REFSORT

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

$$+_{\sqcup}\alpha_{\sqcup}\backslash?\{\kappa\}\omega$$

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 $abort(c, m)$
          {
               $fprintf(stderr, \texttt{"\%s!\textbackslash n\%s"}, m, buf)$; **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 $*x, **y$;

    $input\_status \leftarrow fgets(buf, max\_size, stdin)$;
    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 $(*buf \neq \texttt{'!'})$ $abort(-1, \texttt{"missing\textsubscript{⊔}'!'"})$;
  if $(strlen(buf + 1) > 11)$ $abort(-2, \texttt{"page\textsubscript{⊔}number\textsubscript{⊔}too\textsubscript{⊔}long"})$;
  for $(p \leftarrow buf + 1, q \leftarrow cur\_page; *p \neq \texttt{'\textbackslash n'}; p{+}{+})$ $*q{+}{+} \leftarrow *p$;
  $*q \leftarrow \texttt{'\textbackslash 0'}$;
This code is used in section 2.

4.   ⟨Read and sort additional lines, until buf terminates a group 4⟩ ≡
  $n \leftarrow 0$;
  while (1) {
    $input\_status \leftarrow fgets(buf, max\_size, stdin)$;
    if $(input\_status \equiv \Lambda \lor *buf \neq \texttt{'+'})$ **break**;
    $x \leftarrow \&items[n]$; ⟨Copy buf to item x 6⟩;
    ⟨Sort the new item into its proper place 8⟩;
    if $({+}{+}n > max\_items)$ $abort(-11, \texttt{"too\textsubscript{⊔}many\textsubscript{⊔}lines\textsubscript{⊔}in\textsubscript{⊔}group"})$;
  }
This code is used in section 2.

5. ⟨Output the current group 5⟩ ≡

  {

    register int k;

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

    $printf($`"\\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 $\sqcup$. Then \\$_$ will be alphabetically less than alphabetic letters, as desired.

⟨ Copy *buf* to item x 6 ⟩ ≡
  if ($*(buf + 1) \neq$ '$\sqcup$') *abort*(-3, "missing␣blank␣after␣+");
  ⟨ Scan past α 9 ⟩;
  if ($*p \neq$ '$\sqcup$') *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 ($*p \equiv$ '_') $*q\text{++} \leftarrow$ '$\sqcup$';
        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 ($p \mathrel{+}= 3, q \leftarrow x\text{-}key$; $*p \neq$ '$\sqcup$' $\wedge *p$; $p\text{++}$) {
      if (*isupper*($*p$)) $*q\text{++} \leftarrow *p +$ ('a' $-$ 'A');
      else $*q\text{++} \leftarrow *p$;
    }
    if ($*p \neq$ '$\sqcup$') *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.

⟨ Scan past α 9 ⟩ ≡
   {
      register int $toggle \leftarrow 0$;
      for $(p \leftarrow buf + 2;\ (*p \neq\ '\textvisiblespace'\ \vee toggle) \wedge *p;\ p{+}{+})$
         if $(*p \equiv\ '"')\ toggle \oplus{=} 1$;
   }

This code is used in section 6.

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

⟨ Copy the buffer to $x{\rightarrow}entry$ 10 ⟩ ≡
   {
      register int $toggle \leftarrow 0$;
      for $(p \leftarrow buf + 2, q \leftarrow x{\rightarrow}entry;\ (*p \neq\ '\textvisiblespace'\ \vee toggle) \wedge *p;\ p{+}{+})$ {
         if $(*p \equiv\ '"')\ toggle \oplus{=} 1$;
         if $(*p \neq\ '\textvisiblespace')\ *q{+}{+} \leftarrow *p$;
      }
      for $(\ ;\ *p;\ p{+}{+})\ *q{+}{+} \leftarrow *p$;
   }

This code is used in section 6.

⟨ 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.