TWINX

June 11, 2023 at 13:13

1. Introduction. This short program compiles a master index for a set of programs that have been processed by `CTWILL`. To use it, you say, e.g., `twinx *.tex >index.tex`. The individual programs should define their names with a line of the form '`\def\title{NAME}`'.

```
#include <stdio.h>
  ⟨ Type definitions 4 ⟩
  ⟨ Global variables 2 ⟩
  ⟨ Procedures 5 ⟩
  main(argc, argv)
      int argc;
      char *argv[];
  {
    ⟨ Local variables 9 ⟩;
    ⟨ Initialize the data structures 8 ⟩;
    while (--argc) {
      f ← fopen(*++argv, "r");
      if (¬f) fprintf(stderr, "twinx: Couldn't open file %s for reading!\n", *argv);
      else {
        ⟨ Scan file f until coming to the title 3 ⟩;
        fclose(f); strncpy(*argv + strlen(*argv) − 3, "idx", 3); f ← fopen(*argv, "r");
        if (¬f) fprintf(stderr, "twinx: Couldn't open file %s for reading!\n", *argv);
        else {
          ⟨ Copy the index file f into the data structures 10 ⟩;
          fclose(f);
        }
      }
    }
    ⟨ Output the data structures to make a master index 13 ⟩;
    return 0;
  }
```

2. `#define buf_size 100` ▷ input lines won't be this long ◁

```
⟨ Global variables 2 ⟩ ≡
  FILE *f;
  char buf[buf_size];
  char title[buf_size];
  char cur_name[buf_size];
```

See also sections 7 and 18.

This code is used in section 1.

3.

```
⟨ Scan file f until coming to the title 3 ⟩ ≡
  while (1) {
    if (fgets(buf, buf_size, f) ≡ Λ) {
      fprintf(stderr, "twinx: (no title found in file %s)\n", *argv); title[0] ← '\0'; break;
    }
    if (strncmp(buf, "\\def\\title\{", 11) ≡ 0) { register char *p, *q;
      for (p ← buf + 11, q ← title; *p ∧ *p ≠ '}'; p++) *q++ ← *p;
      *q ← '\0'; break;
    }
  }
```

This code is used in section 1.

4. Data structures. Our main task is to collate a bunch of texts associated with keys that have already been sorted. It seems easiest to do this by repeatedly merging the new data into the old, even though this means we'll be passing over some of the same keys 30 times or more; the computer is fast, and this program won't be run often.

Further examination shows that a merging strategy isn't so easy after all, because the sorting done by `CTWILL` (and by `CWEAVE`) is weird in certain cases. When two index entries agree except for their "ilk," the order in which they appear in the index depends on the order in which they appear in the program. Thus, they might well appear in different order in two of the indexes we are merging. (There's also another glitch, although not quite as devasting: When two index entries have the same letters and the same ilk, but differ with respect to uppercase versus lowercase, the order in which they appear depends on the hash code used in `CWEB`'s `common.w` code!)

So we'll use Plan B: All index entries will first be copied into a long list. The list will almost always consist of many sorted sublists, but we will not assume anything about its order. After all the copying has been done, we will use a list-merge sort to finish the job.

The data structure is built from nodes that each contain three pointers. The first pointer is to an *id* string; the third pointer is to the *next* node; and the second pointer is either *data.s*, a pointer to a string of text, or *data.n*, a pointer to a node. In the main list, the *id* fields are the keys of the index, and the *data.n* fields point to lists of associated texts. In the latter lists, the *id* fields are the individual program titles, while the *data.s* fields are the texts.

```
⟨ Type definitions 4 ⟩ ≡
  typedef union {
    char *s;
    struct node_struct *n;
  } mixed;
  typedef struct node_struct {
    char *id;
    mixed data;
    struct node_struct *next;
  } node;
```

This code is used in section 1.

5. We copy strings into blocks of storage that are allocated as needed. Here's a routine that stashes away a given string. It makes no attempt to handle extremely long strings, because such strings will arise only if the input is all screwed up.

```
#define string_block_size 8192 ▷ number of bytes per string block ◁
⟨ Procedures 5 ⟩ ≡
  char *save_string(s)
       char *s;
  {
    register char *p, *q;
    register int l;
    for (p ← s; *p; p++) ;
    l ← p − s + 1;
    if (l > string_block_size) {
      fprintf(stderr, "twinx: Huge string `%.20s...' will be truncated!\n", s);
      l ← string_block_size; s[l − 1] ← '\0';
    }
    if (next_string + l ≥ bad_string) {
      next_string ← (char *) malloc(string_block_size);
      if (next_string ≡ Λ) {
        fprintf(stderr, "twinx: Not enough room for strings!\n"); exit(−1);
      }
      bad_string ← next_string + string_block_size;
    }
    for (p ← s, q ← next_string; *p; p++) *q++ ← *p;
    *q ← '\0'; next_string ← q + 1; return next_string − l;
  }
```

See also sections 6, 17, and 20.

This code is used in section 1.

6. Nodes are allocated with a similar but simpler mechanism.

```
#define nodes_per_block 340
⟨ Procedures 5 ⟩ +≡
  node *new_node( )
  {
    if (next_node ≡ bad_node) {
      next_node ← (node *) calloc(nodes_per_block, sizeof(node));
      if (next_node ≡ Λ) {
        fprintf(stderr, "twinx: Not enough room for nodes!\n"); exit(−2);
      }
      bad_node ← next_node + nodes_per_block;
    }
    next_node++; return next_node − 1;
  }
```

7.

```
⟨ Global variables 2 ⟩ +≡
  char *next_string, *bad_string;
  node *next_node, *bad_node;
  node header; ▷ the main list begins at header.next ◁
  node sentinel; ▷ intermediate lists will end at this node ◁
```

8. We don't really have to initialize the string and node storage pointers, because global variables are zero already. But we might as well be tidy and state the initial conditions explicitly.

It will be convenient to have extremely small and large keys in the dummy nodes.

```
⟨ Initialize the data structures 8 ⟩ ≡
  next_string ← bad_string ← Λ; next_node ← bad_node ← Λ; header.next ← Λ; header.id ← "␣␣{";
    ▷ smaller than any valid id ◁
  sentinel.id ← "␣␣{\200}";    ▷ larger than any valid id ◁
  main_node ← &header;
```

See also section 19.

This code is used in section 1.

9.

```
⟨ Local variables 9 ⟩ ≡
  register node *main_node;    ▷ current end of main list ◁
```

This code is used in section 1.

10. Copying. Lines in the index file f that we're reading either begin a new entry or continue a long entry. In the first case, the line begins with `\I` and then either `\\{key}` or `\|{key}` or `\.{key}` or `\&{key}` or `\${key}` or `\9{key}` or just `␣␣{key}`. (These correspond to multi-character italic, single-digit italic, typewriter, bold, custom, variable, and roman styles.) In the second case, the line begins with a page number or `\[`; however, we recognize the second case by the fact that the previous line did not end with a period.

```
⟨ Copy the index file f into the data structures 10 ⟩ ≡
  while (1) { register node *cur_node;
    if (fgets(buf, buf_size, f) ≡ Λ) break;      ▷ end of file ◁
    if (strncmp(buf, "\\I", 2) ≡ 0) {
      ⟨ Copy a new index entry into cur_name and cur_node 11 ⟩;
      main_node→next ← new_node( ); main_node ← main_node→next;
      main_node→id ← save_string(cur_name); main_node→data.n ← cur_node;
    }
    else if (buf[0] ≠ '\n')
      fprintf(stderr, "twinx:␣couldn't␣deal␣with␣`%.10s...'␣in␣file␣%s!\n", buf, *argv);
  }
```

This code is used in section 1.

```
11. ⟨ Copy a new index entry into cur_name and cur_node 11 ⟩ ≡
  if (buf[4] ≠ '{') {
    fprintf(stderr, "twinx:␣missing␣brace␣in␣file␣%s:␣`%.20s...'\n", *argv, buf); break;
  }
  { register char *p, *q; register int bal ← 1;
    cur_name[0] ← buf[2]; cur_name[1] ← buf[3]; cur_name[2] ← '{';
    for (p ← buf + 5, q ← cur_name + 3; *p ∧ (bal ∨ *p ≡ '{'); p++) {
      if (*p ≡ '{') bal++;
      else if (*p ≡ '}') bal−−;
      *q++ ← *p;
    }
    if (bal) {
      fprintf(stderr, "twinx:␣unbalanced␣entry␣in␣file␣%s:␣`%.20s...'\n", *argv, buf); break;
    }
    if (*p++ ≠ ',') {
      fprintf(stderr, "twinx:␣missing␣comma␣in␣file␣%s:␣`%.20s...'\n", *argv, buf); break;
    }
    if (*p++ ≠ '␣') {
      fprintf(stderr, "twinx:␣missing␣space␣in␣file␣%s:␣`%.20s...'\n", *argv, buf); break;
    }
    *q ← '\0'; ⟨ Copy the text part of the index entry into cur_node 12 ⟩;
  }
```

This code is used in section 10.

12. When we get here, p points to the beginning of the text following a key in the index. The index entry ends with the next period, possibly several lines hence. In the multiple-line case, *cur_node* will point to the final line, which points to the penultimate line, etc.

```
⟨ Copy the text part of the index entry into cur_node 12 ⟩ ≡
  { int period_sensed ← 0;
    node *continuation;
    cur_node ← new_node( ); cur_node→id ← save_string(title); do {
      for (q ← p; *q ∧ *q ≠ '\n' ∧ *q ≠ '.'; q++) ;
      if (*q ≡ '.') period_sensed ← 1;
      *q ← '\0'; cur_node→data.s ← save_string(p);
      if (period_sensed) break;
      continuation ← new_node( );      ▷ the id field is Λ ◁
      continuation→next ← cur_node; cur_node ← continuation; p ← buf;
    } while (fgets(buf, buf_size, f));
    if (¬period_sensed) {
      fprintf(stderr, "twinx: File %s ended in middle of entry for %s!\n", *argv, cur_name);
      break;
    }
  }
```

This code is used in section 11.

13. Sorting. Let us opt for simplicity instead of tuning up for speed. The idea in this step is to take a list that contains k ascending runs and reduce it to a list that contains $\lceil k/2 \rceil$ runs, repeating until $k = 1$. We could make the program about twice as fast if we took the trouble to remember the boundaries of runs on the previous pass; here, every pass will be the same.

```
⟨ Output the data structures to make a master index 13 ⟩ ≡
  ⟨ Sort the main list, collapsing entries with the same id 14 ⟩;
  ⟨ Output the main list in suitable TEX format 21 ⟩;
```

This code is used in section 1.

14. The *compare* subroutine, which specifies the relative order of *id* fields in two nodes, appears below. Let's get the sorting logic right first.

The algorithm is, in fact, rather pretty—I hate to say cute, but that's the word that comes to mind. Some day I must write out the nice invariant relations in these loops. Too bad it's not more efficient.

Remember that $header.id$ is $-\infty$ and $sentinel.id$ is $+\infty$. Also remember that the main list begins and ends at the header node.

```
⟨ Sort the main list, collapsing entries with the same id 14 ⟩ ≡
  main_node→next ← &header;
  while (1) { register node *p, *q, *r, *s, *t;
    t ← &header; r ← t→next;
    while (1) {
      if (r ≡ &header) break;
      p ← s ← r; ⟨ Advance s until it exceeds r ← s→next 15 ⟩;
      if (r ≡ &header) break;
      s→next ← &sentinel; q ← s ← r; ⟨ Advance s until it exceeds r ← s→next 15 ⟩;
      s→next ← &sentinel; ⟨ Merge p and q, appending to t 16 ⟩;
      t→next ← r;
    }
    if (t ≡ &header) break;
  }
```

This code is used in section 13.

15.

```
⟨ Advance s until it exceeds r ← s→next 15 ⟩ ≡
  do { register int d;
    r ← s→next; d ← compare(s, r);
    if (d > 0) break;        ▷ s→id > r→id ◁
    if (d ≡ 0) {             ▷ s→id ← r→id ◁
      collapse(s, r);        ▷ put r's data into s's list ◁
      s→next ← r→next;       ▷ node r will be unclaimed garbage ◁
    }
    else s ← r;              ▷ this is the normal case, s→id < r→id ◁
  } while (1);
```

This code is used in section 14.

16. Merging takes place in such a way that sorting is stable. Thus, index entries for a key that appears in different programs will remain in the order of the `.tex` files on the command line.

```
⟨ Merge p and q, appending to t 16 ⟩ ≡
  do { register int d;
    d ← compare(p, q);
    if (d > 0) {        ▷ p→id > q→id ◁
      t→next ← q; t ← q; q ← q→next;
    }
    else if (d < 0) {        ▷ p→id < q→id ◁
      t→next ← p;        ▷ p→id < q→id ◁
      t ← p; p ← p→next;
    }
    else if (p ≡ &sentinel) break;
    else {
      collapse(p, q);        ▷ put q's data into p's list ◁
      q ← q→next;
    }
  } while (1);
```

This code is used in section 14.

17. Comparison is a three-stage process in general. First we compare the keys without regarding case or format type. If they are equal with respect to that criterion, we try again, with case significant. If they are still equal, we look at the format characters (the first two characters of the *id* field).

```
⟨ Procedures 5 ⟩ +≡
  int compare(p, q)
      node *p, *q;
  { register unsigned char *pp, *qq;
    for (pp ← (unsigned char *) p→id + 3, qq ← (unsigned char *) q→id + 3; *pp ∧ ord[*pp] ≡ ord[*qq];
        pp++, qq++) ;
    if (*pp ∨ *qq) return ord[*pp] − ord[*qq];
    for (pp ← (unsigned char *) p→id + 3, qq ← (unsigned char *) q→id + 3; *pp ∧ *pp ≡ *qq;
        pp++, qq++) ;
    if (*pp ∨ *qq) return (int) *pp − (int) *qq;
    if (p→id[0] ≠ q→id[0]) return p→id[0] − q→id[0];
    return p→id[1] − q→id[1];
  }
```

18. The collation order follows a string copied from `CWEAVE`.

```
⟨ Global variables 2 ⟩ +≡
  char collate[102];        ▷ collation order ◁
  char ord[256];        ▷ rank in collation order ◁
```

19. The right brace is placed lowest in collating order, because each key is actually followed by a right brace when we are sorting.

Apology: I haven't had time to update this part of the program to allow 8-bit characters. At present the data is assumed to be 7-bit ASCII, as it was in the early versions of `CWEAVE`.

```
⟨ Initialize the data structures 8 ⟩ +≡
  collate[0] ← 0; strcpy(collate + 1,
      "}␣\1\2\3\4\5\6\7\10\11\12\13\14\15\16\17\20\21\22\23\24\25\26\27\30\31\32\33\34\
      \35\36\37!\42#$%&'()*+,-./:;<=>?@[\\]^`{|~_abcdefghijklmnopqrstuvwxyz0123456789");
  { register int j;
    for (j ← 1; collate[j]; j++) ord[collate[j]] ← j;
    ord[128] ← j;      ▷ this affects the ordering of sentinel.id ◁
    for (j ← 'A'; j ≤ 'Z'; j++) ord[j] ← ord[tolower(j)];
  }
```

20. When two lists are combined, we put the data from the second node before the data from the first node, because we are going to reverse the order when printing. After this procedure has acted, the field $q\rightarrow data.n$ should not be considered an active pointer.

```
⟨ Procedures 5 ⟩ +≡
  collapse(p, q)
      node *p, *q;
  { register node *x;
    for (x ← q→data.n; x→next; x ← x→next) ;
    x→next ← p→data.n; p→data.n ← q→data.n;
  }
```

21. The only remaining trick is to format the underline characters properly, especially in the "custom" format when they must become `x`'s.

```
⟨ Output the main list in suitable TEX format 21 ⟩ ≡
  { register node *x;
    printf("\\input␣cttwinxmac\n");
    for (x ← header.next; x ≠ &header; x ← x→next) {
      printf("\\I"); ⟨ Output x→id in suitable TEX format 22 ⟩;
      ⟨ Output the lines of x→data.n in reverse order 23 ⟩;
    }
    printf("\\fin\n");
  }
```

This code is used in section 13.

22. ⟨ Output $x\rightarrow id$ in suitable TEX format 22 ⟩ ≡

```
{ register char *p ← x→id;
  if (*p ≡ '␣') {
    if (*(p + 1) ≠ '␣') goto unknown;
    goto known;
  }
  if (*p ≠ '\\') goto unknown;
  switch (*(p + 1)) {
  case '\\': case '|': case '.': case '&': case '9': printf("\\%c", *(p + 1)); goto known;
  case '$': printf("$\\");
    for (p += 3; *p ≠ '}'; p++)
      if (*p ≡ '_') putchar('x');
      else putchar(*p);
    putchar('$'); goto done;
  default: goto unknown;
  }
unknown: fprintf(stderr, "twinx:␣`%s'␣has␣unknown␣format!\n", p);
known:
  for (p += 2; *p; p++) {
    if (*p ≡ '_') putchar('\\');
    putchar(*p);
  }
done: ;
}
```

This code is used in section 21.

23. ⟨ Output the lines of $x\rightarrow data.n$ in reverse order 23 ⟩ ≡

```
{ register node *y ← x→data.n, *z ← Λ;
  while (y) { register node *w;
    w ← y→next; y→next ← z; z ← y; y ← w;
  }
  while (z) {
    if (z→id) printf("\\unskip␣{\\sc␣%s}~", z→id);
    fputs(z→data.s, stdout); z ← z→next;
    if (z) putchar('\n');
    else puts(".");
  }
}
```

This code is used in section 21.

24. Index.