by Jeffreys Copeland and Haemer

PAUL STODDARD

# Reader, Part 2

Last month, we began building a reader for text files in C++. Our intention this month is to provide software that will allow you to metaphorically make annotations, mark pages you wish to double-check or bookmark where you stop reading.

As we've occasionally done in the past, this column is written using the `CWEB` literate programming tool, which melds program documentation and code into the same file for printing. You can pick up further information on `CWEB` from our Web page.

The module we'll be building this time needs some overall structure.

```
<header files>
<bookmark class definition>
<prototypes>
<global data>
<auxilary routines>;
```

We're also going to need some header files. Many of the header files will be familiar because they were used in last month's column.

```
<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>
```

Last time, we built the source code and methods for the `Bookmark` class, which stores the location where we stopped reading our list of files. So let's backtrack a little and include the header with our class prototypes.

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 only 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 all 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 are reading a ZIP archive; for convenience, we'll simply save the file name of the archive if we are. (We define a macro to make testing easier.) Also, if we are reading a ZIP, we unfurl the current file into the current directory. This means we need to access basename(file) if it is a ZIP we are reading, otherwise, we access 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)
```

Also, the Boolean type isn't 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 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 file 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"
```

Also, we need to define global instances of the Bookmark class and zip_name:

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

## Calling the Pager

Now is as good a time as any to remind you that we're going to adopt a modified version of less for the pager, which we'll call lessrdr. We'll use it through a local routine called page(). Wrapping an interface routine like page() around the pager provides us with a little insulation, making it easier to insert a different paging program later.

**_Wrapping an interface routine like page() around the pager provides us with a little insulation, making it easier to insert a different paging program later._**

Notice that despite all the setup we did in the class definition for supplementary bookmarks, we'll ignore them here because we don't have room in this column to cover them. So supplementary bookmarks are this month's main exercise for the reader. (Never fear. We'll supply our solution to the exercise in the form of a CWEB change file and some more modifications to less on our Web page.)

What do we need in our page() routine, and how do we need to modify less for what we're trying to achieve? We do know that in addition to the normal complement of functions a program like less provides, our pager will need some extras to deal with bookmarks. We could handle the bookmarks entirely in `page()`, but that would require calling directly into our C++ methods and require the pager to have more knowledge about the mechanics of bookmarks than we really want. Instead, we'll provide callback routines to service the reader's bookmark requests. (This will also provide further insulation of the pager from the reader body.)

Let's begin by assuming that we'll call `page()` with a file name and the normal complement of flags to `less`, such as a begin line. The user will proceed to look at this file, moving forward and back, interacting only with the code in the pager until he needs interaction with the calling routines for one of the following reasons:

• He wants to quit the program, saving the current location in the main bookmark.

• He wants to quit the program, but wants to abandon all the new bookmarks he set and does not want to set the main bookmark.

• He finishes the file and wants to move on to the next.

• He wants to return to the previous file.

• He sets a supplementary bookmark (which is part of the exercise for the reader).

• He wants to return to a previously saved bookmark (which is also part of the exercise for the reader).

We'll address each of these scenarios in the following sections.

We could define a `struct` for transmitting the file/line pair, but it will be easier to pass them around by reference. Notice that the pager is going to be sent a file name and line number and return a line number; the pager won't change files without interaction from the main program. Generally, the main program will keep its notion of the main bookmark in the local instantiation of `Bookmark` as `file_which` and `file_where`, modifying it when we interact with the pager. In other words, when we're in the pager, the main program's idea of bookmarks will be wrong and we'll synchronize them when we return to the main program.

Overall, the situation is as follows:

```
<process the files>=
int x;
do {
    <special handling before ZIP text>
    <ensure that we have the file>
    <display current file returning retval>
    <special handling after ZIP text>
    x = retval & ~BOOKMARK_MASK;
    switch( x ) {
    case RDR_NEXT:  marks->Next_File();  break;
    case RDR_PREV:  marks->Prev_File();  break;
    case RDR_QUIT:  break;
    case RDR_EXIT:  break;
    default:    fatal("odd return from pager");
                break;
    }
} while(  (x != RDR_QUIT)  &&
   (x != RDR_EXIT)  &&
   !marks->No_More_Files() );
```

In last month's column, we handwaved over the above code with a single call, to wit:

```
void
process_the_files( void )
{
    <process the files>;
}
```

We also need a routine for the pager interface to tell the main program where we left off:

```
<auxilary routines>=
void
SetMainBookmark(int loc)
{
    ssize_t xx = loc;
    marks->Set_main(marks->Current_FileNr(),xx);
}
```

Now, we'll define the interface between the main program and the pager:

```
<header files>=
static char *hdrid = "$Id: rdrr.h,v 1.2 1998/11/03 \
17:08:33 jeff Exp $";
#define RDR_NEXT (0x100)
#define RDR_PREV (0x200)
#define RDR_QUIT (0x400)
#define RDR_EXIT (0x800)
#define BOOKMARK_MASK (0xFF)
#define COOKIE  "__cookie"
#define PRT   "___prt"
```

A quick and easy thing to deal with next is the special handling for `ZIP`s. We need to extract the file from the `ZIP` before reading:

```
<special handling before ZIP text>=
if( we_are_zip )
  <unzip the named file>
```

We also need to delete the file if we extracted it from a `ZIP`, lest the directory get cluttered and confused–after all, we purposely stored our reading material in a `ZIP` archive in part to help with clutter.

```
<special handling after ZIP text>=
if( we_are_zip )
  unlink(basename(marks->Current_File()));
```

For insurance, we convince ourselves that we actually have the file before we call the pager-interface routine.

```
<ensure that we have the file>=
if( access(locate(marks->Current_File()),
      R_OK) != 0 )
{
    warning("file %s in index, not on disk\n",
        marks->Current_File());
    marks->Next_File();
    continue;
}
```

Displaying the current file involves preparing a calling sequence for the pager-interface routine, `page()`. We'll assemble the `argv[]` vector that we'll pass to `page()`, which in turn will execute `lessrdr`, our modified version of `less`. (For more on the flags we're passing to `lessrdr`, see the man page for `less`.)

```
<display current file returning retval>=
#define SMALLBUF 256
int pargc;
char *pargv[30];
int retval;
char b1[SMALLBUF];
char b2[SMALLBUF];
char b3[SMALLBUF];
struct stat sb;

pargc = 0;
pargv[pargc++] = "lessrdr";
pargv[pargc++] = "-q";  // operate quietly
pargv[pargc++] = "-r";  // leave ctrl chars
pargv[pargc++] = "-e";  // quit on 2nd EOF
pargv[pargc++] = "-i";  // ignore case in search
sprintf(b1,
    "-P[%%f, %ld of %d] ?e(end):%%p\\%%. ",
    marks->Current_FileNr()+1,
    marks->File_Count() );
pargv[pargc++] = b1;
//  add the flag to get us to the right line
if( marks->Current_Line() > 1 )
{
    sprintf(b2,"+%ld", marks->Current_Line());
    pargv[pargc++] = b2;
}
//  add a flag to buffer the whole file
//     in memory, if we can
if( stat(locate(marks->Current_File()),
        &sb) == 0 )
{
    sprintf(b3, "-b%d",
            (sb.st_size + 1023) / 1024);
    pargv[pargc++] = b3;
}
pargv[pargc++] =
    locate(marks->Current_File());
retval = page(pargc, pargv);
```

## The Pager Interface

We begin simply enough with a prototype for the routine we called above:

```
<prototypes>=
int page(int ac, char *av[]);
```

The pager interface must do a bunch of things with the arguments we assembled above. To wrap it up into code,

```
int
page(int pargc, char *pargv[])
{
    int len = 0, i;
    <find length of arguments>
    <assemble the command>
    <do it!>
    <read back the results>
    <store and return results>
}
```

We need to count the length of the arguments so we know how much space to allocate for the command line. We should also leave some space for possible quote marks. (We'll concede that this may be a little anal retentive of us. We could probably have just allocated a fixed, large buffer.)

```
<find length of arguments>=
for( i = 0;  i < pargc;  i++ )
{
    len += strlen(pargv[i]) + 3;
}
```

Our next step is to assemble the command into a single string:

```
<assemble the command>=
char *cmd = (char *) malloc(len+1);
strcpy( cmd, pargv[0] );
for( i = 1;  i < pargc;  i++ )
{
    strcat( cmd, " " );
    if( strchr(pargv[i],' ') != NULL )
        strcat(cmd, "\"");
    strcat( cmd, pargv[i] );
    if( strchr(pargv[i],' ') != NULL )
        strcat(cmd, "\"");
}
```

The process of invoking `lessrdr` should be pretty obvious. But we also remember to free the command buffer, lest we cause a memory leak.

```
<do it!>=
system(cmd);
free(cmd);
```

When we return, we need to read back the cookie that the external reader left us; we can delete it when we're done.

```
<read back the results>=
FILE *fp;
int retval, location;
fp = fopen(COOKIE,"r");
if( fp == NULL )
    fatal( "can't open cookie file!" );
fscanf(fp, "%d %d", &retval, &location);
fclose(fp);
unlink(COOKIE);
```

Those results need to be stored as a bookmark using the call-back routine we developed earlier:

```
<store and return results>=
if( retval == RDR_QUIT )
    SetMainBookmark(location);
return(retval);
```

When we're done and fall out of the big `do` loop above, either by finishing the last file or quitting, we need to wrap things up. We do this by writing the index file and saving it, if we're reading a `ZIP`. However, we don't write the index at all if we want to abandon all changes.

```
<process the files>=
if( x != RDR_EXIT )  marks->Write(INDEX);
if( we_are_zip )
    <zip the index and print files>
```

## Service Routines

We need to spend a few sections talking about service routines. We'll begin with the service routines to extract and return files to `ZIP` archives. We'll start by writing the routine to provide us, one at a time, with the list of files in a `ZIP` archive:

```
<auxilary routines>=
char *
next_file_from_listing( void )
{
    static FILE *fp = NULL;
    static char *listname;
    static char *buf;
    char *s;
    if( fp == NULL ) {
#ifdef DEBUG
        warning( "generating listing file\n", NULL );
#endif
        <generate the listing file>
    }
    if( fgets(buf,BUFSIZ,fp) == NULL ) {
        <clean up the listing file>
#ifdef DEBUG
        warning("next_file...() -> NULL\n",
              NULL );
#endif
        return NULL;
    }
    if( (s=strchr(buf,'\r')) != NULL )  *s = 0;
    if( (s=strchr(buf,'\n')) != NULL )  *s = 0;
    if( (s=strrchr(buf,' ')) != NULL )
        s++;
    else
        s = buf;
    <special handling if s points at caret>
#ifdef DEBUG
    warning( "next_file...() -> %s\n", s);
#endif
    return s;
}
```

We need to fill in the blanks from the last routine. First, let's generate the listing file and open it. At the same time, we'll set up the transient storage.

```
<generate the listing file>=
listname = (char *) malloc(FILENAME_MAX);
buf = (char *) malloc(BUFSIZ);
<pick a zip listing file name>
<zip listing>
if( (fp=fopen(listname,"r")) == NULL )
  fatal("can't open list file just built");
```

Next, we need to clean up after ourselves. To indicate that we don't have an active listing file, we set `FILE *` to `NULL:`

```
<clean up the listing file>=
fclose(fp);
unlink(listname);
free(listname);
free(buf);
fp = NULL;
```

The name of the file for the `ZIP` listing depends on a number of factors. In particular, if we're on a DOS system and we are getting the archive off a floppy, we want to put the listing file on the `C:` drive so as to not take up space on the floppy. (Exercise for the reader: What should this code do on DOS if you're operating out of RAMDISK?)

```
<pick a zip listing file name>=
#define LISTFILE "@@@"
if( zip_on_floppy() ) {
    strcpy(listname, "C:");
    strcat(listname,LISTFILE);
} else
    strcpy(listname,LISTFILE);
```

We need to deal with the situation on UNIX where the file name in the listing is preceded by a caret to indicate "case folding," that is, in a `ZIP` archive created on DOS. We don't need to fold this into an `#ifdef` because the caret is an illegal file name character in both DOS and UNIX.

```
<special handling if s points at caret>=
if( *s == '^' )  s++;
```

Errors need their own routines, as follows:

```
<auxilary routines>=
void fatal( char *msg )
{
    fflush(stdout);
    fprintf(stderr, "%s\n", msg );
    fflush(stderr);
    exit( 1 );
}

void warning( char *msg, char *s )
{
    fflush(stdout);
    fprintf(stderr, msg, s );
    fflush(stderr);
}
```

We've also got a bunch of holdovers for handling ZIP files. We'll handle them below, but first we need a service routine to deal with them. The pksystem() service routine should handle the placement of the working file so we don't overrun the floppy, but it doesn't.

(Exercise for the reader: Fix the pksystem() routine so that it can handle the possibility that we are reading off a floppy disk.)

```
<auxilary routines>=
void
pksystem( char *cmd, char *flag,
        char *flag2, char *file )
{
    char buf[BUFSIZ];
    sprintf(buf,"%s %s %s %s %s%s", cmd,
        "-q",
        flag, zip_name, flag2, file );
#ifdef DEBUG
    printf( "%s\n", buf );  sleep(1);
#endif
    system( buf );
}
```

We also need to define the ZIP utilities:

```
<header files>=
#define ZIP  "zip"
#define UNZIP "unzip"
```

In a couple of quick passes, we'll deal with calling that service routine. Some explanation is in order: When unzipping, we use the -o flag to force the overwriting of existing files (we want to look at the file in the archive, not an old local copy), and the -j flag to ignore the directory names on extract (we want to extract the name locally, and not leave directories as debris when we're done).

Similarly, for zipping the index and clipping files back into the archive, we use the -o flag to set the archive date to the latest member (this keeps our archive dated identically with the index file, that is, with our last read) and the -m flag, which moves the file back into the archive, removing it as debris from the local playing field. For the listing, the qq appendage lists only the file lines, without headers and trailers. The one-liner for the ZIP file listing is

```
<zip listing>=
pksystem(UNZIP,"-lqq",">",listname);
```

The code for extracting the index and print files is

```
<zip the index and print files>=
pksystem(ZIP,"-o -m","",INDEX_PRINT);
```

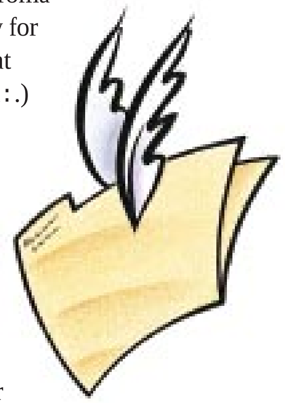
The code statment to extract a file is

```
<unzip the named file>=
pksystem(UNZIP,"-o -j","",
    marks->Current_File());
```

And here's one last utility routine, which we defined last month:

```
<auxilary routines>=
void
unzip_index_print( void )
{ pksystem(UNZIP,"-o","",INDEX_PRINT); }
```

We need the routine to tell us if our ZIP is a floppy for occasions like the floppy overrun test we didn't add to the pksystem() routine above. (Exercise for the reader: We never assume that we're reading froma floppy, but what's the correct strategy for figuring out if we are? Remember that our current disk on DOS might be A:.)



```
<auxilary routines>=
bool
zip_on_floppy(void)
{
    return false;
}
```

We need an interface like the familiar shell basename function:

```
<auxilary routines>=
char *
basename( char *path )
{
    char *s;
    s = strrchr(path,'/');
    if( s == NULL )  return path;
    return ++s;
}
```

We also need prototypes for all these routines:

```
<prototypes>=
void fatal( char *msg );
void warning( char *msg, char *s );
bool zip_on_floppy(void);
void pksystem( char *cmd, char *flags,
        char *flag2, char *file );
char *basename(char *path);
```

## Finishing Up

We're almost out of space, but we've left some information out. For example, you now have the wrapper code for the pager, but not the pager itself. The pager code is just a modified version of less, and the context differences are provided on our

Web page. But, more important, now that you have code to read text, where can you find some text? As we pointed out last time, we wrote this originally to allow us to read back issues of RISKS digests, which can be found at `ftp://ftp.sri.com:/risks`, but there are other sources of text:

• Biblomania, The Network Library, `http://www.bibliomania.com`

• U.S. Congressional Web site, which contains the text of bills introduced, `http://thomas.loc.gov`

• Open Book Initiative, `ftp://ftp.std.com/obi`

• Project Gutenberg, `http://www.gutenberg.net`

Furthermore, there is a whole list of improvements that can be made to our code. You may have noted some of them in the margins already. Perhaps you, Gentle Reader, will add one of the features below we haven't had time to build. If you do, let us know, and we'll add it to the code on our Web site.

• We'll repeat the one we've already mentioned: build the code to implement supplementary bookmarks.

• The Project Gutenberg version of *Alice's Adventures in Wonderland* is accompanied by GIF images of John Tenniel's illustrations for the original printed version. How would you display some text followed by a drawing, or even display them simultaneously?

• Can the formatting be improved from flat text? Can the open source version of a Web browser be used as the pager? How would you do the conversion from flat text input files to HTML on the fly?

• Can you think of a portable way to hide the index file on both DOS and UNIX? On DOS, a file is hidden by setting an attribute flag; on UNIX, it's hidden by beginning its name with a dot. We're not sure it's legal to have a DOS file with an empty name and just an extension.

• We haven't been very consistent in our strategy for loading text into RAMDISK on DOS. What improvements could be made?

• Can we implement an interactive index? In this mode, we would display the list of files in our current index and click on the one we want. This would allow us to skip from chapter to chapter in a book, or go backwards and forwards to chase references in RISKS.

• We need a command-line flag to print the version number.

• There are some flags in `ZIP` mode we haven't used but we might have. We used the `-o` flag so the archive retains the date of the latest file, but we could also have used the `-k` flag for Info-ZIP on UNIX to maintain compatibility with `PKZIP` files on DOS.

• When we print output into the clipping file, we should head each page with the file and line number.

• The wrapper program is fairly simple in concept. Could it be rewritten as a Perl script?

• As currently constituted, we need to keep the list of bookmarks in our head. Can you figure out a way to display and annotate them?

• Sequentially running this program in a single directory will overwrite an existing index file. Can you fix that?

As we mentioned in our November column, "A Short History of Reading," Page 58, commercial products are now available to handle the function of this code in stand-alone boxes. The most recent mentions of these products we've seen were in two wire-service stories in the *Rocky Mountain News*. In the first article (reprinted from *The Wall Street Journal*), NuvoMedia Inc. (`http://www.nuvomedia.com`), which offers the Rocket eBook, announced distribution agreements with publishers and says it expects to sell books electronically for $18 to $25. This price is close to the cost of a hardcover book, but the printing and distribution costs are essentially nil. It suggests that their marketing folks have come up with an absurd pricing model, or that the publishers–two of which are already investors in NuvoMedia–are insisting on more than 100% royalties on content. Part of the appeal of this technology is that the publishing cost is lowered; not pricing the content to reflect that difference will kill the idea.

In the other article, from Scripps Howard News Service (`http://www.shns.com`), both Martin Eberhard of NuvoMedia and Tom Pomeroy of SoftBook Press (`http://www.softbook.com`), maker of the SoftBook System, acknowledged that they are aiming their products at people who need a lot of data to be portable. Eberhard is quoted as saying, "This is designed for…the person who reads and travels a lot." We already carry a lot of hardware when we travel. Our normal mode of operation is to clear an airport with more weight in our briefcase than in our suitcase. Just thinking about adding an extra two pounds for the specialized hardware makes our shoulders hurt.

We want to issue a challenge to both companies: Because your profits are going to come from the content you're providing, why not provide reader software for your content to run on laptops? Most of those folks who read and travel a lot are already lugging around much more computing power than they need to run your applications. The real barriers to bringing your software to a general-purpose computer are security of the content and communications, but those problems are already solved with specialized hardware. If it's helpful, you can begin your laptop work with the software we provide here; we'll offer very nice royalty rates.

That's it from the Rocky Mountains for the beginning of 1999. Happy New Year and, until next time, 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`.