<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Sorting</td>
<td>4</td>
</tr>
<tr>
<td>A bugfix</td>
<td>5</td>
</tr>
</tbody>
</table>
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

```
+\alpha\?\{\kappa}\omega
```

where \(\alpha\) is a string containing no spaces, \(?\) 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 `+\_\_]`, sorted alphabetically with respect to the \(\kappa\) fields, followed by a closing line that says ‘\`doneywithpage’ followed by the page number copied from the original identification line.

Exception: In the case of a “custom” identifier, \(\?\{\kappa}\) takes the alternative form `$\kappa$_$` instead.

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

```
#define max_key 30 \(\triangleright\) greater than the length of the longest identifier \(<\)
#define max_size 100 \(\triangleright\) greater than the length of the longest mini-index entry \(<\)
#define max_items 300 \(\triangleright\) the maximum number of items in a single mini-index \(<\)
```
\section{Introduction}

Here's the layout of the C program:

```c
#define abort(c, m)
    { fprintf(stderr, "%s!\n%a", 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⟩
        ⟨Read and sort additional lines, until buf terminates a group⟩
        ⟨Output the current group⟩
        return 0;        ▶ normal exit
    }
}
```

This code is used in section 2.

\section{Check that buf contains a valid page-number line}

```c
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.

\section{Read and sort additional lines, until buf terminates a group}

```c
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⟩
    ⟨Sort the new item into its proper place⟩
    if (++n > max_items) abort(-11,"too many lines in group");
}
```

This code is used in section 2.
5. (Output the current group $\delta$) \equiv \\
\{ \\
\text{register int } k; \\
\text{for (} y \leftarrow \text{sorted}; y < \text{sorted + } n; y++ \text{) printf(“%s
”, (*y)-entry); \\
printf(“\done\with\page\%s
”, 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⟩ ≡
  if \((\* (\buf + 1) \neq \'\_')\) abort\((-3, "missing\_blank\_after\_\_\+")\);
  ⟨Scan past \α⟩
  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 \(p += 4, q \leftarrow x\_key; \* p \neq \'\}\) \& \*p; \(p++\) \{ 
      if \((\* p \neq \'\\\')\) {
        if \((\isupper(\* p))\) \*q++ ← \*p + (\'a' − \'A');
        else if \((\* p \equiv \'\_\')\) \*q++ ← \'\_';
        else \*q++ ← \*p;
      }
    }
    if \((\* p \neq \'}\')\) abort\((-8,"missing\_right\_brace")\);
  }

if \((q \geq &x\_key[\max\_key])\) abort\((-9, "key\_too\_long")\);
\*q ← \'\0'; ⟨Copy the buffer to x\_entry 10⟩;
if \((p \equiv \buf + \max\_size - 1)\) abort\((-10,"entry\_too\_long")\);
*(q - 1) ← \'\0';

This code is used in section 4.

7. ⟨Process a custom-formatted identifier 7⟩ ≡
  {
    if \((\* (p + 2) \neq \'\\\')\) abort\((-11,"missing\_custom\_backslash")\);
    for \(p += 3, q \leftarrow x\_key; \* p \neq \'\_\' \& \*p; \(p++\) \{ 
      if \((\isupper(\* p))\) \*q++ ← \*p + (\'a' − \'A');
      else \*q++ ← \*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 \((y \leftarrow &\_sorted[n]; y > &\_sorted[0] \& \strmp((\* (y - 1))-key, x\_key) > 0; y --)\) \*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 $\alpha$ 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.

\[
\text{Scan past $\alpha$ 9} \equiv \\
\begin{array}{l}
\text{register int toggle } \leftarrow 0; \\
\text{for } (p \leftarrow \text{buf} + 2; (*p \neq ' ' \lor \text{toggle}) \land *p; p++) \\
\quad \text{if } (*p \equiv ',' ) \text{ toggle } \oplus 1;
\end{array}
\]

This code is used in section 6.

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

\[
\text{Copy the buffer to } x\text{-entry 10} \equiv \\
\begin{array}{l}
\text{register int toggle } \leftarrow 0; \\
\text{for } (p \leftarrow \text{buf} + 2, q \leftarrow x\text{-entry}; (*p \neq ' ' \lor \text{toggle}) \land *p; p++) \\
\quad \text{if } (*p \equiv ',' ) \text{ toggle } \oplus 1; \\
\quad \text{if } (*p \neq ' ' ) *q++ \leftarrow *p;
\end{array}
\]

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 \textit{buf} contains a valid page-number line 3) Used in section 2.
(Copy the buffer to \textit{x}-entry 10) Used in section 6.
(Copy \textit{buf} to item \textit{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 \textit{buf} terminates a group 4) Used in section 2.
(Scan past \textit{α} 9) Used in section 6.
(Sort the new item into its proper place 8) Used in section 4.