 0, so we shouldn't count its
                         * segments.
                         */

                if (hour1==0)
                {
                        total_segments = segments[hour2] + segments[tens]
                                + segments[ones];
                        total_digits = hour2 + tens + ones;
                        if (total_segments == total_digits)
                        {
                                (*matches)++;
                                printf(" %1d:%1d%1d\n",hour2,tens,ones);
                        }
                }
                else    /* Here we do count the leftmost hour digit */
                {
                        total_segments = segments[hour1] + segments[hour2]
                                + segments[tens] + segments[ones];
                        total_digits = hour1 + hour2 + tens + ones;
                        if (total_segments == total_digits)
                        {
                                (*matches)++;
                                printf("%1d%1d:%1d%1d\n",
                                        hour1,hour2,tens,ones);
                        }
                }
        }
}
```