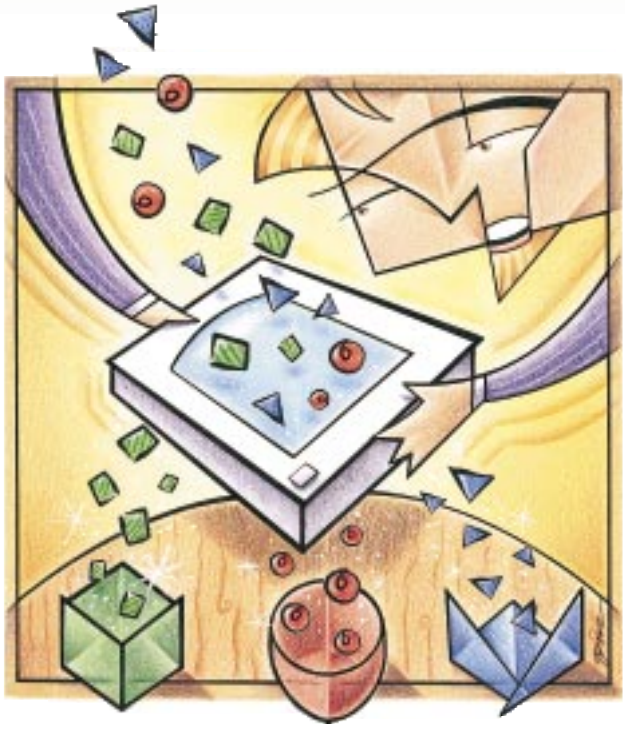
*by Jeffreys Copeland and Haemer*



PAUL STODDARD

# Cathedrals, Bazaars, and News Readers

Since the birth of the notion, late last year, of converting *SunExpert* exclusively to columns about bazaars and cathedrals, the editors, inspired by Eric Raymond's paper on Linux development entitled "The Cathedral and the Bazaar" (see `http://www.linuxresources.com/Eric/cathedral.html`), have taken the route of releasing "early and often." The initial release, Rich Morin's "Of Cathedrals and Bazaars," January 1998, Page 32, was succeeded by Mike O'Brien's longer version, "The Cathedral, the Bazaar and Mr. Protocol," April 1998, Page 24.

Peter Collinson had been assigned responsibility for the next release, but became lost while doing research at Canterbury cathedral, which is down the street from his house.

The editors then came to us, explaining that they'd decided to reassign the task of producing an interim release to us, owing to our familiarity with the bazaar. "Homonyms," we tried to explain, "are tricky things." "Jeffs," they said, "it's either that or wander around the Canterbury catacombs looking for Peter." As squeamish as we are underground, we immediately set pen to paper.

Because Peter obviously prefers to deal with real cathedrals than metaphorical ones–and by the time you read this Mike's and Rich's columns will be as far in the past as Boulder's winter snow–let's recap.

Eric's paper is an exploration of two different team approaches to software development. In the first, more traditional method, a single architect or a small group brings a software concept to fruition (that's "small" in the special sense that requires a team of two to implement UNIX, and a team of thousands to implement Windows NT). The group madly tests the software and doesn't release it until it is reasonably certain the program is relatively bug-free ("relatively bug-free" in that same special sense). The first external customers are usually beta testers who are sworn to secrecy ("secrecy" in the special sense that allows end users to be beta testers for some manufacturers' operating systems). The development cycle parallels that of a 15th-century cathedral.

By contrast, if software is built by a loosely coupled group of developers, each of whom is also a user and tester, in close contact via the Internet, the development cycle more closely resembles a noisy bazaar. In the bazaar model, software is released early and often. Feedback is constant. The software is often better debugged because the many hands have made light work of finding, characterizing and fixing the bugs.

Which brings us to *our* current problem. About two months ago, a friend suggested that we might want to read a bunch of Usenet articles on science fiction writer Arthur C. Clarke. We didn't have time to read them then, so we grabbed all the articles into files for reading at our leisure. Unfortunately, once it was too late, we remembered that outside of a threaded reader like `trn` or `tin`, we've lost all notion of the order in which the articles need to be

read. What does this have to do with Eric's cathedral and bazaar notion? Plenty.

First, we're writing this column with Donald Knuth and Silvio Levy's literate programming tool `CWEB`, which combines explanatory text formatted in TeX with code in C. We can write our code independently of the order in which it needs to be presented to the compiler. `CWEB` came out of Knuth's work on TeX, which in turn resulted from his work on the multivolume *Art of Computer Programming* series published by Addison-Wesley (see `http://www-cs-staff.stanford.edu/~knuth/`). *Art of Computer Programming*, with its long development cycle and unifying notions provided by a single person, is a prime example of the cathedral development methodology. TeX itself has a number of crossover features: while it's still the brainchild of one developer, it has an army of debuggers and developers of support software. Much of that support software follows the bazaar model of development.

**The task, says Knuth, is not to describe to the computer what to do, but to explain to another human being what we want the computer to do.**

So listen up. This article isn't just about a program. It *is* the program. Feed the source of this article to `ctangle`, which extracts the code from the `CWEB` source, present the resulting C source to `gcc` and you get executable code. (The task, says Knuth, is not to describe to the computer what to do, but to explain to another human being what we want the computer to do.) Because of the process of typesetting for printing in the magazine, we lose some features like cross-references and module numbering; you can get these back by running the source through `cweave` and `tex`.

Those in the dark about `CWEB` shouldn't feel too much at sea, this is more or less the same approach we've used in explaining code in our columns to date–a little explanation, followed by a little code, followed by a little more explanation–but with a little more structure. If you need more background information, see our Work columns on literate programming ("An Introduction to Literate Programming," *RS/Magazine*, January 1995, Page 26, and "Literate Programming: Parts I and II," *RS/Magazine*, February and March 1995, Pages 32 and 31, respectively), Knuth and Levy's *The CWEB System of Structured Documentation* (Addison-Wesley, 1994, ISBN 0-201-57569-8), Knuth's *Literate Programming* (Cambridge University Press, 1992, ISBN 0-937073-80-6), or just pick up the `CWEB` software from our Web site at `http://alumni.caltech.edu/~copeland/work.html`.

We're going to be looking at data from one of the great examples of the bazaar model in the universe, namely Usenet news. As we develop code, we're going to be following one of the principles Eric cites in his paper: "Good programmers know what to write. Great ones know what to rewrite and reuse."

To the extent we can, we'll try to steal ideas, if not code, from the news readers. So the code you'll read below owes some ideas to `trn`'s thread manager, a program called `mthreads`. Why don't we just use `mthreads` directly? Because it assumes a fair amount of the news database cruft is set up–remember we want to run this locally, not on our news server. It is also optimized to build onto an existing database of threads. We just want the simple case of threading a (relatively) small list of articles.

## Program Overview

Our goal is to take a list of news articles and emit the list in threaded order, that is, the order in which they relate to one another. That order is almost certainly not the same as the strict order in which they were written, nor is it likely to be the order in which they arrived at our news server. We rely on two little bits of information in the header of the news article to achieve our goal: the `Message-id` and `References` headers. The first gives a unique identifier for this article and the second lists every article that preceded it. We'll also save the article subject line and date for backup information.

Our main program is very simple. We read the list of articles from `stdin`, open them and then relegate processing of the articles to a subroutine. When we're done, we walk the resulting article tree in another subroutine and deliver the tree to `stdout`. Because we're operating as a strict filter, we don't need any command-line arguments:

```
<main program>=
main( )
{
   char buf[BUFSIZ];
   FILE *artf;

   while( fgets(buf,BUFSIZ,stdin) != NULL )
   {
     chomp(buf);
     artf = fopen(buf,"r");
     if( artf == NULL)
     {
       perror(buf);
       continue;
     }
     process_art(artf,buf);
     fclose(artf);
   }
   display_tree();
}
```

We also need some data structures here. The most important is going to be the structure for holding the article data. We need to assemble these structures into a tree so each one will potentially link to both a sibling (at the same level in the tree) and a child. Notice that we keep track of a list of references and point to both the first and last in that list:

```
<data structures>=
typedef struct _article {
  struct _article *sibling;
```

```
  struct _article *child;
  char *message_file;
  char *message_id;
  char *subject;
  struct _reference *refs;
  struct _reference *end_refs;
  time_t date;
} ART;
```

Instead of storing the references as a single string, we will store them as a linked list, which will make scanning the list of references a bit easier at the cost of complicating the storing of data into that structure:

```
<data structures>=
typedef struct _reference {
  struct _reference *next, *prev;
  char *reftext;
} REF;
```

A quick word about `Message-id`s: `Message-id` strings in news articles are made unique by containing a domain part and an article part, `<31415926@gateway.opennt.com>`, for example. A lot of handwaving happens in `mthreads` to separate the article and domain parts of the `Message-id` in order to save space in memory. On a PDP-11, this was necessary, but now we're typically running machines that have more main memory than our first PDP-11 had disk, so we won't bother. (Or as the Jeffreys keep debating among themselves, when is it OK to be profligate with memory rather than CPU cycles?). We're going to assume that `References` lines grow to the right, which may result in some misplacements.

## Reading the Files

Each time we open an article file, we need to read the headers and then place the article data into the tree of articles in some reasonable fashion. Like the main program, the structure of this routine is pretty simple, but the problem is in the underlying details. The `place_article()` routine is sufficiently complicated that we'll be putting it off a while.

```
<service routines>=
process_art( FILE *art_fp, char *art_name )
{
  <process_art local variables>
  <allocate and initialize an article>
  <parse the article headers>
#ifdef DEBUG
  show_headers(art);
#endif
  place_article(art);
}
```

Local variables are an interesting problem. We know before the fact that we're going to need an `article` structure and a buffer to read lines into. We'll also need some pointers into the buffer. We'll add to this list later. One of the joys of literate programming, like writing in C++, is that we can declare variables as we need them:

```
<process_art local variables>=
char buf[BUFSIZ];
ART *art;
char *s, *t;
```

Allocating the `article` is very simple. We also initialize the `message_file` entry:

```
<allocate and initialize an article>=
 art = (ART *) malloc(sizeof(ART));
 art->message_file = strdup(art_name);
 art->sibling = art->child = NULL;
 art->message_id = NULL;
 art->refs = art->end_refs = NULL;
```

Now we parse the article headers. We read from the `article` file up to a blank line–the end of the header lines. We're only interested in a few of the headers, though. We'll postulate a useful routine, `headerEQ`, which checks a case-invariant header tag. If it matches, the routine returns a pointer to the text following the header; otherwise, it returns `NULL`. On matches, it also strips the trailing new line. In the case of the date header, we'll use a variation `getdate()` parser, which Steve Bellowin wrote while he was at the University of North Carolina, and which is supplied with the `trn` code to convert the date string into a `time_t`. The `References` lines are a special case: We need to get all the continuation lines for them, so we invoke a different paragraph of code to do so. It is important that we check `References` first for reasons we explain in the next module.

```
<parse the article headers>=
while ( fgets(buf,BUFSIZ,art_fp) != NULL )
{
  if( *buf == '\n' )
      break;
  if( (s = headerEQ("references",buf)) )
    <get all references lines>
  if( (s = headerEQ("message-id",buf)) )
    art->message_id = strdup(s);
  else if( (s = headerEQ("subject",buf)) )
    art->subject = strdup(s);
  else if( (s = headerEQ("date",buf)) )
    art->date = getdate(s);
}
```

We want to collect all the continuation lines for `References` headers, so that we can have *all* the references. Continuation lines begin with a space or tab. Notice we fall out of this routine when we find a noncontinuation line, passing that line unprocessed to the main header parser in the previous module:

```
<get all references lines>=
{
```

```
  extract_refs(art, s);
  while( fgets(buf,BUFSIZ,art_fp) != NULL )
  {
    chomp(buf);
    if( *buf != ' '  &&  *buf != '\t' )
        break;
    extract_refs(art, buf);
  }
}
```

We need to talk about the routine that extracts references from the `References` line. We're given the article pointer, so we can update the first and last references. We scan for each reference on the line delimited by open and close angle brackets and add it to the linked list of `REF`s in the article:

```
<service routines>=
void
extract_refs(ART *art, char *line)
{
  REF *this;
  REF *last;
  char *s, *t;
  char save;
  s = line + strspn(line," \t");
  while( s && *s )
  {
    if( (s = strchr(s,'<')) == NULL )
        return;
    if( (t = strchr(s,'>')) == NULL )
        return;
    this = malloc(sizeof(REF));
    save = *(++t);
    *t = 0;
    this->reftext = strdup(s);
    this->next = NULL;
    this->prev = art->end_refs;
    if( this->prev )
        this->prev->next = this;
    if( art->refs == NULL )
        art->refs = this;
    art->end_refs = this;
    *t = save;
    s = t;
  }
}
```

## Utility Routines

We've used a number of utility routines that we haven't defined yet. We'll start with the easy Perl analog, `chomp()`, which removes the trailing new line from a string:

```
<service routines>=
char *
chomp(char *buf)
{
  char *s;
  s = strpbrk(buf,"\r\n");
  if( s )
    *s = 0;
  return buf;
}
```

We need to define the `headerEQ()` routine, too. We do a case-insensitive comparison of the given string against the supplied buffer and return a pointer the first character of the header line's text. As a side effect, we remove the trailing new line on matching lines.

```
<service routines>=
char *
headerEQ(char *hdr, char *buf)
{
  char *s;
  if( strncasecmp(hdr,buf,strlen(hdr)) != 0)
    return NULL;
  chomp(buf);
  if( (s = strchr(buf,':')) == NULL )
    s = strchr(buf,' ');
  while( isspace(*(++s)) )  continue;
  return s;
}
```

A debugging routine:

```
<service routines>=
void
show_headers(ART *art)
{
  REF *t;
  printf("===== ");
  printf("%s: date %ld; subj %.25s\n id %s\n",
    art->message_file, art->date,
    art->subject, art->message_id);
  printf(" refs from 0x%x to 0x%x\n",
    art->refs, art->end_refs);
  for( t = art->refs;  t;  t = t->next )
    printf("  %s @ 0x%x\n", t->reftext, t);
}
```

We finish up the utilities by completing our collection of function prototypes:

```
<function prototypes>=
char *headerEQ(char *, char *);
char *chomp(char *);
void extract_refs(ART *, char *);
void show_headers(ART *);
```

## Wrapping Up

We've almost run out of space for this month, but there's one thing we need to finish. We need to outline the complete program by collecting all the source code together and wrapping it with `include` files:

```
#define _ALL_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
<data structures>
<function prototypes>
<main program>
<service routines>
```

We've left the playing field with a big blank space in the middle, much like that old *American Scientist* cartoon where there's a flock of equations on the left side of the blackboard, a flock of equations on the right side, and, in the

middle, the legend "and then a miracle occurs." (The caption reads, "I think you're a little vague in step two here.")

In our case, the missing miracle comes in the form of the `place_article()` routine, which drops the parsed article into the tree, and the `display_tree()` routine, which prints the final result.

We'll cover those and other stories from the bazaar next month.

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.html`.