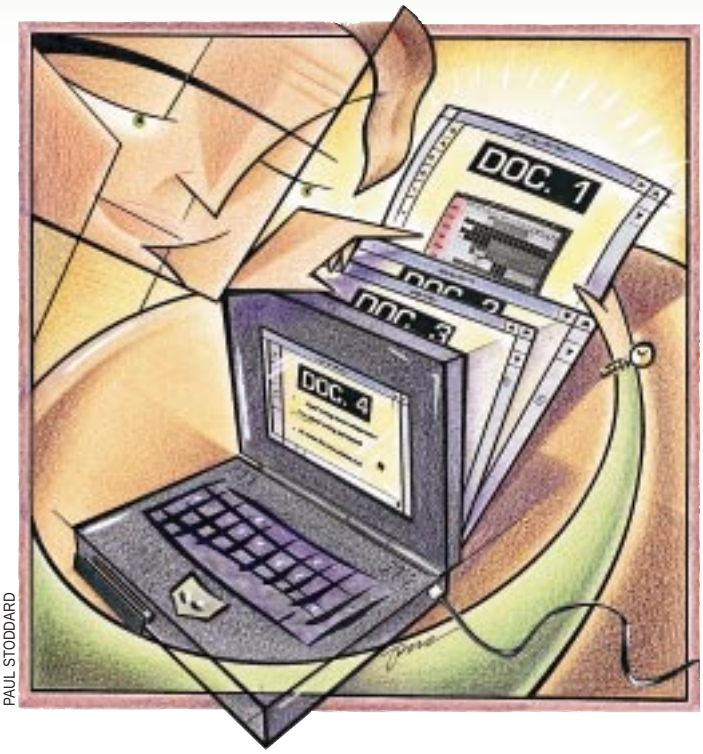
by Jeffreys Copeland and Haemer



PAUL STODDARD

Reader, Part 1

Last month, we talked about reading text online. This month and next, we'll provide some code for you to do so. We wrote the original version of the program we present here during a standards meeting. We brought along a few dozen back issues of the RISKS-Forum Digest (see Usenet newsgroup, `comp.risks`, or `ftp://ftp.sri.com/risks`) to read on our laptop during the boring parts of the meeting, but kept having to interrupt our reading to check data in other files. "Gosh," we thought, "how come `more` doesn't have a way to start up where we left off last time?" (Reading RISKS during a meeting is certainly better for our sense of well-being than reading it on an airplane!)

We wrote a pager that did just that. You started it with a list of files on the command line, and it saved the list of files and the point at which you stopped reading. When you started the program without arguments, it found the last list of files–and your electronic equivalent of a bookmark–and restored itself to where you left off.

We wrote the current version of the program when we found ourselves fixing the paging code for the seventh time to more closely match the behavior of `more`. This version is actually a front-end to a modified version of `less`. As a result, the underlying pager code has a full set of features. We have written the wrapper code in C++ (so we can use some data-hiding capabilities) using the `CWEB` literate programming tool. (If you're not familiar with this tool, `CWEB` allows us to pleasantly mix program and documentation and then extract one or the other as needed. It's well-suited to articles like this one. For more details, see "Cathedrals, Bazaars, and News Readers," July 1998, Page 57, and "Virtual Threaded News Reader," August 1998, Page 54, or take a look at our Web page.)

In addition to those outlined above, we needed some other features:

- The code needed to be portable because we wanted to run it on our desktop SPARCs and our DOS-based laptops. (It turns out this code is sufficiently portable that we could also run it on Windows NT-based laptops loaded with Interix, a soon-to-be-certified port of UNIX to Windows NT. Interix used to be called OpenNT; Windows NT will soon be called Windows 2000.)
- The data files needed to be portable for the same reason–they couldn't depend on byte order.
- We wanted to be able to store a collection of files, such as RISKS, in a single `ZIP` archive and read that as if it were a directory.
- The ability to store "clippings" (that is, save parts of a file for later reference) would be useful.
- We wanted something small enough that we could run it out of RAMDISK on DOS. This means it would use minimum power when we were reading on an airplane. (Peter Neuman reports in RISKS that until airlines made provisions for power onboard, trans-Pacific flights

often found their restrooms occupied by laptops, more so than humans.)

We'll discuss each of these features as we develop the code, and we'll throw in some exercises for the reader along the way. Having said all that by way of setup, we can jump right in.

Let's begin by laying out the basic program:

```
<header files>
<prototypes>
<global data>
<class definitions>
<auxilary routines>
<main program>
```

It will be helpful to have two versions of the main program: "test" and "production." So we can swap in test or production versions as necessary.

```
<main program>=
#ifdef TEST
<test main>
#else
<production main>
#endif
```

We can also guess a couple of the header files we'll need. We're going to do our input using the `stdio` library, rather than the newer C++ `streamio` package. (Exercise for the reader: How would you go about converting the program to use `streamio`?) We'll also need some standard definitions. Similarly, we know a priori that we're going to need to do some string processing–what do we write that doesn't? We've included both a DOS and a UNIX complement of `include` files. Notice that the `strcasecmp` interface exists under various names, so we use a `#define` to work around that:

```
<header files>=
#define _ALL_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __MSDOS__
#include <io.h>
#include <dos.h>
#define strcasecmp stricmp
#else
#include <unistd.h>
#include <strings.h>
#endif
#include <sys/stat.h>
```

We're going to make use of a string routine that is common, `strdup()`, but we have run across C implementations without it. Let's add it just in case it's missing:

```
<prototypes>=
char *strdup(char *s);
```

And the routine itself:

```
<auxilary routines>=
char *
strdup(char *s)
{
   char *x;
   x = (char *) malloc(strlen(s)+1);
   if( x != NULL )
       strcpy(x,s);
   return x;
}
```

Last, we've got some routines that we will define later but for which we need prototypes now:

```
<prototypes>=
void fatal( char *msg );
void warning( char *msg, char *s );
char *next_file_from_listing(void);
```

Bookmark Class

We actually have two types of bookmarks. One is used to mark the point where we last left off on our linear reading of the material, but we also need to simulate fingers holding our place in the book as we flip back and forth chasing references, or places we want to read to our spouses later. For convenience, we'll refer to these as the *main bookmark* and the *supplementary bookmarks*. In the normal course of events, we'll only have a single instantiation of the `Bookmark` class.

```
<class definitions>=
<bookmark class definition>
<bookmark interface code>
```

The class internals are simple at the top level:

```
<bookmark class definition>=
class Bookmark {
public:
<bookmark interfaces>
private:
<bookmark data>
};
```

For the data, we need to include the following. (If we had structured our classes differently, we might have included a *FileList* class containing the list of files and a *BookMark* class containing a single bookmark. We could then instantiate a "BookMark" for the main bookmark, and one for each supplementary bookmark we used. Instead, we've decided to include all the data in a single class.)

```
<bookmark data>=
int file_count;  // how many files
char **file_names;   // the actual file names
ssize_t file_which, file_where; // main bookmark
```

```
   // next some constants
enum { BOOKMARKS=26 }; // the array sizing
enum { EMPTY_BM=-1L }; // empty bookmark defn
   // supplementary bookmarks
ssize_t bm_which[BOOKMARKS];
ssize_t bm_where[BOOKMARKS];
```

Let's add the prototypes for our interfaces to the class definition:

```
<bookmark interfaces>=
Bookmark(FILE *fp);
Bookmark();
~Bookmark();
void Add_File(char *name);
int Find_File(char *name);
void Write(char *name);
bool Query(char c,
        ssize_t &which, ssize_t &where);
bool Set(char c, ssize_t which, ssize_t where);
void Set_main(ssize_t which, ssize_t where);
bool Set_main(char c);
```

In addition, we'll add some inline routines to return data from inside the class:

```
<bookmark interfaces>=
char *
Current_File()
{ return file_names[file_which]; }

ssize_t
Current_FileNr()
{ return file_which; }

ssize_t
Current_Line()
{ return file_where; }

int
File_Count()
{ return file_count; }
```

And we need a handful of interfaces to modify data inside the class. (Exercise for the reader: Can you implement these as C++ overloaded operators?)

```
<bookmark interfaces>=
void
Prev_File()
{
  if( file_which > 0 )  --file_which;
  file_where = 0L;
}

void
Next_File()
{
  if( ++file_which > file_count )
      --file_which;
  file_where = 0L;
}

bool
No_More_Files()
{
  return( file_which == file_count );
}
```

We need to define a global (variable) to tell us if we're (reading) a `ZIP` archive; for convenience, we'll save the file name of the archive. Also, if we're reading a `ZIP`, we unfurl the current file into the current directory. This means we need to access `basename(file)` if it's a `ZIP` we're reading; otherwise, `file`. We define a macro to test this for us:

```
<header files>=
#define we_are_zip ((zip_name != NULL))
#define locate(x) (we_are_zip? basename(x) : x)
```

The Boolean type is not necessarily part of the language–it has been in and out of the C++ specification so many times no one's quite sure. So we do some more defensive programming:

```
<header files>=
#ifndef bool
#define bool short
#define false 0
#define true 1
#endif
```

We've discovered that in older versions of certain DOS compilers, some useful constants may be missing:

```
<header files>=
#ifdef __MSDOS__
# ifndef FILENAME_MAX
#  define FILENAME_MAX BUFSIZ
# endif
# ifndef R_OK
#  define R_OK 04
#  define W_OK 02
# endif
#endif
```

Similarly, we're going to need a file in which to save our list of files and bookmarks. We'll define its name and the name of the printed clippings now, and save ourselves some grief later. We'll also define a single string containing both names. This will save us some effort in function calls later on, as we shall see.

```
<header files>=
#define INDEX "___ndx__"
#define PRINT "___prt"
```

```
#define INDEX_PRINT "___ndx__ ___prt"
```

At the same time, we'll need to define the global instances of the `Bookmark` class and the `zip_name` defined:

```
<global data>=
class Bookmark;
char *zip_name = NULL;
Bookmark *marks;
```

How will the data be stored in the index file? In an effort to make the index file portable, we will write the file in flat ASCII. For example, the following sample index file for a selection of RISKS Digests contains a count of files, the names of the files, the main bookmark (in the form of file number and line number within file) and the supplementary bookmarks, which are lettered for convenience:

```
10
risks17.60
risks17.61
risks17.62
risks17.63
risks17.64
risks17.65
risks17.66
risks17.67
risks17.68
risks17.69
7 90
a 3 24
c 4 93
q 7 114
```

Next, we must provide the code for the methods we prototyped earlier. We begin by providing a method to read an open bookmark file. Let's do this in the constructor, as follows:

```
<bookmark interface code>=
Bookmark::Bookmark(FILE *fp)
{
     char buf[BUFSIZ];
     if( fgets(buf,BUFSIZ,fp) == NULL )
         fatal("bad bookmark file: file_count");
     file_count = atoi(buf);
     // now we allocate the file names array:
     file_names =
      (char **)malloc(file_count*sizeof(char *));
     // read the file names:
     for( int i = 0;  i < file_count;  i++ )
     {
         if( fgets(buf,BUFSIZ,fp) == NULL )
             fatal("bad bookmark file: file_name");
         char *s;
         if( (s=strchr(buf,'\r')) != NULL )  *s = 0;
         if( (s=strchr(buf,'\n')) != NULL )  *s = 0;
         file_names[i] = strdup(buf);
     }
     // main bookmark:
     if( fgets(buf,BUFSIZ,fp)==NULL )
         fatal("bad bookmark file: main bookmark");
     sscanf(buf,"%ld %ld", &file_which,
                &file_where);
     // range check current file number:
     if( file_which >= file_count )
         file_which = file_count - 1;
     /* supplementary bookmarks: */
     // ...begin by clearing them:
     for_all_bookmarks(c) {
         bm_which[MARK(c)] =
         bm_which[MARK(c)] = EMPTY_BM;
     }
     // ...now read the ones in the file:
     while( fgets(buf,BUFSIZ,fp) != NULL )
     {
         char c;
         sscanf(buf,"%c %ld %ld", &c,
              &bm_which[c-'a'], &bm_where[c-'a']);
     }
}
```

We've done a trick with the loop around the supplementary bookmarks that needs some explaining. We're going to be looping through those supplementary bookmarks frequently and we don't want to institutionalize the notion that there are only 26 of them. What we'll do is set up a macro for the loop and a macro to decode a bookmark name (whatever form that may take) into an array index. Note that we specify the loop variable to `for_all_bookmarks` so that we can use the same one with `MARK`. (This is a case where we might have been better with an inline function rather than a macro.)

```
<header files>=
#define for_all_bookmarks(x) \
            for(char x='a';  x<='z'; x++ )
#define MARK(x)   ((x)-'a')
```

We'll also need a method to provide an empty `Bookmark` instance. This gives us a way to generate a `Bookmark` with no files in it. In other words, it lets us bootstrap the `Bookmark` when we don't already have an index:

```
<bookmark interface code>=
Bookmark::Bookmark()
{
    file_count = 0;
    file_names = NULL;
    file_which = file_where = 0L;
    for_all_bookmarks(c){
        bm_which[MARK(c)] =
            bm_which[MARK(c)] = EMPTY_BM;
   }
}
```

We only need one destructor for the class:

```
<bookmark interface code>=
Bookmark::~Bookmark()
{
    for( int i = 0;  i < file_count;  i++ )
        free( file_names[i] );
    free( file_names );
    file_count = 0;
    file_which = file_where = EMPTY_BM;
    for_all_bookmarks(c) {
        bm_which[MARK(c)] =
            bm_where[MARK(c)] = EMPTY_BM;
   }
}
```

When starting up with a fresh file list, we need to be able to add files to the end of the list in the `Bookmark`. This routine will also be useful if we want to add files to an existing index. To do this, we just need to expand the existing array of file names:

```
<bookmark interface code>=
void
Bookmark::Add_File(char *name)
{
    file_count++;
    if( file_names )
        file_names =
           (char **) realloc(file_names,
               file_count*sizeof(char *));
    else
        file_names = (char **) malloc(sizeof(char *));
    if( file_names == NULL )
        fatal("can't add file to bookmarks");
    file_names[file_count-1] = strdup(name);
}
```

Similarly, we're going to need a method to find a file name in the file name list. If we don't have it, we'll return `-1`; otherwise, we'll return the index in the `file_names` array.

```
<bookmark interface code>=
int
Bookmark::Find_File(char *name)
{
    for( int i = 0;  i < file_count;  i++ )
        if( strcmp(file_names[i], name) == 0 )
            return i;
    return -1;
}
```

We will eventually need to write the bookmarks to a file. Normally, we would use the file specified by the `INDEX` macro. Given its name, producing the file itself is easy. (Exercise for the reader: How could we name the index file so it is hidden on DOS and UNIX, and use the same name on both systems?)

```
<bookmark interface code>=
void
Bookmark::Write(char *name)
{
    FILE *fp;
    if( (fp = fopen(name, "w")) == NULL )
       fatal("can't open bookmarks for writing");
    fprintf(fp, "%d\n", file_count);
    for( int i = 0;  i < file_count;  i++ )
       fprintf(fp, "%s\n", file_names[i] );
    fprintf(fp, "%ld %ld\n",
       file_which, file_where );
    for_all_bookmarks(c)
    {
        if( bm_which[MARK(c)] != EMPTY_BM )
            fprintf(fp, "%c %ld %ld\n",
              c, bm_which[MARK(c)], bm_where[MARK(c)]);
    }
    fclose(fp);
}
```

We'll also need to query and set supplementary bookmarks. When we ask for a bookmark, we will handle the file and line number by reference and return `false` if the bookmark is unset. When we want to set a bookmark, we'll return `false` only if the bookmark is out of range.

```
<bookmark interface code>=
bool
Bookmark::Query(char c,
    ssize_t &which, ssize_t &where)
{
    if( bm_which[MARK(c)] == EMPTY_BM )
        return false;
    which = bm_which[MARK(c)];
    where = bm_where[MARK(c)];
    return true;
}

bool
Bookmark::Set(char c,
    ssize_t which, ssize_t where)
{
    if( MARK(c) < 0 || MARK(c) > BOOKMARKS )
        return false;
    bm_which[MARK(c)] = which;
    bm_where[MARK(c)] = where;
    return true;
}
```

We'll also need a method for setting the main bookmark. If we choose a file out of the range of our array of file names, we only change the line number, not the file.

```
<bookmark interface code>=
void
Bookmark::Set_main(ssize_t which, ssize_t where)
```

```
{
    if( which >= 0  &&  which < file_count )
        file_which = which;
    file_where = where;
}
```

Also, we want a version of `Set_main` that sets from a given secondary bookmark and returns `true` if the bookmark is set:

```
<bookmark interface code>=
bool
Bookmark::Set_main(char bkmk)
{
    ssize_t which, where;
    if( !Query(bkmk,which,where) )
    return false;
    Set_main(which, where);
    return true;
}
```

We need to add some code so that we can handle the previous context as we do in `vi`: We can get back to the last place we were reading using a `''` command. But we won't do that now. (Exercise for the reader: How would you add that code?)

Initial Setup, Command-Line Parsing

We need to get in here and define the skeleton of the main program, which will include the following:

- How to parse the files and flags on the command line.
- How to distinguish text files from `ZIP` archives.
- How to set up the initial index file.
- How to instantiate the `Bookmark` class.

```
<production main>=
main(int ac, char *av[])
{
    <parse the command line>
    <instantiate and populate Bookmark>
    <additional index processing, if needed>;
    <process the files>
    return( 0 );
}
```

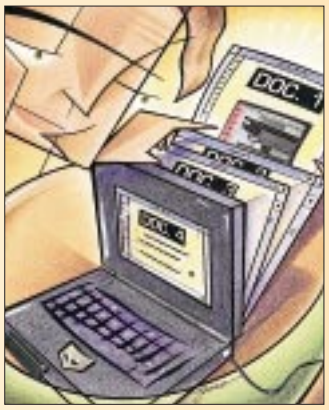
Next, we can parse the command line. There are a few cases to consider:

1. There are no arguments.
2. The first argument on the line is a plus sign (+).
3. The first argument on the command line is a `ZIP` archive.
4. The first argument on the line is a file.

We'll explain the action in each case as we go along, but the fundamental goal is to identify and open the index file, resulting in a `FILE *` for the index.

An earlier version of this program recognized additional flags, in particular, the flags for encryption in the various `ZIP` tools. We're not going to support that here, for two reasons: First, there is a lot of difference between encryption in the Info-ZIP utilities on UNIX and in the `PKZIP` utilities on DOS–the UNIX versions want the passwords from the console not on the command line, for example. Second, such encryption is adequately provided by external programs such as the Info-ZIP `zipcloak` and Phil Zimmerman's Pretty Good Privacy (PGP). When we wrote the original program, we added a note apologizing if the U.S. cryptographic regulations made it difficult for users to get those software packages, but now it's probably easier to get them from outside the United States.

If there are no arguments, then we should already have an index file in our current directory. If we have no arguments and cannot open an index file, we've got a problem.




```
<parse the command line>=
FILE *fp = NULL; // the index file pointer
```

If there no arguments, then we should already have an index file in our current directory. If we have no arguments and cannot open an index file, we've got a problem.

```
<parse the command line>=
if( ac == 1  )
{
    if( access(INDEX,R_OK) != 0 )
        fatal( "no arguments and no index file?" );
}
```

If the first argument is a plus sign (+), we want to add the files given here to the existing index file. We should already have an index file. We'll set a flag to tell us that we're adding and deal with their names later:

```
<parse the command line>=
else
if( strcmp(av[1],"+") == 0 )
{
    if( access(INDEX,R_OK) != 0 )
        fatal( "files to add but no index file?" );
    adding_files++;
}
```

For that last case, we need to declare the variable:

```
<global data>=
int adding_files = 0;
```

Next, we need to determine if our first argument is a file.

Because it may be a ZIP archive, we'll try both the given name and the name with .zip appended. If the first argument is not a file, we're in trouble. The status messages are not strictly necessary but allow us to see what progress is being made.

```
<parse the command line>=
else
{
   char name[FILENAME_MAX];
   char namezip[FILENAME_MAX];
   strcpy(name, av[1]);
   strcpy(namezip, av[1]);
   strcat(namezip, ".zip");
   if( access(name,R_OK) != 0 )
   {
       printf("can't open %s, checking %s\n",
           name, namezip); //???
       if( access(namezip,R_OK) == 0 )
           strcpy(name,namezip);
       else
           fatal( "can't open first file" );
       printf("whew!  got %s\n", name); //???
   }
   <check if we're a zip>
}
```

If we reach this next section of code, we have a readable file. (If not, we invoked `fatal()` in the last section.) Let's check if we're reading a ZIP file: The magic cookie is `PK\003\004`. If so, we'll set `zip_name` and try to extract the index and clippings files:

```
<check if we're a zip>=
char buf[BUFSIZ];
fp = fopen( name, "rb" );
fgets(buf,BUFSIZ,fp);
if( strncmp(buf,"PK\003\004",4) == 0 )
{
    zip_name = strdup(name);
    <clobber an existing index file>
    <unzip the index and print files>;
}
fclose(fp);
```

If we are reading a ZIP (which we must be if we're inside this `if`) we must remove any existing index file before we unpack the new index file. We're set up to overwrite it, but if the current directory has an old index file, and our current zip doesn't have one yet, we could be looking at an unrelated index.

```
<clobber an existing index file>=
if( access(INDEX,R_OK) == 0 ) unlink(INDEX);
```

If we can open an index file, we can instantiate a populated `Bookmark` from it. If not, we can create an empty `Bookmark` object and proceed from there. We still haven't dealt with the issue of adding files if we had a plus sign (+) argument; we'll do that at the end.

```
<instantiate and populate Bookmark>=
if( (fp=fopen(INDEX,"r")) != NULL )
{
     marks = new Bookmark(fp);
     fclose(fp);
} else {
     marks = new Bookmark();
     <populate the bookmarks>
}
<possibly add files>
```

If we don't already have an index file, we need to populate one. We only need to worry about this in the non-ZIP case. In the ZIP case, we populate the file names into the `Bookmark` when we synchronize the listing with the list in the `Bookmark`, as follows:

```
<populate the bookmarks>=
if( !we_are_zip )
{
    <populate text bookmarks>
}
```

Populating a non-ZIP bookmark is pretty easy, we just add all the file names to the list on the command line:

```
<populate text bookmarks>=
for( int i=1;  i < ac;  i++ )
     marks->Add_File(av[i]);
```

Now, we return to conditionally adding files from the command line:

```
<possibly add files>=
if( adding_files )
  for( int i = 2;  i < ac;  i++ ) {
    if( access(av[i],R_OK) == 0 )
        marks->Add_File(av[i]);
    else
        warning("can't open file to add %s", av[i]);
  }
```

There is one other major order of business. In general, if we are reading a ZIP file, we want to synchronize the index file to the files actually in the archive. That is, if files have been added to the archive since the index file was created, we want to add them to the file list in the `Bookmark` class (we ignore files removed from the archive). We use the service routine we just postulated to achieve this. Note that this will also handle the initial population of the index from the ZIP if we are lacking an index file.

We perform the synchonization only if there is an index file present; if there is not an index file, then we just create

a Bookmark from the listing (we already know what the full complement of files are). We ignore the index and clipping files.

We finish this step by unlinking the index file, because we don't need it while we're reading–it's in memory–and we'll recreate it at the end:

```
<additional index processing, if needed>=
if( we_are_zip  )
{
    char *s;
    while((s=next_file_from_listing()) != NULL)
    {
        if( marks->Find_File(s) < 0  &&
            strcasecmp(s,INDEX) != 0 &&
            strcasecmp(s,PRINT) != 0 )
            marks->Add_File(s);
     }
     unlink(INDEX);
}
```

The Test Driver

This is slightly out of order, but we'll address some test code next. In the best of all possible worlds, we would provide a full set of test scripts to run in parallel with this. Instead, we provide some pretty simple-minded routines to exercise the methods in our Bookmark class. To ensure that the Bookmark suite is working, we need to examine the generated bookmark file foo.(Exercise for the reader: Build a more comprehensive test driver.)

```
<test main>=
main(int ac, char *av[])
{
     FILE *fp;
     fp = fopen("a","w"); fclose(fp);
     fp = fopen("b","w"); fclose(fp);
     fp = fopen("c","w"); fclose(fp);
     marks = new Bookmark();
     marks->Add_File("a");
     marks->Add_File("b");
     marks->Add_File("c");
     unlink("a"); unlink("b"); unlink("c");
     marks->Set_main(1L,27L);
     marks->Set('r',2L,84L);
     marks->Write("___foo");
}
```

Leftovers

We have a few leftover routines from the work above, which we'll write next time. In the meantime, we need to name them and provide their prototypes.

The first thing we need to do is display the files:

```
<process the files>=
process_the_files();
```

We also need code to extract the index and print files if we're reading from a ZIP archive:

```
<unzip the index and print files>=
unzip_index_print();
```

And prototypes for both routines:

```
<prototypes>=
   void unzip_index_print( void );
   void process_the_files( void );
```

Next time, we'll provide the code for these and also build the actual interface to less.

Until then, happy trails.

Jeffrey Copeland *(copeland@alumni.caltech.edu) lives in Boulder, CO, and works at Softway Systems Inc. on UNIX internationalization. He spends his spare time rearing children, raising cats and being a thorn in the side of his local school board.*

Jeffrey S. Haemer *(jsh@usenix.org) works at QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm, and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at http://alumni.caltech.edu/~copeland/work.