REFSORT

March 14, 2024 at 18:52

1. Introduction. This short program sorts the mini-indexes of listings prepared by `CTWILL`.

More precisely, suppose you have said `ctwill foo.w`, getting a file `foo.tex`, and that you've then said `tex foo.tex`, getting files `foo.dvi` and `foo.ref`. If you're happy with `foo.dvi` except for the alphabetic order of the mini-indexes, you can then say

```
refsort <foo.ref >foo.sref
```

after which `tex foo` will produce `foo.dvi` again, this time with the mini-indexes in order.

Still more precisely, this program reads from standard input a file consisting of groups of unsorted lines and writes to standard output a file consisting of groups of sorted lines. Each input group begins with an identification line whose first character is `!`; the remaining characters are a page number. The other lines in the group all have the form

$$\texttt{+}␣\alpha␣\texttt{\?\{}\kappa\texttt{\}}\omega$$

where α is a string containing no spaces, `?` is a single character, κ is a string of letters, digits, and `\_`'s, and ω is an arbitrary string. The output groups contain the same lines without the initial `+␣`, sorted alphabetically with respect to the κ fields, followed by a closing line that says '`\donewithpage`' followed by the page number copied from the original identification line.

Exception: In the case of a "custom" identifier, `\?{`κ`}` takes the alternative form `$\`$\kappa␣$`$` instead.

We define limits on the number and size of mini-index entries that should be plenty big enough.

```
#define max_key 30      ▷ greater than the length of the longest identifier ◁
#define max_size 100    ▷ greater than the length of the longest mini-index entry ◁
#define max_items 300   ▷ the maximum number of items in a single mini-index ◁
```

2. Here's the layout of the C program:

```
#define abort(c, m)
        {
          fprintf(stderr, "%s!\n%s", m, buf); return c;
        }
#include "stdio.h"
#include "strings.h"
#include "ctype.h"
  typedef struct {
    char key[max_key];
    char entry[max_size];
  } item;
  item items[max_items];      ▷ all items of current group ◁
  item *sorted[max_items];      ▷ pointers to items in alphabetic order ◁
  char cur_page[10];      ▷ page number, as a string ◁
  char buf[max_size];      ▷ current line of input ◁
  char *input_status;      ▷ Λ if end of input reached, else buf ◁
  main()
  {
    register char *p, *q;
    register int n;      ▷ current number of items ◁
    register item *x, **y;
    input_status ← fgets(buf, max_size, stdin);
    while (input_status) {
      ⟨ Check that buf contains a valid page-number line 3 ⟩;
      ⟨ Read and sort additional lines, until buf terminates a group 4 ⟩;
      ⟨ Output the current group 5 ⟩;
    }
    return 0;      ▷ normal exit ◁
  }
```

3. ⟨ Check that *buf* contains a valid page-number line 3 ⟩ ≡

```
  if (*buf ≠ '!') abort(−1, "missing '!'");
  if (strlen(buf + 1) > 11) abort(−2, "page number too long");
  for (p ← buf + 1, q ← cur_page; *p ≠ '\n'; p++) *q++ ← *p;
  *q ← '\0';
```

This code is used in section 2.

4. ⟨ Read and sort additional lines, until *buf* terminates a group 4 ⟩ ≡

```
  n ← 0;
  while (1) {
    input_status ← fgets(buf, max_size, stdin);
    if (input_status ≡ Λ ∨ *buf ≠ '+') break;
    x ← &items[n]; ⟨ Copy buf to item x 6 ⟩;
    ⟨ Sort the new item into its proper place 8 ⟩;
    if (++n > max_items) abort(−11, "too many lines in group");
  }
```

This code is used in section 2.

5. ⟨ Output the current group 5 ⟩ ≡

```
{
  register int k;
  for (y ← sorted; y < sorted + n; y++) printf("%s\n", (*y)→entry);
  printf("\\donewithpage%s\n", cur_page);
}
```

This code is used in section 2.

6. Sorting. We convert the key to lowercase as we copy it, and we omit backslashes. We also convert `_` to ␣. Then `\_` will be alphabetically less than alphabetic letters, as desired.

```
⟨ Copy buf to item x 6 ⟩ ≡
  if (*(buf + 1) ≠ '␣') abort(−3, "missing␣blank␣after␣+");
  ⟨ Scan past α 9 ⟩;
  if (*p ≠ '␣') abort(−4, "missing␣blank␣after␣alpha");
  if (*(p + 1) ≡ '$') ⟨ Process a custom-formatted identifier 7 ⟩
  else {
    if (*(p + 1) ≠ '\\') abort(−5, "missing␣backslash");
    if (¬*(p + 2)) abort(−6, "missing␣control␣code");
    if (*(p + 3) ≠ '{') abort(−7, "missing␣left␣brace");
    for (p += 4, q ← x→key; *p ≠ '}' ∧ *p; p++) {
      if (*p ≠ '\\') {
        if (isupper(*p)) *q++ ← *p + ('a' − 'A');
        else if (*p ≡ '_') *q++ ← '␣';
        else *q++ ← *p;
      }
    }
    if (*p ≠ '}') abort(−8, "missing␣right␣brace");
  }
  if (q ≥ &x→key[max_key]) abort(−9, "key␣too␣long");
  *q ← '\0'; ⟨ Copy the buffer to x→entry 10 ⟩;
  if (p ≡ buf + max_size − 1) abort(−10, "entry␣too␣long");
  *(q − 1) ← '\0';
```

This code is used in section 4.

7.

```
⟨ Process a custom-formatted identifier 7 ⟩ ≡
  {
    if (*(p + 2) ≠ '\\') abort(−11, "missing␣custom␣backslash");
    for (p += 3, q ← x→key; *p ≠ '␣' ∧ *p; p++) {
      if (isupper(*p)) *q++ ← *p + ('a' − 'A');
      else *q++ ← *p;
    }
    if (*p ≠ '␣') abort(−12, "missing␣custom␣space");
    if (*(p + 1) ≠ '$') abort(−13, "missing␣custom␣dollarsign");
  }
```

This code is used in section 6.

8.

```
⟨ Sort the new item into its proper place 8 ⟩ ≡
  for (y ← &sorted[n]; y > &sorted[0] ∧ strcmp((*(y − 1))→key, x→key) > 0; y−−) *y ← *(y − 1);
  *y ← x;
```

This code is used in section 4.

9. A bugfix. The program specification had a subtle bug: There are cases where α includes spaces that should be removed in the output.

These cases occur when a space occurs after an odd number of doublequote characters. Ergo, the following routine replaced a simpler original loop.

```
⟨ Scan past α 9 ⟩ ≡
  {
    register int toggle ← 0;
    for (p ← buf + 2; (*p ≠ '␣' ∨ toggle) ∧ *p; p++)
      if (*p ≡ '"') toggle ⊕= 1;
  }
```

This code is used in section 6.

10. A corresponding change to the copying loop is also needed.

```
⟨ Copy the buffer to x→entry 10 ⟩ ≡
  {
    register int toggle ← 0;
    for (p ← buf + 2, q ← x→entry; (*p ≠ '␣' ∨ toggle) ∧ *p; p++) {
      if (*p ≡ '"') toggle ⊕= 1;
      if (*p ≠ '␣') *q++ ← *p;
    }
    for ( ; *p; p++) *q++ ← *p;
  }
```

This code is used in section 6.