REFSORT

July 9, 2024 at 19:32

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

$$+_{\sqcup}\alpha_{\sqcup}\verb|\?{|\kappa\verb|}|\omega$$

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 `+`$_{\sqcup}$, 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_{\sqcup}$`$` 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 $(y \leftarrow sorted;\ y < sorted + n;\ y{+}{+})\ printf(\texttt{"\%s\textbackslash n"}, (*y)\rightarrow entry);$

    $printf(\texttt{"\textbackslash\textbackslash donewithpage\%s\textbackslash n"}, cur\_page);$

  }

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 $(*(buf + 1) \neq$ '␣'$)$ $abort(-3,$ "missing␣blank␣after␣+"$)$;
  ⟨ Scan past α 9 ⟩;
  if $(*p \neq$ '␣'$)$ $abort(-4,$ "missing␣blank␣after␣alpha"$)$;
  if $(*(p + 1) \equiv$ '$'$)$ ⟨ Process a custom-formatted identifier 7 ⟩
  else {
    if $(*(p + 1) \neq$ '\\'$)$ $abort(-5,$ "missing␣backslash"$)$;
    if $(\neg *(p + 2))$ $abort(-6,$ "missing␣control␣code"$)$;
    if $(*(p + 3) \neq$ '{'$)$ $abort(-7,$ "missing␣left␣brace"$)$;
    for $(p \mathrel{+}= 4, q \leftarrow x\text{→}key;\ *p \neq$ '}' $\wedge *p;\ p\text{++})$ {
      if $(*p \neq$ '\\'$)$ {
        if $(isupper(*p))$ $*q\text{++} \leftarrow *p + ($'a' $-$ 'A'$)$;
        else if $(*p \equiv$ '_'$)$ $*q\text{++} \leftarrow$ '␣';
        else $*q\text{++} \leftarrow *p$;
      }
    }
    if $(*p \neq$ '}'$)$ $abort(-8,$ "missing␣right␣brace"$)$;
  }
  if $(q \geq \&x\text{→}key[max\_key])$ $abort(-9,$ "key␣too␣long"$)$;
  $*q \leftarrow$ '\0'; ⟨ Copy the buffer to $x\text{→}entry$ 10 ⟩;
  if $(p \equiv buf + max\_size - 1)$ $abort(-10,$ "entry␣too␣long"$)$;
  $*(q - 1) \leftarrow$ '\0';

This code is used in section 4.

7.   ⟨ Process a custom-formatted identifier 7 ⟩ ≡
  {
    if $(*(p + 2) \neq$ '\\'$)$ $abort(-11,$ "missing␣custom␣backlash"$)$;
    for $(p \mathrel{+}= 3, q \leftarrow x\text{→}key;\ *p \neq$ '␣' $\wedge *p;\ p\text{++})$ {
      if $(isupper(*p))$ $*q\text{++} \leftarrow *p + ($'a' $-$ 'A'$)$;
      else $*q\text{++} \leftarrow *p$;
    }
    if $(*p \neq$ '␣'$)$ $abort(-12,$ "missing␣custom␣space"$)$;
    if $(*(p + 1) \neq$ '$'$)$ $abort(-13,$ "missing␣custom␣dollarsign"$)$;
  }

This code is used in section 6.

8.   ⟨ Sort the new item into its proper place 8 ⟩ ≡
  for $(y \leftarrow \&sorted[n];\ y > \&sorted[0] \wedge strcmp((*(y - 1))\text{→}key, x\text{→}key) > 0;\ y\text{--})$ $*y \leftarrow *(y - 1)$;
  $*y \leftarrow x$;

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.

$\langle$ Scan past α 9 $\rangle \equiv$
```
{
    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.

$\langle$ Copy the buffer to $x{\rightarrow}entry$ 10 $\rangle \equiv$
```
{
    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.

abort: 2, 3, 4, 6, 7.
buf: 2, 3, 4, 6, 9, 10.
cur_page: 2, 3, 5.
entry: 2, 5, 10.
fgets: 2, 4.
fprintf: 2.
input_status: 2, 4.
isupper: 6, 7.
item: 2.
items: 2, 4.
k: 5.
key: 2, 6, 7, 8.
main: 2.
max_items: 1, 2, 4.
max_key: 1, 2, 6.
max_size: 1, 2, 4, 6.
n: 2.
p: 2.
printf: 5.
q: 2.
sorted: 2, 5, 8.
stderr: 2.
stdin: 2, 4.
strcmp: 8.
strlen: 3.
toggle: 9, 10.
x: 2.
y: 2.

⟨ Check that *buf* contains a valid page-number line 3 ⟩   Used in section 2.

⟨ Copy the buffer to *x→entry* 10 ⟩   Used in section 6.

⟨ Copy *buf* to item *x* 6 ⟩   Used in section 4.

⟨ Output the current group 5 ⟩   Used in section 2.

⟨ Process a custom-formatted identifier 7 ⟩   Used in section 6.

⟨ Read and sort additional lines, until *buf* terminates a group 4 ⟩   Used in section 2.

⟨ Scan past α 9 ⟩   Used in section 6.

⟨ Sort the new item into its proper place 8 ⟩   Used in section 4.