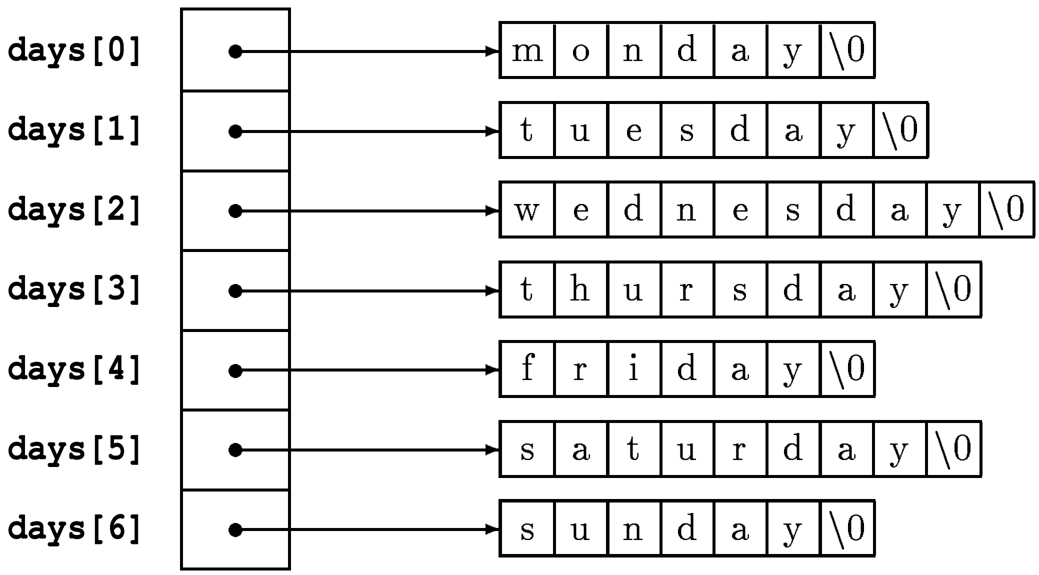
CHAPTER 13

ARRAYS OF POINTERS

STORING STRINGS

day_table

m	o	n	d	a	y	\0			
t	u	e	s	d	a	y	\0		
w	e	d	n	e	s	d	a	y	\0
t	h	u	r	s	d	a	y	\0	
f	r	i	d	a	y	\0			
s	a	t	u	r	d	a	y	\0	
s	u	n	d	a	y	\0			



prdays.c

(Figure 13.2 – Page 313)

```
/*
 * Main program that prints a table of days.
 */
#include <stdio.h>

char *days[] =
{
  "monday",  "tuesday", "wednesday", "thursday",
  "friday", "saturday", "sunday"
};

int main()
{
  void print_strings(char *table[], int n);

  print_strings(days, sizeof(days)/sizeof(char *));
  return 0;
}
```

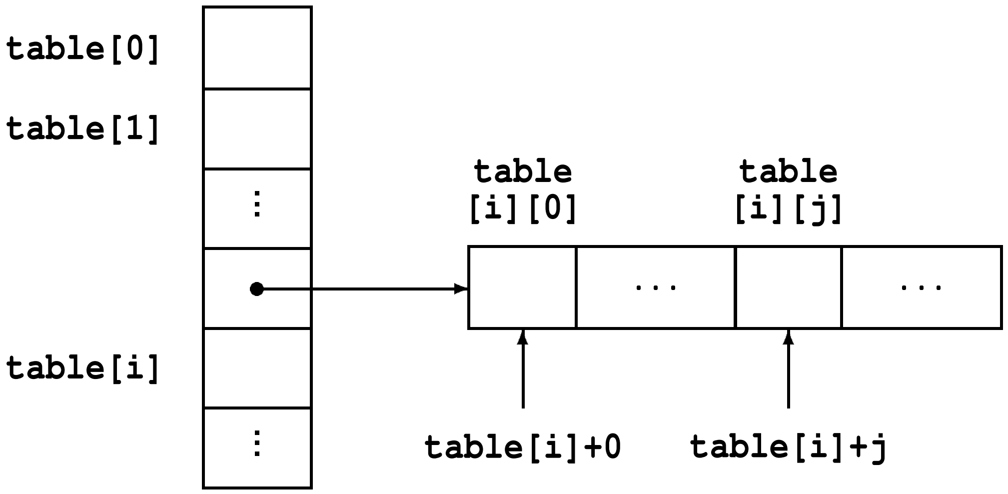
prstr1.c
(Figure 13.3 – Page 313)

```
/*
 * Print a table of strings, one per line.
 */
#include <stdio.h>

void print_strings(char *table[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    printf("%s\n", table[i]);
}
```

prstr2.c
(Figure 13.4 – Page 314)

```
/*
 *  Print a table of strings, one per line, using putchar.
 */
#include <stdio.h>

void print_strings(char *table[], int n)
{
  int i, j;

  for (i = 0; i < n; i++)
  {
    for (j = 0; table[i][j] != '\0'; j++)
      putchar(table[i][j]);
    putchar('\n');
  }
}
```

ACCESSING CHARACTERS



prstr3.c
(Figure 13.6 – Page 315)

```
/*
 * Print table of strings using pointers to traverse the string.
 */
#include <stdio.h>

void print_strings(char *table[], int n)
{
  int i;
  char *ptr;

  for (i = 0; i < n; i++)
  {
    for (ptr = table[i]; *ptr != '\0'; ptr++)
      putchar(*ptr);
    putchar('\n');
  }
}
```

revinp.c

(Figure 13.7 – Page 316)

```
/*
 * Reverse input, one line at a time.
 */
#include <stdio.h>
#include <stddef.h>

#define  MAXLINES     100          /* lines to store */

int main()
{
  void print_rev_strings(char *table[], int n);
  int  read_strings(char *table[], int max);
  int  lines;                      /* input lines read */
  char *strings[MAXLINES];         /* array of pointers to strings */

  if ((lines = read_strings(strings, MAXLINES)) < 0)
    printf("Out of memory after reading %i lines\n", -lines);
  else
  {
    if (lines >= MAXLINES)
      printf("Only %i out of %i lines stored\n", MAXLINES, lines);
    print_rev_strings(strings, lines > MAXLINES ? MAXLINES : lines);
  }
  return lines < 0 || lines >= MAXLINES;
}
```

strtab.c

(Figure 13.8 – Page 317)

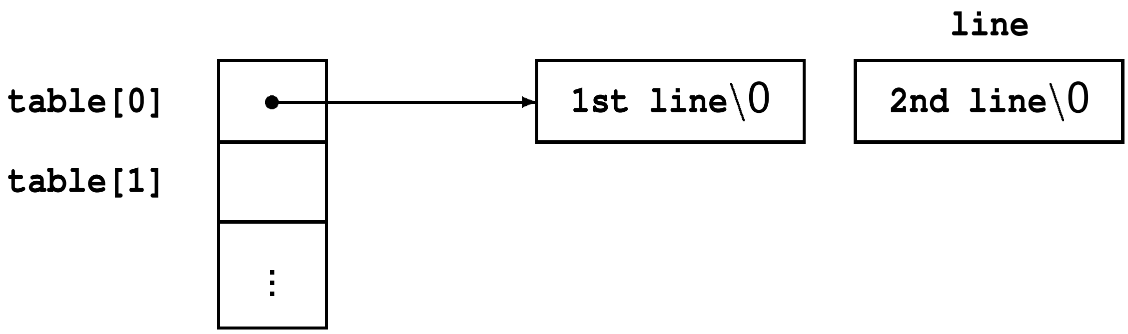
```
/*
 * Functions to read and print string table.
 *   read_strings - read strings into table, dynamically allocating
 *     space for each string.  it indicates an out of memory error
 *     by returning the negative number of lines it successfully read.
 *   print_rev_strings - print table in reverse order.
 *   makedupstr - duplicate a single string.
 */
#include <stdio.h>
#include <string.h>
#include <stddef.h>                 /* for NULL */
#include <stdlib.h>                 /* for malloc */

#define  MAXLEN   80                /* chars per line */

static char *makedupstr(const char *str);

int read_strings(char *table[], int max)
{
  int   getline(char *line, int n);
  int   lines;                      /* lines read */
  char  *sptr;                      /* ptr to space from malloc */
  char  line[MAXLEN + 1];           /* current input line */

  for (lines = 0; getline(line, MAXLEN) != -1 && lines >= 0; lines++)
    if (lines < max)
      if ((sptr = makedupstr(line)) != NULL)
        table[lines] = sptr;
      else
        lines = -lines;             /* indicate out of memory error */
  return lines;
}
```

strtab.c

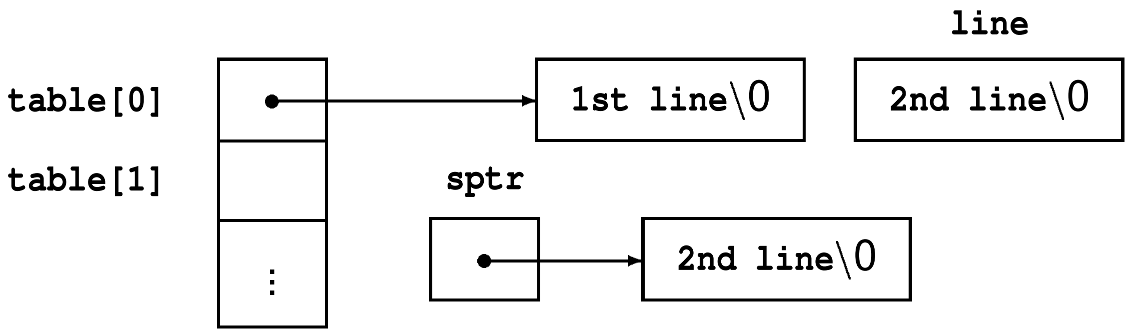
(Figure 13.8 – Page 317)

```
void print_rev_strings(char *table[], int n)
{
  while (--n >= 0)
    printf("%s\n", table[n]);
}

static char *makedupstr(const char *str)
{
  char *newstr = malloc(strlen(str) + 1);

  if (newstr != NULL)
    strcpy(newstr, str);
  return newstr;
}
```

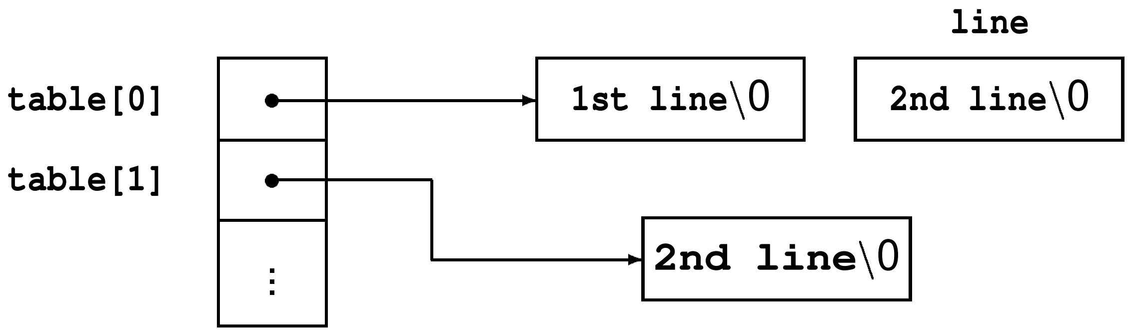
ALLOCATING A STRING TABLE



(a) The array after placing the first input line in it and reading the second input line.



(b) The array after allocating space for the second input line and copying the line into that space.



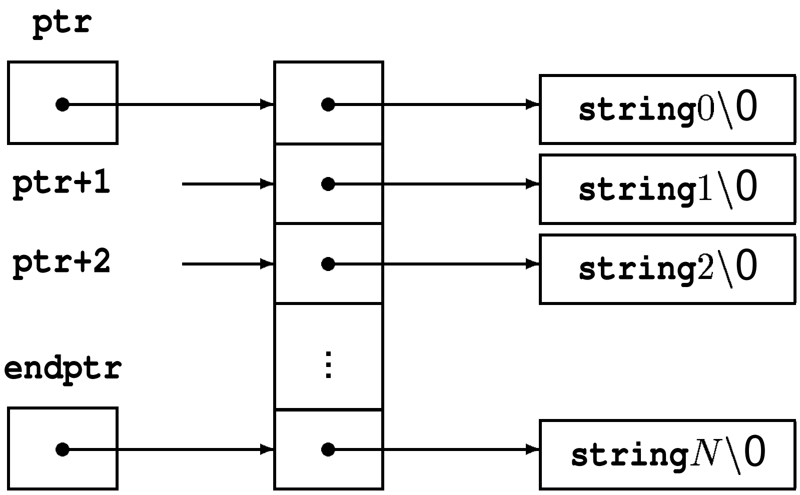
(c) The array after copying the line into that space and hooking the line into the table.

prstr4.c

(Figure 13.10 – Page 319)

```
/*
 * Print a table of strings, one per line.
 */
#include <stdio.h>

void print_strings(char **ptr, int n)
{
  char **endptr = ptr + n;         /* ptr to just past last element */

  for (; ptr < endptr; ptr++)
    printf("%s\n", *ptr);
}
```

POINTERS AND STRING ARRAYS



testsrch.c

(Figure 13.12 – Page 321)

```c
/*
 * Main program to test our string-searching function.
 */
#include <stdio.h>

#define MAXLEN 80

char *days[] =
  {"monday", "tuesday", "wednesday", "thursday",
   "friday", "saturday", "sunday"};
const int numdays = sizeof(days)/sizeof(days[0]);

int main()
{
  int         getline(char *line, int max);
  char        *search_strings(const char *target, char *t[], int n);
  const char *match;
  char        line[MAXLEN + 1];

  while (printf("Enter day of week: "), getline(line, MAXLEN) != -1)
    if ((match = search_strings(line, days, numdays)) != NULL)
      printf("%s is a day of the week\n", match);
    else
      printf("%s is not a day of the week.\n", line);
  return 0;
}
```

srchstr1.c

(Figure 13.13 – Page 321)

```
/*
 * Search table for given string, array-subscripting version.
 */
#include <stddef.h>
#include <string.h>

char *search_strings(const char *string, char *table[], int n)
{
  int i = 0;

  for (; i < n; i++)
    if (strcmp(table[i], string) == 0)
      return table[i];                /* return pointer to match */
  return NULL;                        /* no match */
}
```

srchstr2.c

(Figure 13.14 – Page 322)

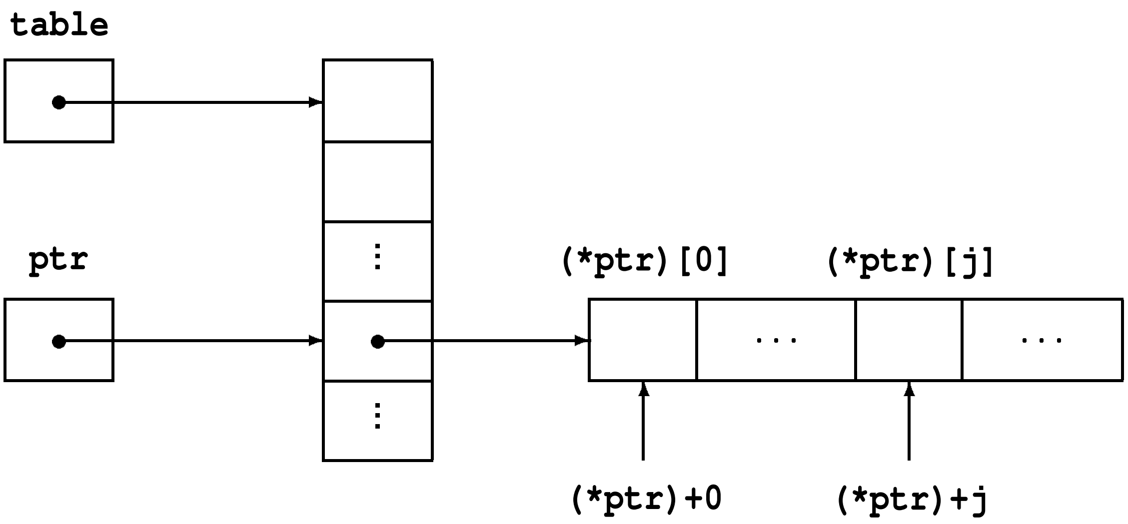
```
/*
 * Search table for a given string, pointer-indexing version.
 */
#include <stddef.h>
#include <string.h>

char *search_strings(const char *string, char **ptr, int n)
{
  char **const endptr = ptr + n;

  for (; ptr < endptr; ptr++)
    if (strcmp(*ptr, string) == 0)
      return *ptr;                     /* return pointer to match */
  return NULL;                         /* no match */
}
```

prstr5.c

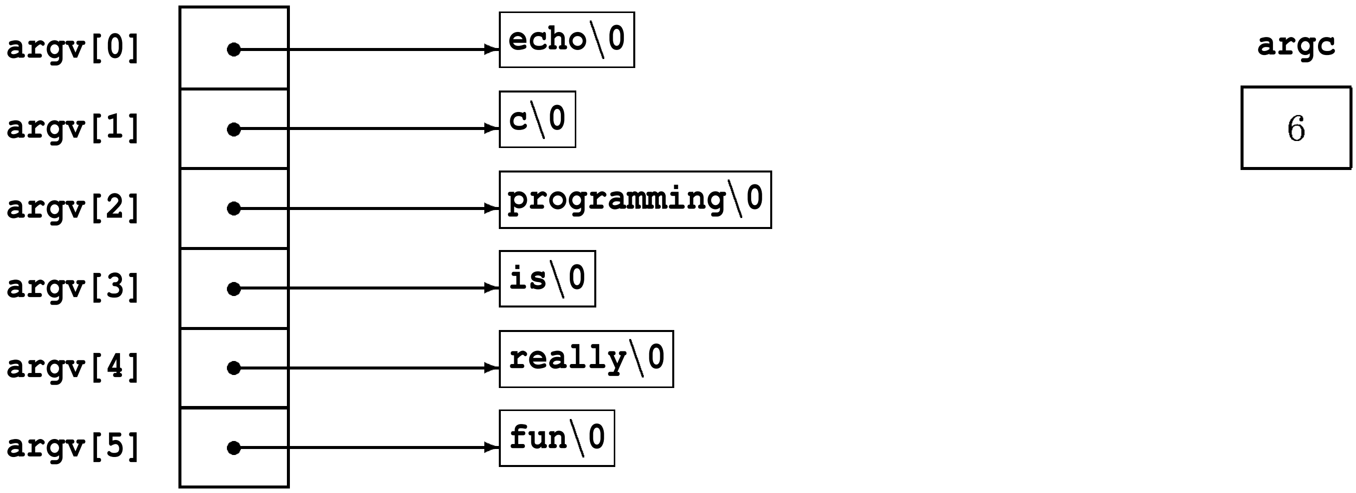
(Figure 13.15 – Page 322)

```
/*
 * Print a table of strings, one per line, using a pointer to
 * traverse the table.
 */
#include <stdio.h>

void print_strings(char **ptr, int n)
{
  char **endptr = ptr + n;    /* pointer to just past last element */
  int j;                      /* index to individual items */

  for(; ptr < endptr; ptr++)
  {
    for (j = 0; (*ptr)[j] != '\0'; j++)
      putchar((*ptr)[j]);
    putchar('\n');
  }
}
```

POINTER ACCESS



COMMAND-LINE ARGUMENTS



echo.c

(Figure 13.18 – Page 325)

```
/*
 * Echo arguments, using array subscripting.
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
  int next;                              /* index to next argument */

  for (next = 1; next < argc; next++)
    printf("%s%c", argv[next], (next < argc - 1) ? ' ' : '\n');
  return 0;
}
```

echo2.c

(Figure 13.19 – Page 325)

```
/*
 * Echo arguments, using pointer indexing.
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
  char **argptr       = argv + 1;      /* pointer to first argument */
  char **const endptr = argv + argc;   /* pointer to last argument */

  for (; argptr < endptr; argptr++)
    printf("%s%c", *argptr, (argptr < endptr - 1) ? ' ' : '\n');
  return 0;
}
```

ws.h

(Figure 13.20 – Page 326)

```
/*
 * Header file defining useful search constants and types.
 */
#include <stdio.h>
#include <string.h>

#define MAXLEN 80

typedef short BOOLEAN;

#define TRUE  1
#define FALSE 0

extern BOOLEAN PrintNumber;          /* current line number */
extern BOOLEAN PrintNoMatch;         /* print unmatched lines? */
extern BOOLEAN PrintCountOnly;       /* print match count only? */
```

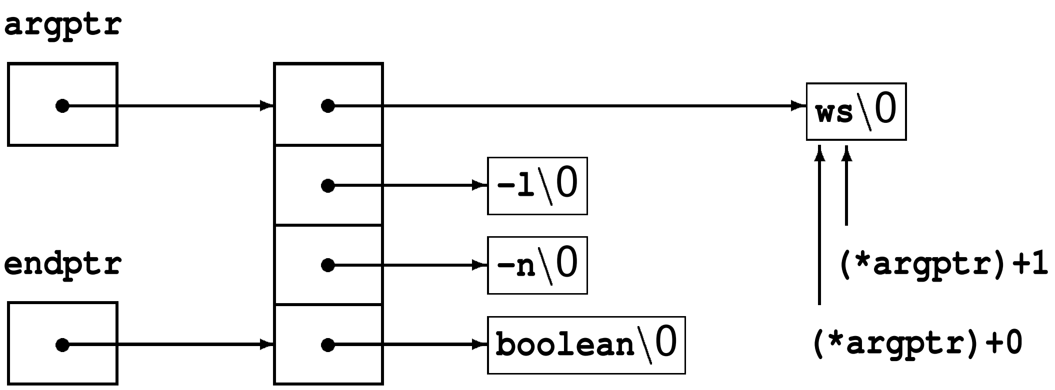
WS.C
(Figure 13.21 – Page 327)

```
/*
 * Search for strings provided in program's arguments.
 */
#include "ws.h"

BOOLEAN PrintNumber;                    /* line number */
BOOLEAN PrintNoMatch;                   /* print doesn't match lines */
BOOLEAN PrintCountOnly;                 /* just print match count */

int main(int argc, char *argv[])
{
  void  search_input(char *strtab[], int n);
  char **argptr          = &argv[1];    /* pointer to next real arg */
  char **const endptr    = &argv[argc]; /* pointer to past last arg */
  BOOLEAN badopt         = FALSE;       /* bad option flag */
  int     status         = 0;           /* return value */

  for (; argptr < endptr && (*argptr)[0] == '-'; argptr++)
    if ((*argptr)[1] == '\0')
      badopt = TRUE;
    else
      switch ((*argptr)[1])
      {                                /* process next option */
        case 'c':
          PrintCountOnly = TRUE;
          break;
        case 'l':
          PrintNumber    = TRUE;
          break;
        case 'n':
          PrintNoMatch   = TRUE;
          break;
        default:
          printf("ws: bad option %c\n", (*argptr)[1]);
          badopt         = TRUE;
      }
```

WS.C
(Figure 13.21 – Page 327)

```
  if (badopt || argptr >= endptr)                /* oops */
  {
    printf("usage: ws [-c][-l][-n] patterns...\n");
    status = 1;
  }
  else
    search_input(argptr, endptr - argptr);
  return status;
}
```

PROCESSING OPTIONAL ARGUMENTS



wssrch.c

(Figure 13.22 – Page 328)

```
/*
 * Do the actual search.
 *   search_input - see if any of the strings passed to it are in the
 *     program's input.
 *   find_match - find out if string is substring of any entry in table.
 *   display_line - write any matching lines.
 */
#include "ws.h"

static BOOLEAN find_match(const char *str, char *tab[], int n);
static void    display_line(const char *line, long lines);

void search_input(char *strptrs[], int n)
{
  int     getline(char *line, int maxlen);
  char    line[MAXLEN + 1];           /* holds next line */
  long    lines;                      /* count lines */
  long    matched_lines = 0L;         /* matched lines */
  BOOLEAN found;

  for (lines = 1; getline(line, MAXLEN) != -1; lines++)
  {
    found = find_match(line, strptrs, n);
    if ((found && !PrintNoMatch) || (!found && PrintNoMatch))
    {
      matched_lines++;
      if (!PrintCountOnly)
        display_line(line, lines);
    }
  }
  if (PrintCountOnly)
    printf("%li\n", matched_lines);
}
```

wssrch.c

(Figure 13.22 – Page 328)

```
static BOOLEAN find_match(const char *str, char *tab[], int n)
{
  int i;

  for (i = 0; i < n; i++)
    if (strstr(str, tab[i]) != NULL)
      return TRUE;
  return FALSE;
}

static void display_line(const char *line, long lines)
{
  if (PrintNumber)
    printf("%li ", lines);
  printf("%s\n", line);
}
```

sort.c

(Figure 13.24 – Page 330)

```
/*
 * String sorting program.
 */
#include "sort.h"

BOOLEAN Reverse;               /* flag: reverse sense of sort */
int     FieldCol = 0;          /* flag: starting column for sort */
int     FieldLen = MAXLEN;     /* flag: length of field to sort */

int main(int argc, char *argv[])
{
  char *strings[MAXLINES];           /* input lines */
  int   lines;                       /* count of input lines */
  int   status = 1;                  /* program return status */

  if (argc >= 1 && handle_options(argv + 1, argv + argc))
    if ((lines = sorted_read_strings(strings, MAXLINES)) < 0)
      printf("Out of memory after %i lines\n\n", -lines);
    else if (lines == MAXLINES)
      printf("More than %i lines of input\n\n", lines);
    else
    {
      display_strings(strings, lines);
      status = 0;
    }
  return status;
}
```

sortopt.c

(Figure 13.25 – Page 331)

```c
/*
 * Handle various options.
 */
#include "sort.h"

int handle_options(const char *const *argptr, const char *const *endptr)
{
  BOOLEAN goodopt = TRUE;           /* flag to indicate bad option */

  for (; argptr < endptr && (*argptr)[0] == '-'; argptr++)
    if ((*argptr)[1] == '\0')
      printf("sort: missing option\n");
    else
      switch ((*argptr)[1])
      {                             /* process next option */
        case 'b':
          if (sscanf(*argptr + 2, "%i", &FieldCol) != 1)
          {
            printf("sort: invalid starting column\n");
            goodopt = FALSE;
          }
          break;
        case 'l':
          if (sscanf(*argptr + 2, "%i", &FieldLen) != 1)
          {
            printf("sort: invalid field length\n");
            goodopt = FALSE;
          }
          break;
        case 'r':
          Reverse = TRUE;           /* ignore anything after option */
          break;
        default:
          printf("sort: bad option %c\n", *argptr[1]);
          goodopt = FALSE;
      }
```

sortopt.c

(Figure 13.25 – Page 331)

```
  if (argptr < endptr)
  {
    printf("sort: extra arguments\n");
    goodopt = FALSE;
  }

  if (!goodopt)
    printf("Usage: sort [-r] [-bNUM] [-eNUM]\n");
  return goodopt;
}
```

sort.h

(Figure 13.26 – Page 332)

```
/*
 * Define useful constants for string searcher.
 */
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>

typedef enum {FALSE, TRUE} BOOLEAN;

#define MAXLINES  1000
#define MAXLEN      80

extern BOOLEAN Reverse;
extern int     FieldCol;
extern int     FieldLen;

extern int  sorted_read_strings(char *table[], int n);
extern void display_strings(const char *const table[], int n);
extern int  handle_options(const char *const *xptr,
                           const char *const *yptr);
extern int  getline(char *line, int n);
```

sortstrs.c

(Figure 13.27 – Page 333)

```
/*
 * Functions to read input strings using insertion sort.
 */
#include "sort.h"

static int linecmp(const char *x, const char *y)
{
  int i;

  for (i = 0; i < FieldCol && *x != '\0' && *y != '\0'; i++, x++, y++)
    ;
  return strncmp(x, y, FieldLen);
}

static void insert(const char *a[], const char *str, int n)
{
  int pos;

  for (pos = n; pos > 0 && linecmp(str, a[pos - 1]) < 0; pos--)
    a[pos] = a[pos - 1];
  a[pos] = str;
}
```

sortstrs.c

(Figure 13.27 – Page 333)

```
int sorted_read_strings(char *table[], int max)
{
  char *sptr;                    /* pointer to malloc-allocated space */
  char line[MAXLEN + 1];         /* current input line */
  int  lines = 0;                /* number of lines read */
  int  len;                      /* length of current line */

  for (; (len = getline(line, MAXLEN)) != -1 && lines >= 0; lines++)
    if (lines < max)
      if ((sptr = malloc(len + 1)) != NULL)
        insert(table, strcpy(sptr,line), lines);
      else
        lines = -lines;          /* indicate out of memory error */
  return lines;
}

void display_strings(const char *const table[], int n)
{
  int i;

  if (!Reverse)
    for (i = 0; i < n; i++)      /* print table in forward order */
      printf("%s\n", table[i]);
  else
    while (--n >= 0)             /* print table in reverse order */
      printf("%s\n",table[n]);
}
```

CHAPTER SUMMARY

- C allows us to have arrays of pointers, which are often called ragged arrays.
- We can initialize ragged arrays at compile time by providing a list of values or at run time by using dynamic allocation.
- Although these ragged arrays are not stored as two-dimensional arrays, we can access their elements using two-dimensional array accessing.
- As with other arrays, we can use either array subscripting or pointer indexing to traverse ragged arrays.
- The arguments we type on the command line invoking the program are placed into an array of pointers **`argv`**. The number of arguments is placed into an integer **`argc`**.