by Jeffreys Copeland and Haemer

PAUL STODDARD

Virtual Threaded News Reader

When last we left our plucky heroes, they had started to organize a list of Usenet news articles into useful order. The task was to take a random list of articles and output that list in the order in which a threaded reader like `trn` might present them. We had finished the input processing when we ran out of time, leaving us with a large blank space on our blackboard, onto which was scrawled "and then a miracle occurs."

This month, we'll show you how to drop articles into the data structure we invented last time, and present a quick recursive routine to print the list of articles in threaded order. As we mentioned last month, we're writing this column using Donald Knuth and Silvio Levy's `CWEB` literate programming tool, so the column doesn't contain the source code; the original text of the column *is* the source code (for details, see our Web site).

For review, the two important data structures are for the article itself and for the references to its predecessors. We'll present them here so you don't have to flip back to last month's column. (We're cheating: We're producing this code as a separate source module and we *should* define these structures in an `include` file. Instead, we've just copied the lines from last month.)

```
<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;

typedef struct _reference  {
  struct _reference *next, *prev;
  char *reftext;
} REF;
```

There are some bits of nomenclature we need to get out of the way first. An article's siblings are articles with the same number of references. In a diagram, we list them vertically down the page. An article's children are articles that refer to it, and we list them horizontally, growing to the right.

News articles have ID strings of the form `<31415926@opennt.com>`. These strings appear in both the `Message-id` and `References` headers of the articles. For convenience, we'll refer to these symbolically in our examples (`Message-id`s with italic letters, and we'll use a colon to separate the ID from the references) so an article with ID *j* may have references, or *j:adf*, which point to articles *a*, *d* and *f*. So, when we enter `place_article()`, we may have an existing tree of articles:

```
a:-    d:a    y:adx
       e:a    g:ae
b:-    q:b
       r:b    w:br    z:brw
              h:br
c:-
```

Given an article *x:ad,* for example, we would insert it in the top row of this diagram between *d:a* and *y:adx*. (You may find it easier to envision this as a directory tree. In this analogy, the root directory of the file system contains the articles with no references. When we find an article with references, its "path name" consists of the references in some order.)

We also know what the overall structure of this module will be:

```
#define _ALL_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
<data structures>
<service routines>
<place the articles>
<print the tree>
```

We also need the head of the data tree:

```
<data structures>=
ART *root;
```

Insertion Sorts

As Beth, our local Lisp hacker put it, after listening to a long-winded explanation of the problem, "It's just an insertion sort. Check for a null `cdr` and recurse."

So we break our insertion into the tree of each new article into two basic cases. In the base case, where the existing root is `null`, we insert this article as the root. In the "normal" case, we insert it somewhere into the tree, as in our diagram above.

With that in mind, we can begin to lay out the top level of `place_article()`:

```
<place the articles>=
void
place_article( ART *art )
{
  <null root case>
  <normal case>
}
```

Let's begin with the simplest case:

```
<null root case>=
if( root == NULL )
{
  root = art;
  return;
}
```

Because the normal case is recursive, we front it with a function call, as follows:

```
<normal case>=
normal_case(root, art);
```

The body of the normal case breaks down into some simple cases. We want to loop through the siblings of the root, examining each in turn. If the sibling we're examining is referred to by `art`, we recurse with the root set to the child of the sibling. In effect, `art` has asked the sibling, "are you my mother?" (In the special case, where the sibling has no child, we insert the new article as its child.)

If, on the other hand, `art` appears in the list of references for the sibling we are examining, then `art` is actually the parent of the sibling. (In effect, the sibling is asking `art`, "are you *my* mother?") In this case, we insert `art` as the parent of the sibling. If we fall through the list of siblings without finding a hit, we insert `art` as a new sibling.

(Returning to the directory analogy, we check the files [articles] in each directory to see if the new file we're trying to insert has this file in its path name. If it does, we change into the subdirectory and repeat the process. If the new file has no files from this directory in its path name, then we park the file here for the time being.)

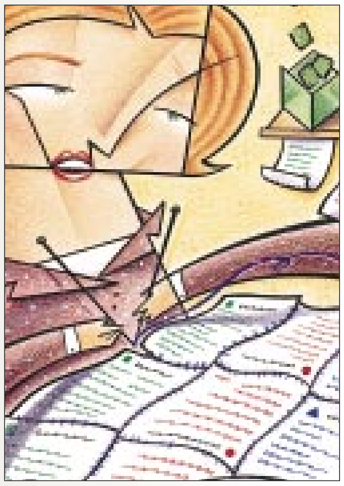
```
<service routines>=
void
normal_case( ART *root, ART *art )
{
  ART *p, *s;

  s = NULL;
  p = root;
  while( p != NULL )
  {
    if( refs_in_list(p->message_id, art->refs) )
    {
      if( p->child )
      {
        normal_case(p->child, art);
        return;
      } else {
        append_child(p, art);
        return;
      }
    }
    if( refs_in_list(art->message_id, p->refs) )
    {
      insert_parent(p, art);
      return;
      }
      s = p;
      p = p->sibling;
    }
    append_sibling(s,art);
}
```

Service Routines

We've postulated some routines that we now need to write: a routine to cross-check references and a flock of insertion routines. (Exercise for the reader: What's the collective noun for software? Would this be a Dilbert of insertion routines?)

Notice that we've chosen to use singly-linked lists for our data structures, which means we need to go to a little more trouble to keep track of the structure links in our insertion routines. With the complications caused by children and parents and siblings and single-links, this could turn out to be as difficult a job as being the booking agent for the Jackson Five.

We've chosen to use singly-linked lists for our data structures, which means we need to go to a little more trouble to keep track of the structure links in our insertion routines.



(Of course, as much fun as we have with this writing gig, we too have suffered at the hands of booking agents. Haemer, for example, once was booked to speak at a Usenix conference immediately *after* Penn Gillette. We keep thinking that if only we had really good agents, one of us would be attempting to hit home runs off Robin Roberts of the Phillies and the other would be wearing a bowler and tooling around England with Uma Thurman. But we digress.)

We'll begin by checking references. If the `Message-id` appears in the specified `References` list, then we return true:

```
<service routines>=
int
refs_in_list( char *id, REF *ref )
{
  REF *p;

  for( p = ref;  p != NULL;  p = p->next )
    if( strcmp(id, p->reftext) == 0 )
      return 1;
  return 0;
}
```

Now we can begin the insertion routines. The first hangs the given article off the current one as its child, thus:

```
<service routines>=
append_child( ART *current, ART *newart )
{
#ifdef DEBUG
  printf("@@@ insert %s/%s as child of %s/%s\n",
    newart->message_file, newart->message_id,
    current->message_file, current->message_id);
#endif
  current->child = newart;
}
```

The next insertion routine hangs the new article off the current one as a sibling:

```
<service routines>=
append_sibling( ART *current, ART *newart )
{
#ifdef DEBUG
  printf("@@@ insert %s as sibling of %s/%s\n",
    newart->message_file, newart->message_id,
    current->message_file, current->message_id);
#endif
  current->sibling = newart;
}
```

The last one of these is the most complicated. We need to exchange the new article and the current article, making the current one the child of the new one. That is, we're inserting *q:krs* in front of *w:krsq*. (Using the directory analogy, we've got a file that doesn't belong in this directory, but rather in a sub-directory, and we've found the parent of that path name. We put the new file in this directory and put the other file in the subdirectory.) This one is complicated because of the singly-linked lists we're using–we don't necessarily know the parent or older sibling of the current article, so we exchange the data, leaving the links to the current article intact. We'll do this in a few steps, to keep our sanity:

```
<service routines>=
insert_parent( ART *current, ART *newart )
{
  ART *temp;

#ifdef DEBUG
  printf("@@@ insert %s as parent of %s/%s\n",
    newart->message_file, newart->message_id,
    current->message_file, current->message_id);
#endif
  temp = malloc(sizeof(ART));
  <copy current to temp>
  <copy newart into current>
  <link temp as current's child>
  <clean up after parent insert>
}
```

First, we copy the current article to the temporary one, ensuring that we don't have dangling pointers to siblings or children:

```
<copy current to temp>=
temp->message_file  = current->message_file;
temp->message_id    = current->message_id;
temp->subject       = current->subject;
temp->refs          = current->refs;
```

```
temp->end_refs      = current->end_refs;
temp->date          = current->date;
temp->sibling       = NULL;
temp->child         = current->child;
```

Then, we copy the data from the new article into the current structure:

```
<copy newart into current>=
current->message_file  = newart->message_file;
current->message_id    = newart->message_id;
current->subject       = newart->subject;
current->refs          = newart->refs;
current->end_refs      = newart->end_refs;
current->date          = newart->date;
```

To complete the links we need to hang the temporary structure as a child of `current`:

```
<link temp as current's child>=
current->child = temp;
```

Finally, we need to clean up by freeing some allocated memory. We only need to free a structure because we've copied its contents. But which structure? Almost paradoxically, it's the new article, `newart`, whose data is now stored in `current`:

```
<clean up after parent insert>=
free(newart) ;
```

Showing the Results

Printing the results is the next step. This, too, is a recursive process. We'll begin with the function declaration and our interface from last month's article.

If we're debugging, we want to separate the trace output as we read and constructed the tree from the display of the tree itself:

```
<print the tree>=
display_tree()
{
#ifdef DEBUG
  printf("\n\n==========\n");
#endif
  display_subtree(root);
}
```

We want to show each subtree in chronological order. This is made easier because we've already stored the article time stamps from the UNIX epoch in a variable of type `time_t`, correcting for time zone differences. This means that (barring a posting host with its clock set incorrectly) the subtrees should be in chronological order if we run through the siblings in the order of the time stamps. We reset the time to zero after we've processed each sibling's subtree. Thus, we look through the list of siblings repeatedly, picking out the oldest, printing it and processing its children, until we have no more siblings with non-zero time stamps:

```
<print the tree>=
display_subtree( ART *root )
{
  ART *p, *earliest;

#ifdef DEBUG
  printf("displaying subtree from %s\n",
    root->message_id);
#endif
  do  {
    earliest = NULL;
    for( p = root;  p != NULL;  p = p->sibling )
    {
      if( p->date == (time_t) 0 )
        continue;
      if( !earliest  ||
          earliest->date > p->date )
        earliest = p;
    }
    if( earliest ) {
      <show this article>
      if( earliest->child )
        display_subtree(earliest->child);
      earliest->date = (time_t) 0;
    }
  } while( earliest != NULL );
}
```

Showing the results of the article in question is very easy. For our purposes, we just print the article file name. We also optionally provide some debugging output showing the sibling and parent of this article. However, we can envision applications where we'd want to do something more complicated, such as to provide a graphic representation of the tree structure:

```
<show this article>=
printf("%s", earliest->message_file);
#ifdef DEBUG
if( earliest->sibling )
  printf(" v%s",
        earliest->sibling->message_file);
if( earliest->child )
  printf(" >%s",
        earliest->child->message_file);
#endif
printf("\n");
```

Wrapping Up

That's more or less it. Like the other software we've written for this series, this represents a real problem that we've run across and solved. The problem provides an interesting technique or lesson: In this case, the problem is solved by a good data structure. Programming around the data structure gives us some interesting utilities. You probably won't run into the same problem, but if you do, you've now got software to solve it. More likely, you'll run into a problem–it may be sitting on your desk right now–where data structures like these, or other tricks we have shown you, will be useful.

Meanwhile, we took a few minutes to visit Chez Protocol earlier in the month and talk to Mike O'Brien about cathedrals and bazaars, our launching point for last month's column. In April ("The Cathedral, the Bazaar and Mr. P.," *SunExpert*, Page 24), he noted that Mr. Protocol has "this picture–I think it's the only one he owns–which has a single vertical black bar on it, and underneath, in script, the words, 'Ceci n'est pas une pipe.' I don't want to think about it."

We were amused to discover that's not exactly the whole story. In fact, in Mr. P.'s wing of Chez Protocol, one wall is entirely covered with the following text:

```
%!
% The Betrayal of Special Characters


/center {
  dup 8.5 72 mul 2 div exch stringwidth
  pop 2 div sub 3 2 roll moveto show
} def
/Helvetica 512 selectfont
11 72 mul 500 sub (\174) center

/KuenstlerScript-Black 50 selectfont
11 72 mul 600 sub
    (Ceci n'est pas une pipe) center

showpage
```

This, of course, means that Mr. Protocol's mutterings aren't the only thing Mike interprets.

What next? We may explore data compression. Or we may discuss some database problems and how UNIX is better than single-purpose tools for solving them. Or we may talk about entertaining off-by-one bugs we've encountered. Or we may spend the month brewing beer. 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`.