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

   ```latex
   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

   ```latex
   +\alpha\gamma\kappa\omega
   ```

   where `\alpha` is a string containing no spaces, `\gamma` is a single character, `\kappa` is a string of letters, digits, and `_`’s, and `\omega` is an arbitrary string. The output groups contain the same lines without the initial `+\alpha`, sorted alphabetically with respect to the `\kappa` 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, `\gamma{\kappa}` takes the alternative form $\kappa$ instead.

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

   ```latex
   #define max_key 30 \triangleright greater than the length of the longest identifier \triangleright
   #define max_size 100 \triangleright greater than the length of the longest mini-index entry \triangleright
   #define max_items 300 \triangleright the maximum number of items in a single mini-index \triangleright
   ```
2. Here's the layout of the C program:

```c
#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 ≠ '
'; 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 \( k \))

\[
\begin{align*}
&\text{register int } k; \\
&\text{for } (y ← \text{sorted}; y < \text{sorted} + n; y++) \text{ printf("%s\n", (*y)-entry);} \\
&\text{printf("\donewithpage%s\n", cur_page);}
\end{align*}
\]

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) \equiv
  if (*buf + 1) \neq 'U') abort(-3,"missing,blank,afterĽ,star");
  (Scan past \a 9);
  if (*p \neq 'U') abort(-4,"missing,blank,afterĽ,star_alpha");
  if (*p + 1) \equiv 'S') { (Process a custom-formatted identifier 7) }
  else {
    if (*p + 1) \neq 'L') abort(-5,"missing,backslash");
    if (*p + 2) abort(-6,"missing,control_code");
    for (p += 4, q \leftarrow xキー; *p \neq 'Ъ' \wedge *p; p++) {
      if (*p \neq 'L') {
        if (isupper(*p)) *q++ \leftarrow *p + ('a' - 'A');
        else *q++ \leftarrow *p;
      }
    }
    if (*p \neq 'Ъ') abort(-8,"missing,right_brace");
  }
  if (q \geq &xキー[max_key]) abort(-9,"キー,too,long");
  *q \leftarrow 'L0';  (Copy the buffer to x-entry 10);
  if (p \equiv buf + max_size - 1) abort(-10,"entry,too,long");
  *(q - 1) \leftarrow 'L0';
This code is used in section 4.

7. (Process a custom-formatted identifier 7) \equiv
  {
    if (*p + 2) \neq 'L') abort(-11,"missing,custom,backslash");
    for (p += 3, q \leftarrow xキー; *p \neq 'Ъ' \wedge *p; p++) {
      if (isupper(*p)) *q++ \leftarrow *p + ('a' - 'A');
      else *q++ \leftarrow *p;
    }
    if (*p \neq 'Ъ') 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) \equiv
  for (y \leftarrow &sorted[n]; y \&& sorted[0] \wedge strcmp(*y-1),xキー) > 0; y--; y) *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 $\alpha$ 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 $\alpha$ 9) \equiv

\begin{verbatim}
{ 
    register int toggle ← 0;
    for (p ← buf + 2; (*p ≠ ' ') ∨ toggle) ∧ *p;  p++)
        if (*p ≡ '"') toggle ⊕ 1;
}
\end{verbatim}

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) \equiv

\begin{verbatim}
{ 
    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;
}
\end{verbatim}

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 $\alpha$ 9) Used in section 6.
(Sort the new item into its proper place 8) Used in section 4.