

# Literate Programming

K. Harwood

June 3, 2011

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>How to work with Literate Programming</b>	<b>2</b>
<b>3</b>	<b>Developing using Literate Programming</b>	<b>4</b>
3.1	Example . . . . .	4
3.2	Generated material . . . . .	14
3.3	Results . . . . .	15
3.4	Alternatives . . . . .	15
<b>A</b>	<b>Project indexes</b>	<b>16</b>
A.1	Index of files . . . . .	16
A.2	Index of fragments . . . . .	16
A.3	Index of identifiers . . . . .	17
<b>B</b>	<b>The generated code</b>	<b>17</b>

## Abstract

Write your detailed design documentation. Get your source code for free. Source code is well commented and cross-referenced back to the documentation. Here is an example of how to do it.

## 1 Introduction

Literate programming is a way of documenting the detailed design of software which saves time and effort and also ensures that the documentation is always up to date with the source code.

It also encourages a narrative style of documentation that allows you to describe, not only what the code does and how it works, but also why you designed it that way, what alternatives you considered and why you rejected them.

Literate programming was introduced by Donald Knuth in 1981 and used initially for his  $\text{\TeX}$  typesetting system. It has spread into all forms of software over the years and there are web sites, newgroups and a web domain ([literateprogramming.org](http://literateprogramming.org)) devoted to it. There is a rather thin Wikipedia article on it and Google comes up with 76,900 hits. However, it has not had the impact that other, more glamorous development methods have; mainly, I feel, because it does not have the “silver bullet” quality that promises the world but requires new ways of working<sup>1</sup>.

If you create the source code for your project using a text editor you can use literate programming without materially changing the way you work. The only difference is that where now you create the source code and documentation separately, with literate programming you create the documentation and you get the source code for free. The source code is generated from the documentation.

The value of this is that, firstly, you save a little time in generating the source code. Not much, because everything that is needed for the source code is in the documentation and you’ve had to write it anyway, but at least you haven’t had to write it twice.

A second, and more important, value is that the code and the documentation always agree. You make every change in the documentation; you never edit the source code. The source code is generated from scratch after every change (it’s a very quick process, so there is no motive for doing otherwise) so they can never get out of step.

A third value is that, with the right tool, you can have cross-references within the source code back to the documentation. In other words, the source code has comments which say, in effect, for more detail on this bit of code see the documentation page so-and-so. And, since the documentation and code always agree, these cross-references are always up to date.

## 2 How to work with Literate Programming

I am fluent in  $\text{\LaTeX}$  so I write the text using a normal editor; in my case *vi*. Emacs users might like to create a special mode for literate programming.

---

<sup>1</sup>And also requires you to pay us a fortune for training and another fortune per seat for the tools. I’m serious. The fact that a technique doesn’t cost an arm and a leg often rules it out of consideration.

If you prefer a WYSIWYG editor you can try LyX.  $\text{\TeX}$  and  $\text{\LaTeX}$  are the preferred method in literate programming mainly because they are mark-up languages and generating them is easy.

The literate programming tool I use is `nuweb`, after using `noweb` for quite a long time. I prefer it because it generates all output files, including the documentation, in one execution. The downside is that `nuweb` is not capable of pretty-printing the code whereas with `noweb` you can potentially include filters to do that.

I have hacked `nuweb` in a number of ways, as have a number of others. The latest version is available on SourceForge at <https://sourceforge.net/projects/nuweb>.

- Provide sections which keep fragments and identifiers separate, with provision for exporting and importing them from one section to another. This turned out not to be as useful as I expected and I don't use it much.
- Parameters to fragments. There was a crude facility to do this, but very limited. My parameters can appear in the grammatically correct places in the fragment names and can be passed on from one fragment to another<sup>2</sup>.
- Fragment names as comments in the generated code, with cross-references back to the documentation if desired.
- Fragment names as explicit parts of the generated code. This is useful, for example, in logging or debugging a program as it allows the output to refer directly to the documentation.
- Allow documentation text to appear in block comments in the code.

So, my development cycle is: edit, `nuweb`, compile and/or  $\text{\LaTeX}$ . I use `pdflatex` because it allows the use of hyperlinks to navigate the cross-references; identifiers and fragments have links for definitions and usage. On-line readers can just click to follow the links, readers of the paper version have to turn the pages themselves. Sorry.

---

<sup>2</sup>Parameters in grammatically correct places maintains the narrative style within the code fragments. Anything that helps the reader is good. I deplore the “the readers can work it for themselves” attitude. There is one writer and, say, hundreds of readers. That attitude shows a great contempt on the part of the writer, suggesting that he thinks his time is hundreds of times more valuable than that of his readers. If you catch me doing that, point it out to me straight away.

## 3 Developing using Literate Programming

The following was supposed to be an “advanced” test example for a job I had applied for. I will use it for a very small example of Literate Programming.

Write an Anagram Solver that generates all permutations for a given word (or phrase) and compares these permutations to a dictionary. Assume you have access to a file containing a list of dictionary words. Assume design and readability are more important than execution speed.

Originally the test was for C++, but since none of its interesting features need be used I shall use the more appropriate language, C.

### 3.1 Example

First we have to describe the source file we are going to create.

```
"anagram.c" 4≡
  < File header 5 >
  < Files included by anagram.c 10a, ... >
  < Macros defined for main 9c, ... >
  < Local routines for main 6b, ... >
  int
  main(int argc, char * argv[])
  {
    < Variables used in main 9b, ... >
    < Check correct usage 9a, ... >
    < Open the dictionary file 14a >
    < Print all anagrams 6a >
    return EXIT_SUCCESS;
  }
  ◇
```

Defines: `main` Never used.

Uses: `EXIT_SUCCESS` 10b.

There you will see a little bit of C in typewriter font and lots of bits in angle brackets. However, you can see that if we replace the bits in angle brackets with code that does what they say, we have a complete program. The next part of the job is to provide code fragments for all those bits.

You see that this is classic top-down programming. The big difference here is that you can write down the top level straight away. You don't have to develop several levels down before you can start writing, as you have to do if you are writing directly in the programming language<sup>3</sup>.

Normally I am creating code for clients who like to have a standard heading on each file so I define a `<File header 5>` fragment and put it at the top of every file. Here I have my own.

```
<File header 5> ≡
  /*
   * file name
   *
   * Copyright K. Harwood, 2008.
   */
  ◇
```

Fragment referenced in [4](#).

Most developments require many files (this one only creates one file because it is such a small job) and the same header is used in every file. That *file name* in the fragment is replaced by the name of each file as it is generated.

Let's start with the interesting bit, the bit that does the actual work.

"Generates all permutations . . ." You've got to be kidding. I will generate one permutation of the given word and the same permutation of each dictionary word and check if they match.

(Yes, it is a very bad page break here. Normally fragments aren't broken across pages, it makes them difficult to read, so bad breaks are inevitable. You can, however, chose to have particular fragments break if necessary.)

---

<sup>3</sup>For example, before you can write down "Loop over all members of such-and-such set" directly in a programming language you have to decide how you are going to access the members of the set and what form the loop itself should take. With literate programming you can defer these decisions until late in the piece, when you have collected all the different operations you need to perform on the set and then implement them all at once with pretty near complete knowledge.

Sort the letters of the given word into alphabetical order. Then go through the dictionary and sort each word the same way. If the two sorted words are the same they are anagrams of each other.

```
< Print all anagrams 6a > ≡
    (Comment)
    < Sort the given word into order 8b >
    < For all words in the dictionary 11b >
    {
        < If the dictionary word is the same length 12b >
        {
            < Sort the dictionary word into order 13a >
            < If the sorted words are the same 13b >
            {
                < Print the dictionary word as an anagram 13c >
            }
        }
    }
    }
    ◇
```

Fragment referenced in 4.

What's that *(Comment)*? When the C file is created the paragraph starting “Sort the letters...” will be inserted here as a block comment. This allows us to have properly typeset material in the documentation and a raw version of it in the generated source code.

The key to this program is sorting the words into order. We do it in two places, so the obvious thing to do is make it a subroutine. (`qsort` is pretty much overkill for this job.)

```
< Local routines for main 6b > ≡
    static void
    asort(char * word, int len)
    {
        < Sort word of size len for < Alphabetic 7b > 7a >
    }
    ◇
```

Fragment defined by 6b, 8a.

Fragment referenced in 4.

Defines: `asort` 8b, 13a.

I have made this more elaborate than it needs to be for the purposes of illustration. It is, of course, insertion sort. It's the most efficient sort for up to twenty-ish items and most quicksorts use it for small arrays anyway. Since there aren't terribly many words much longer than twenty letters we might as well avoid the overhead of `qsort`.

```
< Sort key of size n for ordering 7a > ≡
    for (int j = 1; j < n; j++)
    {
        int i = j - 1;
        int kj = key[j];

        do
        {
            int ki = key[i];

            if (ordering)
                break;
            key[i + 1] = ki;
            i -= 1;
        } while (i >= 0);
        key[i + 1] = kj;
    }
    ◇
```

Fragment referenced in [6b](#).

The words *key*, *n* and *ordering* all appear in a funny font. What's more, the corresponding places where this fragment is used have different words. What's more, the *ordering* appears to be replaced by a fragment use. What's going on here?

Code fragments can have parameters. When the fragment is expanded into real code, the stuff in the parameter positions in the use of the fragment replaces those parameters, even if the replacement is another fragment.

In this case the use of a fragment for the ordering is unnecessarily complicated. Straight text would serve just as well. However, here goes.

```
< Alphanumeric 7b > ≡
    ki < kj◇
```

Fragment referenced in [6b](#).

A more elaborate ordering would handle upper and lower case letters, but I am going to solve that problem another way. We have to copy the words to somewhere to sort them, so I will convert them to lower case on the way.

*< Local routines for main 8a >* ≡

```
static void
cpyToLower(char * d, char * s)
{
    char c;

    do
    {
        c = *s++;
        *d++ = tolower(c);
    }while (c != '\0');
}
◇
```

Fragment defined by [6b](#), [8a](#).

Fragment referenced in [4](#).

Defines: `cpyToLower` [8b](#), [13a](#).

Uses: `tolower` [10a](#).

This makes preparing the word easy.

*< Sort the given word into order 8b >* ≡

```
cpyToLower(word, argv[1]);
asort(word, len);
◇
```

Fragment referenced in [6a](#).

Uses: `asort` [6b](#), `cpyToLower` [8a](#).

We had better make sure there is an `argv[1]` to copy.

*⟨ Check correct usage 9a ⟩* ≡

```
    if (argc != 2)
    {
        fprintf(stderr, "Usage: anagram word\n");
        return EXIT_FAILURE;
    }
    ◇
```

Fragment defined by [9a](#), [11a](#).

Fragment referenced in [4](#).

Uses: `EXIT_FAILURE` [10b](#), `fprintf` [14c](#), `stderr` [14c](#).

The *⟨ Sort the given word into order 8b ⟩* fragment uses some variables which haven't appeared so far. Here comes a disadvantage of not using C++. In C++ I could define these variables where they first get used, the sensible place to do it. However, we aren't allowed to do that in C, so I have to introduce the following fragment.

*⟨ Variables used in main 9b ⟩* ≡

```
    char word[MAX_WORD_LEN + 1]; /* The given word */
    int len; /* The length of the given word */
    ◇
```

Fragment defined by [9b](#), [12a](#).

Fragment referenced in [4](#).

Uses: `MAX_WORD_LEN` [9c](#).

But there's another thing that's just turned up. Fix that.

*⟨ Macros defined for main 9c ⟩* ≡

```
    #define MAX_WORD_LEN 100
    ◇
```

Fragment defined by [9c](#), [14b](#).

Fragment referenced in [4](#).

Defines: `MAX_WORD_LEN` [9b](#), [11ab](#), [12a](#), [13a](#).

You may have noticed that every now and then the fragments are noted as defining or using particular identifiers. I choose which identifiers I want to be defined. The tools search through the other fragments to find uses of those identifiers.

But why aren't `word` and `len` marked as defined and used? It's because I choose not to. My policy on marking identifiers is that they should be in a global context. That way they aren't going to be duplicated inside every routine. The things I mark are routine names, `typedef` names, macros; in general anything that appears in those fragments that are at the top level in a program. (On the other hand, it is sometimes convenient to have other values cross-referenced this way, especially when they have the same name everywhere they appear. It's a policy, not a rule.)

I also mark the fields of structures and classes. These could be duplicated, but I choose to give them unique names, names beginning with an abbreviation of the structure or class's name. Yes, the compiler can keep them apart, but it's sometimes a little difficult for the human reader. And having them cross-referenced where the reader can go straight to where they are described helps as well.

But what about that `tolower`? We can't point to its definition because that is long, long time ago in a galaxy far, far away.

Nah! Easi-peasy.

*< Files included by anagram.c 10a >* ≡

```
#include <ctype.h>
```

◇

Fragment defined by [10ab](#), [12c](#), [14c](#).

Fragment referenced in [4](#).

Defines: `tolower` [8a](#).

While we are thinking of it...

*< Files included by anagram.c 10b >* ≡

```
#include <stdlib.h>
```

◇

Fragment defined by [10ab](#), [12c](#), [14c](#).

Fragment referenced in [4](#).

Defines: `EXIT_FAILURE` [9a](#), [11a](#), [13c](#), [14a](#), `EXIT_SUCCESS` [4](#).

Handling `#include` this way tells the reader not only what files are included, but also why.

We had better make sure that the given word will fit in that buffer.

*< Check correct usage 11a >* ≡

```
len = strlen(argv[1]);
if (len > MAX_WORD_LEN)
{
    fprintf(stderr, "The given word is too long\n");
    return EXIT_FAILURE;
}
◇
```

Fragment defined by [9a](#), [11a](#).

Fragment referenced in [4](#).

Uses: `EXIT_FAILURE` [10b](#), `fprintf` [14c](#), `MAX_WORD_LEN` [9c](#), `stderr` [14c](#), `strlen` [12c](#).

But, hang on there Roy, that's the second time I've given a definition for *< Check correct usage 9a, ... >*. Yup, no problem. Every new definition just adds to the ones that are already there. You will see we do this for several fragments, and very convenient it is too. The ellipsis in the cross-reference gives you the hint that a fragment is defined in several places and every definition of the fragment gives a list of all the places it is defined.

The statement of the problem didn't give any indication as to how the dictionary is stored. I will assume one word per line.

*< For all words in the dictionary 11b >* ≡

```
while (fgets(dictWord, MAX_WORD_LEN + 1, dict) != NULL)◇
```

Fragment referenced in [6a](#).

Uses: `fgets` [14c](#), `MAX_WORD_LEN` [9c](#).

Note that that fragment is the only place where the structure of the dictionary matters. For many other structures, this is the only fragment that would need to be rewritten. I give an example below on page [15](#).

We need dictionary words to be longer than the given words because `fgets` will add a newline character which we have to remove.

*< Variables used in main 12a >* ≡

```
char dictWord[MAX_WORD_LEN + 2];
FILE * dict;
◇
```

Fragment defined by 9b, 12a.

Fragment referenced in 4.

Uses: FILE 14c, MAX\_WORD\_LEN 9c.

We have to remove that extra newline before we can do anything with the word.

*< If the dictionary word is the same length 12b >* ≡

```
char * p = strchr(dictWord, '\n');
size_t dlen;

if (p != NULL)
    *p = '\0';
dlen = strlen(dictWord);
if (dlen == len)◇
```

Fragment referenced in 6a.

Uses: strchr 12c, strlen 12c.

Note that although the name of this fragment makes it look like a simple comparison of lengths, the fragment includes quite a bit of code necessary to calculate one of those lengths. The point is that it is not necessary to consider this where the fragment is used, it is irrelevant to the correctness of the algorithm. Here, where we are implementing it however, it is relevant.

*< Files included by anagram.c 12c >* ≡

```
#include <string.h>
◇
```

Fragment defined by 10ab, 12c, 14c.

Fragment referenced in 4.

Defines: strchr 12b, strcmp 13b, strlen 11a, 12b.

Sorting the dictionary word is pretty much the same as for the given word. The main difference is that the place we put it is local to the fragment.

*Sort the dictionary word into order 13a* ≡

```
char dw[MAX_WORD_LEN + 1];

cpyToLower(dw, dictWord);
asort(dw, dlen);
◇
```

Fragment referenced in [6a](#).

Uses: [asort 6b](#), [cpyToLower 8a](#), [MAX\\_WORD\\_LEN 9c](#).

Because we have already tested that the lengths of the given and dictionary words are the same the test is just that they contain the same characters.

*If the sorted words are the same 13b* ≡

```
if (strcmp(word, dw) == 0)◇
```

Fragment referenced in [6a](#).

Uses: [strcmp 12c](#).

Printing the anagram words is simple. They are going to come out one per line.

*Print the dictionary word as an anagram 13c* ≡

```
if (puts(dictWord) == EOF)
{
    /* Something is seriously wrong. */
    return EXIT_FAILURE;
}◇
```

Fragment referenced in [6a](#).

Uses: [EXIT\\_FAILURE 10b](#), [puts 14c](#).

Not much left to go now.

*⟨ Open the dictionary file 14a ⟩ ≡*

```
dict = fopen(DICTIONARY, "r");
if (dict == NULL)
{
    perror("Can't open dictionary");
    return EXIT_FAILURE;
}
◇
```

Fragment referenced in [4](#).

Uses: [DICTIONARY 14b](#), [EXIT\\_FAILURE 10b](#), [fopen 14c](#), [perror 14c](#).

*⟨ Macros defined for main 14b ⟩ ≡*

```
#define DICTIONARY "dict.txt"
◇
```

Fragment defined by [9c](#), [14b](#).

Fragment referenced in [4](#).

Defines: [DICTIONARY 14a](#).

*⟨ Files included by anagram.c 14c ⟩ ≡*

```
#include <stdio.h>
◇
```

Fragment defined by [10ab](#), [12c](#), [14c](#).

Fragment referenced in [4](#).

Defines: [fgets 11b](#), [FILE 12a](#), [fopen 14a](#), [fprintf 9a](#), [11a](#), [perror 14a](#), [puts 13c](#),  
[stderr 9a](#), [11a](#).

## 3.2 Generated material

This document itself is the principle product of the development.

In [Appendix A on page 16](#) are three indexes. One is the index of files created by this project. There is only one, so it's not a very long index. The second is an index of all the fragments. The third is an index of all the identifiers whose definitions have been marked.

[Appendix B on page 17](#) shows the source code generated for this project.

### 3.3 Results

I would like to boast that the developed code worked first time. Alas, I cannot do so.

The initial version of *Sort the dictionary word into order 13a* calculated the length of the word in `dw` before putting the word in there. Doh! However, I then realised there was no point in sorting the dictionary words if they were the wrong length. I therefore introduced *If the dictionary word is the same length 12b* which calculated the length from the dictionary word itself.

This new version did work first time.

### 3.4 Alternatives

Here are a couple of different ways that I could have developed this code.

One alternative would be to not use the `asort` routine at all, using *Sort key of size n for ordering 7a* with appropriate arguments where the sorting is required. I do this in other projects and it is especially valuable when the ordering condition requires data which is not accessible from the array to be sorted<sup>4</sup>.

I promised you an alternate implementation of *For all words in the dictionary 11b* Here it is.

```
Alternate For all words 15a ≡  
  For all distinct letters in the given word 15b  
  And all dictionary words beginning with that letter 16◇
```

Fragment never referenced.

The idea is that the dictionary is partitioned on the first letter of the words. We therefore need to look up only those words which begin with a letter of the given word.

```
For all distinct letters in the given word 15b ≡  
  for (char * p = word; *p != 0; p++)  
    if (p[0] != p[1])◇
```

Fragment referenced in [15a](#).

---

<sup>4</sup>This is one of the problems with the standard `qsort`. It assumes that all the information necessary to compare two elements is available in the elements. When it isn't, it has to be provided as a global value. This is inelegant in the first place and in the second prevents re-entrant code from using `qsort`.

```

⟨ And all dictionary words beginning with that letter 16 ⟩ ≡
    if (⟨ Open dictionary at letter p[0] ? ⟩)
        while (⟨ Get the next word from dictionary ? ⟩)◇

```

Fragment referenced in 15a.

The assumption here is that we have one routine that starts a dictionary traverser at a particular letter and returns **FALSE** if there are no entries beginning with that letter.

## A Project indexes

### A.1 Index of files

"anagram.c" Defined by 4.

### A.2 Index of fragments

- ⟨ Alphabetic 7b ⟩ Referenced in 6b.
- ⟨ Alternate For all words 15a ⟩ Not referenced.
- ⟨ And all dictionary words beginning with that letter 16 ⟩ Referenced in 15a.
- ⟨ Check correct usage 9a, 11a ⟩ Referenced in 4.
- ⟨ File header 5 ⟩ Referenced in 4.
- ⟨ Files included by anagram.c 10ab, 12c, 14c ⟩ Referenced in 4.
- ⟨ For all distinct letters in the given word 15b ⟩ Referenced in 15a.
- ⟨ For all words in the dictionary 11b ⟩ Referenced in 6a.
- ⟨ Get the next word from dictionary ? ⟩ Referenced in 16.
- ⟨ If the dictionary word is the same length 12b ⟩ Referenced in 6a.
- ⟨ If the sorted words are the same 13b ⟩ Referenced in 6a.
- ⟨ Local routines for main 6b, 8a ⟩ Referenced in 4.
- ⟨ Macros defined for main 9c, 14b ⟩ Referenced in 4.
- ⟨ Open dictionary at letter p[0] ? ⟩ Referenced in 16.
- ⟨ Open the dictionary file 14a ⟩ Referenced in 4.
- ⟨ Print all anagrams 6a ⟩ Referenced in 4.
- ⟨ Print the dictionary word as an anagram 13c ⟩ Referenced in 6a.
- ⟨ Sort key of size n for ordering 7a ⟩ Referenced in 6b.
- ⟨ Sort the dictionary word into order 13a ⟩ Referenced in 6a.
- ⟨ Sort the given word into order 8b ⟩ Referenced in 6a.
- ⟨ Variables used in main 9b, 12a ⟩ Referenced in 4.

## A.3 Index of identifiers

asort: [6b](#), [8b](#), [13a](#).  
cpyToLower: [8a](#), [8b](#), [13a](#).  
DICTIONARY: [14a](#), [14b](#).  
EXIT\_FAILURE: [9a](#), [10b](#), [11a](#), [13c](#), [14a](#).  
EXIT\_SUCCESS: [4](#), [10b](#).  
fgets: [11b](#), [14c](#).  
FILE: [12a](#), [14c](#).  
fopen: [14a](#), [14c](#).  
fprintf: [9a](#), [11a](#), [14c](#).  
main: [4](#).  
MAX\_WORD\_LEN: [9b](#), [9c](#), [11ab](#), [12a](#), [13a](#).  
perror: [14a](#), [14c](#).  
puts: [13c](#), [14c](#).  
stderr: [9a](#), [11a](#), [14c](#).  
strchr: [12b](#), [12c](#).  
strcmp: [12c](#), [13b](#).  
strlen: [11a](#), [12b](#), [12c](#).  
tolower: [8a](#), [10a](#).

## B The generated code

```
/* File header */
/*
 * anagram.c
 *
 * Copyright K. Harwood, 2008.
 */

/* Files included by |anagram.c| */
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

/* Macros defined for |main| */
#define MAX_WORD_LEN 100
#define DICTIONARY "dict.txt"

/* Local routines for |main| */
static void
```

```

asort(char * word, int len)
{
    /* Sort 'word' of size 'len' for <Alphabetic> */
    for (int j = 1; j < len; j++)
    {
        int i = j - 1;
        int kj = word[j];

        do
        {
            int ki = word[i];

            if (ki < kj)
                break;
            word[i + 1] = ki;
            i -= 1;
        } while (i >= 0);
        word[i + 1] = kj;
    }
}

static void
cpyToLower(char * d, char * s)
{
    char c;

    do
    {
        c = *s++;
        *d++ = tolower(c);
    }while (c != '\0');
}

int
main(int argc, char * argv[])
{
    /* Variables used in |main| */
    char word[MAX_WORD_LEN + 1]; /* The given word */
    int len; /* The length of the given word */
    char dictWord[MAX_WORD_LEN + 2];
    FILE * dict;

```

```

/* Check correct usage */
if (argc != 2)
{
    fprintf(stderr, "Usage: anagram word\n");
    return EXIT_FAILURE;
}
len = strlen(argv[1]);
if (len > MAX_WORD_LEN)
{
    fprintf(stderr, "The given word is too long\n");
    return EXIT_FAILURE;
}

/* Open the dictionary file */
dict = fopen(DICTIONARY, "r");
if (dict == NULL)
{
    perror("Can't open dictionary");
    return EXIT_FAILURE;
}

/* Print all anagrams */
/* Sort the letters of the given word into alphabetical order.
 * Then go through the dictionary and sort each word the same
 * way. If the two sorted words are the same they are anagrams
 * of each other.
 */
/* Sort the given word into order */
cpyToLower(word, argv[1]);
asort(word, len);

/* For all words in the dictionary */
while (fgets(dictWord, MAX_WORD_LEN + 1, dict) != NULL)
{
    /* If the dictionary word is the same length */
    char * p = strchr(dictWord, '\n');
    size_t dlen;

    if (p != NULL)
        *p = '\0';
}

```

```

dlen = strlen(dictWord);
if (dlen == len)
{
    /* Sort the dictionary word into order */
    char dw[MAX_WORD_LEN + 1];

    cpyToLower(dw, dictWord);
    asort(dw, dlen);

    /* If the sorted words are the same */
    if (strcmp(word, dw) == 0)
    {
        /* Print the dictionary word as an anagram */
        if (puts(dictWord) == EOF)
        {
            /* Something is seriously wrong. */
            return EXIT_FAILURE;
        }
    }
}
}

return EXIT_SUCCESS;
}

```