

Chapter 7

In-program Documentation

7.1 Goals—recognizing that programs are documents

In-program documentation is the most common, and most neglected, form of writing for computer programmers. We hope that this exercise teaches you

- what information to put in in-program documentation,
- one way to format the information so that it can be used, and
- the importance of in-program documentation.

7.2 Audience assessment—maintenance programmers

Just as tailors spend more time making alterations than sewing new suits, most programmers spend more of their working time modifying old code than they do writing new code. If a program is difficult to change, it will be clumsily modified, and the resulting mess will be blamed on the original programmer. To protect your reputation, it is essential that you make your programs easy to modify cleanly.

Writing easily modifiable code is still an art, rather than a science, but all the techniques of structuring and modularization taught in programming classes help. Kevin disagrees with Niklaus Wirth, who used a book title to claim that *Algorithms+Data Structures=Programs* [Wir76]. Certainly, algorithms and data structures are necessary for programs, but interface specifications and maintenance documentation are also essential.

Programs are read by compilers and by people. Far too many programmers stop when they get a program that is readable by a compiler—totally ignoring their other audience. Who are the people who read programs? Mainly other programmers. Most often the programmers are not reading to learn the language or to admire your style, but to make specific changes to the program. They usually read only those parts of the program they think will be relevant, so you must provide cross-references to information not immediately visible.

Many programs rely on external documentation. For student work, the external documentation is often the assignment sheet. External documentation may be easier to write than in-program documentation, but is not nearly as useful to a programmer attempting to modify your code. First, the external documentation often gets separated from the source, so the programmer doesn't have it to read. Second, the source is often modified without changing the external documentation, so the two may no longer agree. It is essential that maintenance documentation be included in the source code itself.

7.3 Example—knight's tour

To give you an example of a reasonably well-documented piece of code, I am providing a solution to the knight's-tour problem [Wir76, 137–142]. The program has not been modified substantially from Wirth's code (though I was tempted to do so), but has been translated from Pascal to C.

```

program knightstour(output);
const n=5; nsq=25;
type index = 1..n;
var i,j: index;
    q: boolean;
    s: set of index;
    a,b: array [1..8] of integer;
    h: array [index, index] of integer;

procedure try(i: integer; x,y: index; var q: boolean);
    var k,u,v: integer; q1: boolean;
begin k := 0;
    repeat k := k+1; q1 := false;
        u := x+a[k]; v := y+b[k];
        if (u in s) and (v in s) then
            if h[u,v] = 0 then
                begin h[u,v] := i;
                    if i<nsq then
                        begin try(i+1,u,v,q1);
                            if not q1 then h[u,v] := 0
                                end else q1 := true
                        end
                    end
                until q1 or (k=8);
                q := q1
            end {try};

begin s := [1,2,3,4,5];
    a[1] := 2; b[1] := 1;
    a[2] := 1; b[2] := 2;
    a[3] := -1; b[3] := 2;
    a[4] := -2; b[4] := 1;
    a[5] := -2; b[5] := -1;
    a[6] := -1; b[6] := -2;
    a[7] := 1; b[7] := -2;
    a[8] := 2; b[8] := -1;
    for i := 1 to n do
        for j := 1 to n do h[i,j] := 0;
    h[1,1] := 1; try(2,1,1,q);
    if q then
        for i := 1 to n do
            begin for j := 1 to n do write(h[i,j]:5);
                writeln
            end
        else writeln('NO SOLUTION')
    end.

```

Figure 7.1: Wirth's version of the Knight's tour program in Pascal

Figure 7.1 gives the original Pascal code and Figures 7.2 through 7.7 give the documented rewrite of the program in C, each figure representing one page of the source code. (With most program editors, you can separate source code into pages by including a control-L character as a page separator.)

7.4 Assignment—adding comments to minimal code

You will have to do a similar rewrite to another piece of recursive code. The program you have to rewrite attempts to solve the following puzzle:

Put the numbers 1 through 30 in order such that every adjacent pair adds up to a perfect square.

In fact, the program goes further and attempts to solve the puzzle for 1 through n , where n is varied from 1 to 45. The program works by a simple back-tracking search, using 3 data structures: the “s” array keep tracks of the sequence of numbers, the “u” array keeps track of which numbers have been used, and the “q” array is used for determining quickly whether a particular number is a perfect square. The work is all done by the recursive “c” procedure, which tries to extend the sequence in s into a complete sequence. It is given a partial sequence which has the first “i” slots filled with numbers that satisfy the pairwise summation constraint, and returns 1 if the partial sequence can be completed to a complete sequence (leaving the complete sequence in s). It returns 0 if the partial sequence cannot be completed.

You will have to change the variable names and the procedure names. You can play with the indenting and other aspects of formatting, if you think it makes the program more readable. You may also rearrange parts of the program for better modularity, it that will make the documentation easier to write. Make sure you document the preconditions and postconditions for procedure “c”—what must be true of the globals before each call? what is true of them after each return?

Your finished product should be an executable, well-documented program. Kevin’s name should appear prominently near the beginning (as the original programmer), but you should take credit for the changes and documentation you provided.

7.5 Writing process—in-program documentation

Despite the way this assignment is structured, in-program documentation is not the last thing you do to a working program. It is most needed in the early stages of the programming, as scaffolding to support the incomplete program fragments. Whenever you write a procedure, you should write the explanation of the interface and function before writing the procedure itself. If you aren’t crystal-clear about the purpose of the procedure, you won’t be able to write it correctly.

In good programs, the documentation is written first, and the program written to match the documentation. For large programs with multiple programmers, it is particularly important to get the interfaces between modules well-defined early in the process. If the interface specification isn’t written out completely, isn’t distributed to the programmers, or isn’t comprehensible, it might as well not exist.

At this point in your academic career, you’ve probably been told a hundred times that you need to document your programs better. When pressed, your instructors have probably mumbled vaguely about wanting more documentation but not precisely what was lacking. We might not do much better at telling you what is expected, but we’ll try.

Of course, the best way to find out what is needed in the way of program documentation is to try modifying someone else’s code. Whenever you run into something you can’t figure out, or, worse yet, something you think you understand but are wrong about, then you’ve found a place where the documentation is inadequate.

To forestall such misunderstandings, you have to write far more documentation than you, as the creator of the code, believe you need. Because well-documented programs are so rare, you’ve probably never seen one and aren’t aware what they look like. Perhaps the best example is Knuth’s type-setting program \TeX [Knu86]. We have put the source code on reserve in the Science Library—take a look at it to see what a well-documented program looks like. Note that the ratio of documentation to code is large and that the documentation concentrates on explaining data structures and techniques, not on repeating what the code says.

```

/* knight's tour program
*
* Original code in Pascal by Niklaus Wirth
* For an explanation and the original code see
* Niklaus Wirth
* Algorithms+Data Structures=Programs
* Prentice-Hall 1976
* pages 137--142
*
* Translated to ANSI C by Kevin Karplus, 19 February 1993
* Further modifications made by Kevin Karplus 7 February 1999
*/

/* ABSTRACT:
* This program determines whether a knight can visit all the squares of
* a chess board exactly once, using the normal movements of chess.
* The size of the chess board is specified by compile-time constants
* (NumRows and NumCols), as is the initial position (StartRow and
* StartCol).
* There are no run-time inputs.
*
* A "tour" is a sequence of squares, each visited only once, that
* reflect a possible sequence of knight's moves.
* A "partial tour" includes some, but perhaps not all squares.
* A "complete tour" includes all the squares, and is the desired solution.
*
* The output is either a message indicating that no complete tour exists,
* or a display of the board, with a number in each square.
* The numbers range from 1 to the number of squares on the board, and
* indicate the order in which the squares were visited.
*
* The three examples of knight's tours given on page 141 of
* Algorithms+Data Structures=Programs
* can be obtained with the following settings of the constants:
*
*      NumRow  NumCol      StartRow      StartCol
*      5        5           0             0
*      5        5           2             2
*      6        6           0             0
*/

/* KNOWN BUGS and possible future changes:
*
*      ChangeRow and ChangeCol should probably be a single array
*      of 8 structs, each of which has a row and col value.
*
*      It might be cleaner to replace the PartialTour array
*      with a data structure that would hold the array, the number of
*      squares visited, and the location of the most recently visited
*      square. Even better might be to replace the recursion stack with
*      a stack of moves in the PartialTour data structure, so that
*      output could be provided in other notations, and "backtrack"
*      could be defined as an operator on PartialTours.
*      If this were C++ instead of C, PartialTour would be defined
*      as a class.
*/

```

Figure 7.2: Initial block comment for knight's tour program.

```

/* CHANGE LOG:
* 19 February 1993, Kevin Karplus
* Translated to C, variable names changed, comments added.
*
*     globals and local variables of main program:
* n=> NumRows, NumCols
* nsq => NumSquares (macro)
* type index eliminated
* i=>row made local to main()
* j=>col made local to main()
* q eliminated (using return value, no variable needed)
* s eliminated (replaced by procedure OnBoard(r,c))
* a=> RowChange, b=>ColChange (subscripts now 0..7, not 1..8)
* h=>board (made local to main, and passed as parameter.
* also, made to be of new type RectangularBoard)
*
* parameters and local variables of "try" procedure:
* try=>CompleteTour
* i=>SquaresVisited (meaning changed by 1, for easier naming)
* x,y => AtR, AtC
* q,q1 eliminated (using return value instead)
* k=>direction
* u,v => NextR, NextC
*
* Moved the SquaresVisited test down one level in the recursion,
* to make the termination of the recursion easier to understand,
* and to make the behavior correct on 1x1 boards.
* (Note: this bug fix was not part of the assignment given to the
* class, but having identified the bug in Wirth's code, I couldn't
* stand leaving it in my translation.)
*
* 23 February 1993, Kevin Karplus
*     Minor editing of comments, based on suggestions from David Patmore.
*
* 7 February 1999, Kevin Karplus
* Changed "board" to "when_visited", "RectangularBoard" to "PartialTour".
* Added "NotVisited" constant.
* Added efficiency hack to OnBoard, using unsigned ints.
* Added ASCII-art lines to make output look more like a chess board.
* Added assertions to check that compile-time constants were legal.
* Added return value from main (0 if tour found, 1 otherwise);
* Moved RowChange and ColChange arrays into CompleteTour to eliminate
* all globals from CompleteTour, and made them static const to
* avoid repeated allocation and initialization.
* Modified CompleteTour to accept the NEXT move, rather than the LAST
* move, allowing starting from an empty board.
* Eliminated NextRow and NextCol variables from CompleteTour.
*/

```

Figure 7.3: Change log for knight's tour problem.

```

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

/* COMPILE-TIME CONSTANTS and TYPES */

/* These constants define the size of the board.
 * They are macros (rather than "const int") so that they can be used
 * for specifying array dimensions.
 */
#define NumRows 5
#define NumCols 5

#define NumSquares (NumRows*NumCols) /* the number of squares to visit */

/* StartRow and StartCol give the initial position of the knight.
 * Positions range from (0,0) to (NumRows-1,NumCols-1).
 */
#define StartRow 0
#define StartCol 0

#define NotVisited 0
/* A PartialTour holds a tour or partial tour.
 * If a square has the value NotVisited, then that square has not been
 * visited yet on this partial tour.
 * Otherwise, the value is the time at which the square was visited
 * (from 1 to NumSquares for a complete tour, from 1 to n for a
 * partial tour with n squares visited).
 */
typedef PartialTour[NumRows][NumCols];

typedef short int boolean; /* used for true=1/false=0 values */

```

Figure 7.4: Type definitions and global variables for knight's-tour.

```

/* OnBoard(r,c)
 * tests to see if a coordinate pair is a legal board position.
 * Inputs: r,c a coordinate pair
 * Returns: 1 if the row and column position is on the board,
 * and 0 if it is not.
 */
boolean OnBoard(unsigned int r, unsigned int c)
{
    /* This test uses a trick to detect rows or columns less than zero.
     * Because the parameters are passed as UNSIGNED, negative numbers
     * are interpreted as very large numbers, which then fail the
     * either the r<NumRows or c< NumCols test.
     */
    return r<NumRows && c<NumCols;
}

```

Figure 7.5: OnBoard test for knight's-tour.

```

/* CompleteTour(when_visited, NextMoveNum, AtR, AtC)
 * attempts to complete the partial tour in when_visited.
 *
 * Inputs:
 * when_visited holds the partial tour
 * AtMoveNumber one more than the number of squares already visited
 * AtR, AtC the next position to try in the partial tour
 *
 * Returns: 1 if the partial tour can be completed to a full tour.
 * 0 if no complete tour starts with this partial tour.
 *
 * Outputs:
 * when_visited holds complete tour if 1 is returned, but
 * is restored to the partial tour from before call, if 0 returned.
 */
boolean
CompleteTour(PartialTour when_visited,
int AtMoveNumber, int AtR, int AtC)
{
    /* Try adding the new move to the partial tour.
     * If tour is still not complete, try all possible directions for
     * continuing the tour.
     * If no direction results in complete tour, retract the move
     * and return 0 for failure.
     */

    /* The pair (RowChange[i], ColChange[i]) is one of the eight possible
     * directions for the knight to move.
     * A knight at (r,c) can move to (r+RowChange[i], c+ColChange[i]),
     * if that destination is on the board and not already used.
     */
    static const int RowChange[8] = { 2, 1, -1, -2, -2, -1, 1, 2};
    static const int ColChange[8] = { 1, 2, 2, 1, -1, -2, -2, -1};

    int direction; /* loop counter for the 8 directions a knight can move */

    if (! OnBoard(AtR,AtC) || when_visited[AtR][AtC]!=NotVisited)
        return 0; /* can't visit this square, so no legal tour */

    /* visit the square */
    when_visited[AtR][AtC] = AtMoveNumber;
    if (AtMoveNumber >= NumSquares)
        return 1; /* partial tour is already complete */

    /* Try adding a move and completing the new partial tour. */
    for (direction=0; direction<8; direction++)
    {
        if (CompleteTour(when_visited,AtMoveNumber+1,
AtR+RowChange[direction], AtC+ColChange[direction]))
            return 1; /* tour completed successfully */
    }

    /* None of the eight directions led to a complete tour with this
     * move added, so backtrack.
     */
    when_visited[AtR][AtC] = NotVisited;
    return 0;
}

```

Figure 7.6: Recursive procedure for knight's-tour.

```

/* main
 * looks for a knight's tour, and prints the tour if it finds one,
 * as described in the abstract at the beginning of the file.
 *
 * The main procedure has no inputs, outputs, or return values.
 * All parameters are set by compile-time constants.
 * The only side effect is the printing of the solution,
 * or a message saying that no tour exists.
 */
int main(void)
{
    int row,col; /* row and column loop indices */
    PartialTour when_visited;

    /* check that the compile-time constants are all legal */
    assert(NumRows>=1);
    assert(NumCols>=1);
    assert(OnBoard(StartRow,StartCol));

    /* clear the when_visited array to get an empty partial tour */
    for (row=0; row<NumRows; row++)
        for(col=0; col<NumCols; col++)
            when_visited[row][col]=NotVisited;

    if (CompleteTour(when_visited, 1, StartRow,StartCol))
        { /* print the solution found */
for (row=0; row<NumRows; row++)
    { for(col=0; col<NumCols; col++)
printf("+---");
printf("+\n");
for(col=0; col<NumCols; col++)
    printf("|%3d",when_visited[row][col]);
printf("| \n");
}
for(col=0; col<NumCols; col++)
    printf("+---");
printf("+\n");
return 0;
    }

    printf("No knight's tour for %d x %d board, starting at (%d,%d)\n",
        NumRows, NumCols, StartRow, StartCol);
    printf("Possible positions range from (0,0) to (%d,%d)\n",
        NumRows-1, NumCols-1);
    return 1;
}

```

Figure 7.7: Main procedure for knight's-tour.

7.6 Format for in-program documentation

Everybody has her or his own ideas about the best places to put documentation in source code, and the best ways to format it. The “correct” style depends on the language and on the practices of the company that will maintain the program. We’ll try to concentrate on the universal features of source code documentation.

Much program documentation is contained in comments. In most documentation styles, two different types of comments are used: block comments and one-line comments. Block comments are multi-line chunks of text, often containing several paragraphs. One good style for block comments in C is the following:

```
/*
 *   Block comments contain ordinary text written in good English.
 *   Several lines may be needed, and blank lines are left
 *   between paragraphs.
 *
 *   If variables or expressions are needed, they should be typed
 *   exactly as they would appear in the code. For example,
 *   alpha[char] is 1 for each alphabetic character, 0 for others.
 *
 *   For ease of later editing, each sentence can be started on a new line,
 *   and no "enclosing box" is used.
 *   The vertical line of stars makes the block comment easy to find.
 *   If you feel obligated to enclose the block comments, you can
 *   put a horizontal line of stars as the first and last line,
 *   but don't put a right-hand column of stars, as it discourages
 *   programmers from changing the comment when they change the code.
 *
 *   A similar style can be used in C programs (with /* and */ delimiters).
 *   Ada programs and assembly language programs use a delimiter
 *   ("--" for Ada) at the beginning of each line of a block comment.
 */
```

This style for block comments has several advantages:

- blocks are clearly marked, and so are easy to find,
- sentences and paragraphs are easy to modify as corrections are made to the program,
- comments are not crowded and sloppy,
- not much time and memory are wasted on “noise” that conveys no information.

One-line comments are used to mark interesting places in the program, and often are simple declarative or imperative sentences (for example, “x[i] is now the largest element.” or “Swap leftmost and rightmost incorrect elements.”).

The following sections explain when to use block comments and when to use one-line comments.

7.6.1 Identify your work

Put your name and corporate address in a block comment at the beginning of every file. When you modify someone else’s code, do not remove the original names, but add a comment saying that you modified the code, what you did to it, and when. In this way you retain the modification history of the program with the source code. These block comments can be extremely useful when tracking down a new bug to find out what changes have been most recently made to the program.

7.6.2 Use white space freely

Ideally, each procedure and its accompanying documentation should fit on a single page. Use page breaks and blank lines to make the printed appearance of the source code correspond to the logical structure of the program. The emphasis on appearance is not to satisfy some perverse artistic aesthetic, but to make the program more readable.

Within procedures, adding a blank line between sections of code acts like a paragraph break to indicate a change in emphasis. Starting a new page before each procedure acts like a section or chapter break, to indicate a whole new topic. The page-break character (often called a form-feed) is control-L—put it before a procedure’s block comment to make the whole thing start on a new page.

Because white space is a powerful tool for grouping text together visually, try to use it in semantically meaningful ways. Don’t break code up arbitrarily, just because it has gotten too long—do add white space between sections performing different tasks. If you feel that a section has gotten too long, then look for a way to break into explainable subparts, don’t just add a blank line in the middle.

7.6.3 Indent to show block structure

Block-structured languages are commonly printed with varying indentation. The more deeply nested a statement is, the more deeply it is indented.

Some programmers use tabs for indenting, but the resulting programs are often too wide to fit on a screen or piece of paper. Other programmers use only one or two spaces for indenting, making the level of nesting difficult to see. Each level of indentation should be about four spaces deeper than the previous, and indentation levels should be consistent. (If you use vi, try “:set shiftwidth=4 autoindent”.)

Slightly different styles of indenting are used by different programmers, which can get confusing when reading a program that has been written by a larger team or modified by several different programmers. It is good practice for all members of a programming team to use the same indenting style and for all subsequent modifications to use the same style. Large development projects often have a style guide containing the conventions to be used by the programmers.

One popular style calls for the bodies of loops and then- or else-clauses to be indented 4 spaces deeper than the loop- or if-statement, with the punctuation (`begin` and `end` or `{` and `}`) at the same level as the outer text:

```
if (test)
  {  action;
}
```

A sequence of tests that determine which of several actions to perform should have all the tests indented to the same level, and all the actions one level deeper:

```
if (test1)
{  action1;
}
else if (test2)
{  action2;
}
else
{  default_action;
}
```

This style is particularly nice in that it lines up matching open and close braces, making it easy to find missing-brace errors.

7.6.4 Name variables carefully

A variable name should tell the reader (not just the programmer) what its function is; calling a variable `rownum` is much better than calling it `i`. Try to avoid using the same variable for multiple purposes.

7.6.5 Use a block comment for each procedure interface

A procedure’s block comment should describe the external view of the procedure. As a minimum, the comment should tell which parameters are input, which are output, what global data structures are used, and what side-effects the procedure may have. The function of the procedure should be clearly explained. Any preconditions that must hold before the procedure can be safely executed should also be included. Choose a standard format for all procedures, perhaps something like

```

/*      procedure-name
 *
 *      action:  explain what the procedure does (not how it does it).
 *      inputs:  list each variable that is input and what it means.
 *      outputs: list each parameter that is an output and what it
 *              means.
 *      returns: what is the meaning of the value returned by a
 *              function?
 *      globals: list any global variables or data structures needed by
 *              procedure.
 *      preconditions: what must be true before each call
 *      postconditions: what is guaranteed to be true when the
 *                      procedure returns
 *      side-effects: list any changes expected as a result of running the
 *                      procedure.
 */

```

The same variable may occur in globals, inputs, and outputs. A global constant may be listed as a global, but not as an input. A global variable that is used, but not changed, may be listed as both a global and as an input. A global variable that is set should be listed as an output or as a side-effect. For example, if a procedure reads data from a file and puts it into a global array, that array is properly viewed as an output of the procedure. On the other hand, if a procedure finds a path through a maze and keeps some statistics about the search as it goes, the changes to the global statistics are a side-effect.

A side-effect is any action performed by the procedure that is not completely contained in the values of the output variables. Common side-effects are changing other global variables, reading input (thus advancing the position in the input stream), and printing output. You often have to mention a global variable twice—once in the section listing global variables, to indicate that the global variable is used in some way, and once in the side-effects section, to explain how the global variable is changed by the procedure.

In many cases, it is better to say something like “**side-effects: none**” than to leave the reader trying to decide whether a procedure has no side-effects or you simply forgot to document the side-effects. But be careful—programmers have written block comments that claimed a procedure had no inputs, outputs, global variables, or side-effects—in other words that the procedure did nothing at all! In most cases, the procedure printed a message or read something from an input—both of which are classified as side-effects.

7.6.6 Use a block comment inside each procedure to explain method

Unless a procedure uses a completely trivial technique, a sentence or two of explanation at the beginning will make it much easier for the reader to understand. You do not have to cram 30 pages of analysis from a text book into a couple of sentences. If you can’t say all that needs to be said in a couple of paragraphs, you should at least mention what technique is being used and give a complete citation to a more complete external description.

The interface description comment is not the correct place for an explanation of the techniques used. You should be able to change techniques (say, from insertion sort to heap sort) without changing the interface description. Use a separate comment for methods.

The methods comment explains what is done, why it is done, and when it is done. The details of how it is done should be left for the code. If you are not sure how much to put in the methods comment, it is probably better to put in too much than to put in too little.

7.6.7 Use a block comment for each data type

When you define a record or array structure, explain the purpose of each field. Give an overall view of the data structure (is it a binary tree? a linked list? a heap?). What special properties of the data structure are you relying on?

7.6.8 Use a block comment for each data structure

You will often use apparently simple data structures (such as arrays) without having to declare a special type. Often, the simpler the data structure, the more ways it could be being used. Elementary data structures may need more commentary than the structured types that have special declarations. The more generic the data structure, the more explanation is needed. If you are using an array, it is totally useless to tell the reader that it is an array—any fool can see that from the declaration. A simple array may have many different possible meanings. It may store a mapping from one set to another—what are the sets? what does the mapping mean? Or the array may be being used as a hash table—what are the keys? what method is used to resolve collisions?

Identify global variables that are used as constants throughout the program.

7.6.9 Use a one-line comment for each local variable

Each local variable should have a single, distinct purpose. The purpose is hinted at by the name of the variable, and explicitly stated in a comment where the variable is declared. For example,

```
int L; /* A[L] is the leftmost unchecked value. */
```

The one-line comment may be just a noun phrase, with an implicit “The variable foo is” in front of it. For example,

```
int L; /*the index of the leftmost unchecked value*/
```

Watch out for slightly inaccurate statements. If you’re in a hurry you might write “first unchecked value” when you mean “the index of the leftmost unchecked value”. Saying “SP is the stack pointer” is much less valuable than saying

```
int SP; /* STK[SP] is the top element of the stack, and
 * STK[SP-1] is the next available empty slot
 */
```

If you are cramped for space in the comment, you can sometimes leave out the definite articles (for example, /*index of leftmost unchecked value*/). Don’t do it if any confusion could result; make the comment multi-line instead.

Be sure your names are not misleading. Using *x* for a variable that indexes the vertical direction in a printout, and *y* for the horizontal variable is a common mistake—people are conditioned to expect *x* to be the horizontal direction. Be particularly careful in checking input and output loops.

7.6.10 Use comments sparingly inside the body of the code itself

Many beginning programmers, when told to document, attempt to be explicit and write useless comments like

```
i = i + 1;    /*increment i*/
```

Many short comments interfere with reading the code itself. A long block comment at the beginning of the procedure can be far more helpful and is usually easier to write. Reserve one-line comments for such tasks as

- identifying cases in a long if-then-elseif-else statement
- headers at the beginning of a new section of code (after a blank line)
- identifying end statements
- displaying loop invariants
- translating complex expressions back into meaningful pseudo-code.

7.6.11 Use assertions.

Many languages support (or, by using macro packages, can be made to support) `assert` statements. The format varies with the implementation, but usually consists of an expression that is supposed to evaluate to true and an error message to print if the assertion is not true. When an assertion fails, it usually breaks to a debugging program (or causes the program to abort). For example, at the beginning of a procedure, one might test a precondition as follows:

```
assert(L<R, "left and right pointers are crossed");
```

Loop invariants are particularly good things to put in assertions. David Gries recommends using the loop invariants as the foundation for designing new algorithms and accurately implementing existing algorithms [Gri83].

If your language does not support `assert` statements, it is still worthwhile to put the assertions in as comments, since they tell the reader something about what you were expecting when you wrote the code.

7.7 Things to keep in mind for peer editing

Read your partner's code as if you had never seen the algorithm or data structures before. Consider what you would need to know to make a major modification. For example, consider writing a graphical procedure to draw the sequence as chords on a circle or changing the simple backtracking search to an "intelligent" heuristic search. What do you need to know about the data structures? about the algorithm?

Look at the spacing and indentation on the page. Is the code visually divided into natural chunks? Compare with the guidelines in Sections 7.6.2 and 7.6.3. Can you see at a glance where each loop begins and ends?

Check that all block comments are in good English. Telegraphic style, in which you leave out articles and verbs, may be used only in one-line comments.

Check that the variable names are meaningful. For example,

- The name `width` may be used for a size, but not for a horizontal position.
- The word `solution` is a poor name for a Boolean variable, unless the problem is to answer a yes/no question. One better name would be `SolutionFound`, but this is still a bit too generic. A name specific to the problem, such as `SequenceCompleted`, is better.
- The name `c` is a poor one for the recursive search procedure, as it does not say what is being done. It might be better for the procedure to be a function called something like `CompleteSequence`. After all, the purpose of it is to complete a partially completed sequence, or to report that no such sequence exists.

Check that the global data structures are well-commented. It is important to document the meaning of the contents, not just of the indices. What does the number in the array `s` mean?

Is the output format for the program described? To say that a program "prints the solution" is not particularly helpful. There are many ways to print the solution—the description of the format should be explicit enough that another programmer can write an interface to the program without having to learn all the internal data structures and operations.

Is the input format correctly described? Even if there are no run-time inputs to the program, there may be compile-time constants that a user might want to change. What are they and what do they mean?

Is the explanation of the recursive procedure clear? The best way to describe a recursive procedure is with preconditions and postconditions. How much of the sequence is already completed before each call to the procedure? What do the different possible return values mean? If global variables are changed, what do they contain before and after the procedure? Re-read Section 11.5 about how to explain recursion.

Termination criteria are also useful—when does the recursion stop? Does the programmer clearly distinguish between the action of one iteration of the recursion (lengthening the sequence by one) and the overall effect of the procedure (determining whether there is a sequence that starts with the current partial sequence)?

Are the side-effects properly described? The main purpose of the recursive procedure is to change the global `s` array—the returned Boolean value is just an output to flag when the procedure is finished. The documentation should make it clear what the contents of the array are after the procedure returns. Warning: the values are different depending on whether a sequence was found or not.

Are the words carefully chosen? Don't slavishly copy words used in the external documentation, as they may not be the best ones to describe what is happening in the program. You should write using your own words, being particularly careful with any words that you have only recently acquired.

Watch out for vague words like *valid* or *good*. When they occur, look for a more descriptive word.

When technical words are used, make sure they are used consistently. Try to have only one meaning for each word, and only one word for each concept. As an example of what can go wrong, one programmer on the knight's tour used *square*, *field*, *record*, *record field*, *box*, *location*, *playing field location*, and *board square* interchangeably for two concepts that were not distinguished, the coordinates of an element of the board array and the contents of that element.

Watch out for **off-by-one** errors in both the programs and the comments!

7.8 The final draft

Typos in programs are not just minor inconveniences; they can be fatal errors. Make sure your final program compiles and runs before handing it in. It is easy to insert a comment that isn't properly terminated, and so even if all you edit are the comments, you have to recompile and re-run to make sure you didn't break anything.

Run your final program—it does no good to document the program if you've broken it!

Try to keep each procedure and its documentation on *one page*. If you must print a procedure on two pages, make the page break in a *natural division* in the code. Insert page breaks and white space to make the appearance of the printed source code correspond to the logical structure of the program. The unix filter `pr` will automatically add the file name, date, and time to the top of each page. If you're using UNIX, look into using `pr` (or the equivalent `lpr -p` option). Some of the machines on campus have `enscript`, which does an even better job of printing program listings.

Do not double space what you hand in. There should be enough white space in a well-formatted program for us to write whatever comments we need. If you end up with dense blocks of code or text, something needs to be changed.

In past years, this assignment has proved to be the most failed assignment—with many students not getting the idea of external descriptions of recursive procedures. Because of this difficulty, we have provided an example of the sort of code we are looking for and requested two drafts with peer-editing on each draft before the final draft is due.