CHAPTER 14

FUNCTIONS REVISITED

14.1 SIMULATING CALL BY REFERENCE
- The **`swap`** Function
- Some Common Mistakes
- Avoiding These Common Mistakes

14.2 OLD-STYLE DECLARATIONS AND DEFINITIONS
- Type Checking Problems
- Automatic Conversions
- Parameters and Portability
- Mixing Styles
- Prototypes versus Old-Style Declarations

14.3 PASSING VARIABLE NUMBERS OF ARGUMENTS
- Dealing with Differing Argument Types
- Built-in Variable Argument Functions

14.4 POINTERS TO FUNCTIONS

14.5 RECURSION

14.6 CASE STUDY—BINARY SEARCH

Sample Output – badswap

```
Before swap, s=10, t=20
After swap, s=10, t=20
```

badswap.c

(Figure 14.1 – Page 338)

```c
/*
 * A program that doesn't exchange two values.
 */
#include <stdio.h>

int main()
{
  void swap(int x, int y);
  int s, t;

  s = 10; t = 20;
  printf("Before swap, s=%i, t=%i\n", s, t);
  swap(s, t);
  printf("After swap, s=%i, t=%i\n", s, t);
  return 0;
}

/* Incorrectly exchanges only values of its parameters */

void swap(int x, int y)
{
  int temp;

  temp = x;  x = y;  y = temp;
}
```

Sample Output – swap

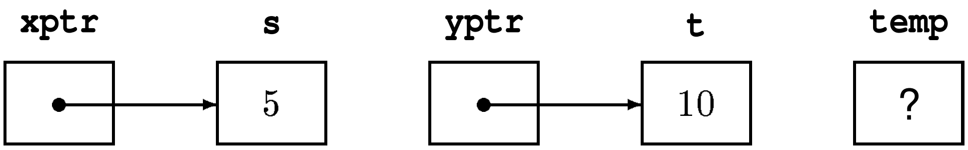
```
Before swap, s=10, t=20
After swap, s=20, t=10
```

swap.c

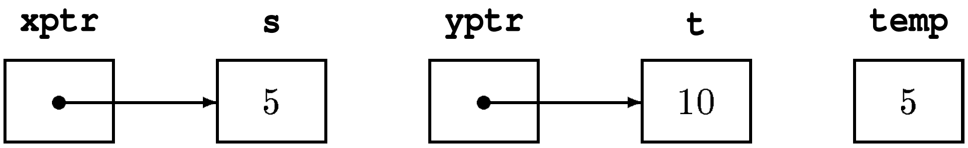
(Figure 14.2 – Page 339)

```
/*
 * A program that does exchange two values.
 */
#include <stdio.h>

int main()
{
  void swap(int *xptr, int *yptr);
  int s, t;

  s = 10; t = 20;
  printf("Before swap, s=%i, t=%i\n", s, t);
  swap(&s, &t);
  printf("After swap, s=%i, t=%i\n", s, t);
  return 0;
}

/* Correctly exchanges a pair of values in caller */

void swap(int *xptr, int *yptr)
{
  int temp;

  temp = *xptr;  *xptr = *yptr;  *yptr = temp;
}
```

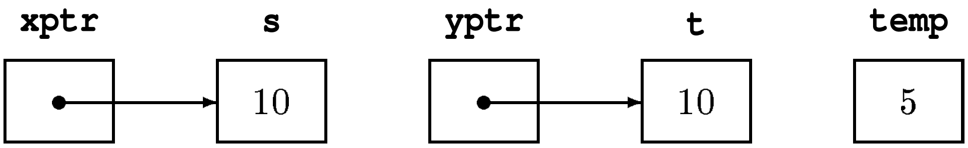
EXCHANGING VALUES THROUGH POINTERS



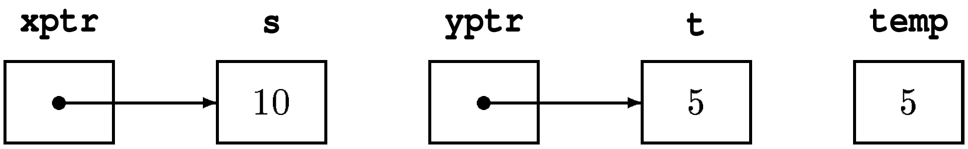
(a) When swap is called



(b) `temp = *xptr`



(c) `*xptr = *yptr`



(d) `*yptr = temp`

oops.c

(Figure 14.4 – Page 341)

```
/*
 * Reverse a table, using "swap" incorrectly.
 */
#include <stdio.h>

int main()
{
  void      table_reverse(int *ptr, int n);
  int       table[] = {56, 84, 51, 78, 11, 96, 38, 24, 35, 9};
  const int entries = sizeof(table)/sizeof(table[0]);
  int       i;

  table_reverse(table, entries);
  for (i = 0; i < entries; i++)               /* dump table */
    printf("%i\n", table[i]);
  return 0;
}

void table_reverse(int *ptr, int n)
{
  void swap(int *xptr, int *yptr);
  int *lastptr = ptr + n - 1;

  for (; lastptr > ptr; lastptr--, ptr++)
    swap(&ptr, &lastptr);                     /* wrong! */
}

void swap(int *xptr, int *yptr)
{
  int temp;

  temp = *xptr, *xptr = *yptr, *yptr = temp;
}
```

minmax.c

(Figure 14.5 – Page 342)

```c
/*
 * Return largest and smallest array values through pointer parameters.
 */
#include <stdio.h>
#include <limits.h>

int main()
{
  void      minmax(const int table[], int n, int *minptr, int *maxptr);
  const int table[] = {10, 56, 79, -21, 51, 5, 64, 70, 0, -10};
  const int entries = sizeof(table)/sizeof(table[0]);
  int       min, max;

  minmax(table, entries, &min, &max);
  printf("min=%i, max=%i\n", min, max);
  return 0;
}

void minmax(const int table[], int n, int *minptr, int *maxptr)
{
  int i;
  int min = INT_MAX;
  int max = INT_MIN;

  for (i = 0; i < n; i++)
    if (table[i] < min)
      min = table[i];
    else if (table[i] > max)
      max = table[i];
  *minptr = min;
  *maxptr = max;
}
```

minmax2.c

(Figure 14.6 – Page 343)

```
/*
 * Return largest and smallest array values thru another array parameter.
 */
#include <stdio.h>
#include <limits.h>

enum index {MIN, MAX};

int main()
{
  void      minmax(const int table[], int n, int results[]);
  const int table[] = {10, 56, 79, -21, 51, 5, 64, 70, 0, -10};
  const int entries = sizeof(table)/sizeof(table[0]);
  int       results[2];

  minmax(table, entries, results);
  printf("min=%i, max=%i\n", results[MIN], results[MAX]);
  return 0;
}

void minmax(const int table[], int n, int results[])
{
  int i;
  int min = INT_MAX;
  int max = INT_MIN;

  for (i = 0; i < n; i++)
    if (table[i] < min)
      min = table[i];
    else if (table[i] > max)
      max = table[i];
  results[MIN] = min;
  results[MAX] = max;
}
```

minmax3.c

(Figure 14.7 – Page 344)

```
/*
 * Return largest and smallest array values through a pointer
 * to an internal static array.
 */
#include <stdio.h>
#include <limits.h>

enum index {MIN, MAX};

int main()
{
  int       *minmax(const int table[], int n);
  const int table[] = {10, 56, 79, -21, 51, 5, 64, 70, 0, -10};
  const int entries = sizeof(table)/sizeof(table[0]);
  int       *resptr = minmax(table, entries);

  printf("min=%i, max=%i\n", resptr[MIN], resptr[MAX]);
  return 0;
}

int *minmax(const int table[], int n)
{
  static int results[2];
  int        i;
  int        min = INT_MAX;
  int        max = INT_MIN;

  for (i = 0; i < n; i++)
    if (table[i] < min)
      min = table[i];
    else if (table[i] > max)
      max = table[i];
  results[MIN] = min;
  results[MAX] = max;
  return results;
}
```

inrange.c

(Figure 14.8 – Page 346)

```
/*
 * A program to check if one value is between two others, using
 * old-style declarations.  But it doesn't work.
 */
#include <stdio.h>

int main()
{
  int inrange();                              /* old-style prototype */
  int inpval;

  printf("Enter a value: ");
  scanf("%i", &inpval);
  printf("%i is %s range!\n",
         inpval, inrange(1, 100, inpval) ? "in" : "out of");
  return 0;
}

int inrange(min, max, value)
double min, max, value;                       /* old-style parameters */
{
  return value >= min && value <= max;
}
```

inrange2.c

(Figure 14.9 – Page 348)

```
/*
 * Another version of our range-checking program that uses
 * old-style declarations.  But this one works correctly.
 */
#include <stdio.h>

int main()
{
  int inrange();                          /* old-style prototype */
  float inpval, lowval = 0.0, highval = 100.0;

  printf("Enter a value: ");
  scanf("%f", &inpval);
  printf("%f is %s range!\n",
         inpval, inrange(lowval, highval, inpval) ? "in" : "out of");
  return 0;
}

int inrange(min, max, value)
double min, max, value;                   /* old-style parameters */
{
  return value >= min && value <= max;
}
```

inrange3.c

(Figure 14.10 – Page 349)

```
/*
 * A version of our range-checking program that works sometimes,
 * but isn't portable.
 */
#include <stdio.h>

int main()
{
  int inrange();                        /* old-style prototype */
  int inpval;

  printf("Enter a value: ");
  scanf("%i", &inpval);
  printf("%i is %s range!\n",
         inpval, inrange(1, 100, inpval) ? "in" : "out of");
  return 0;
}

int inrange(min, max, value)
long min, max, value;                   /* old-style parameters */
{
  return value >= min && value <= max;
}
```

inrange4.c

(Figure 14.11 – Page 350)

```
/*
 * A disastrous version of our range-checking program.  This one
 * doesn't work at all.
 */
#include <stdio.h>

int main()
{
  int inrange();                         /* old-style prototype */
  float inpval, lowval = 1.0, highval = 100.0;

  printf("Enter a value: ");
  scanf("%f", &inpval);
  printf("%f is %s range!\n",
         inpval, inrange(lowval, highval, inpval) ? "in" : "out of");
  return 0;
}

int inrange(float min, float max, float value)
{
  return value >= min && value <= max;
}
```

inrange5.c

(Figure 14.12 – Page 351)

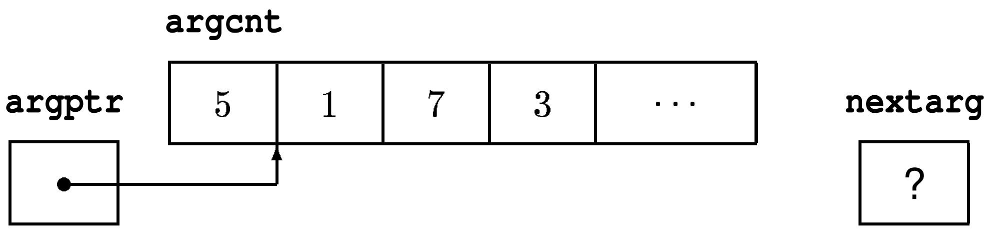
```
/*
 * Another disasterous version of our range-checking program.
 * This one also doesn't work at all.
 */
#include <stdio.h>

int main()
{
  int inrange(float min, float max, float value);
  float inpval, lowval = 1.0, highval = 100.0;

  printf("Enter a value: ");
  scanf("%f", &inpval);
  printf("%f is %s range!\n",
         inrange(lowval, highval, inpval) ? "in" : "out of");
  return 0;
}

int inrange(min, max, value)
float min, max, value;                    /* old-style parameters */
{
  return value >= min && value <= max;
}
```

max.c

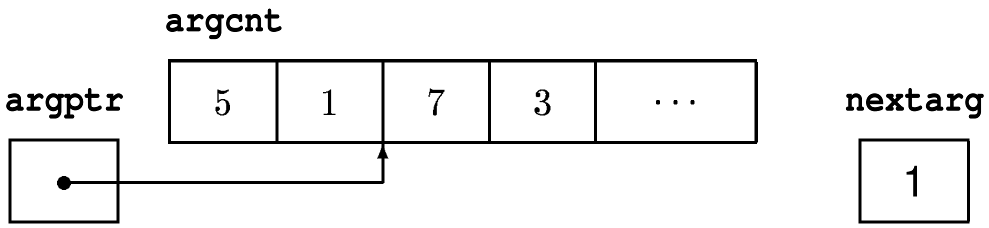
(Figure 14.13 – Page 352)

```
/*
 * Illustrate variable argument "max" function.
 */
#include <stdio.h>
#include <stdarg.h>
#include <limits.h>

int main()
{
  int max(int argcnt, ...);

  printf("%i\n", max(5,1,7,3,4,2));
  printf("%i\n", max(10,87,91,18,34,65,89,99,45,99,56));
  return 0;
}

/* Return maximum value of its arguments */

int max(int argcnt, ...)
{
  va_list argptr;                     /* argument pointer */
  int     nextarg;                    /* next variable argument */
  int     i;                          /* argument count */
  int     biggest = INT_MIN;          /* largest value*/

  va_start(argptr, argcnt);
  for (i = 0; i < argcnt; i++)
    if ((nextarg = va_arg(argptr, int)) > biggest)
      biggest = nextarg;
  va_end(argptr);
  return biggest;
}
```

VARIABLE NUMBERS OF ARGUMENTS



(a) Initially, after **`va_start`**.



(b) After the first call to **`va_arg`**.

getinp.c

(Figure 14.15 – Page 355)

```c
/*
 * A nice input fetching function.
 */
#include <stdio.h>
#include <string.h>
#include <stdarg.h>

#define  MAXLEN  80

int main()
{
  int    getinput(const char *control, ...);
  char   name[MAXLEN + 1];
  double sal;
  int    age;

  getinput("sri", "Name?", name, "Salary?", &sal, "Age?", &age);
  printf("Name: %s\nSalary: %f\nAge: %i\n", name, sal, age);
  return 0;
}
```

getinp.c

(Figure 14.15 – Page 355)

```
/* Get an input value according to provided formatting string */

int getinput(const char *control, ...)
{
  int        getline(char *, int);
  va_list    argptr;                    /* argument pointer */
  const char *cptr = control;           /* next char in control */
  char       line[MAXLEN + 1];          /* next input line */

  va_start(argptr, control);
  for (; *cptr != '\0'; cptr++)
  {
    printf("%s ", va_arg(argptr, char *));
    if (getline(line, MAXLEN) == -1)
      break;
    switch(*cptr)
    {                                 /* assume successful reads here */
      case 's': strcpy(va_arg(argptr, char *), line);
                break;
      case 'r': sscanf(line, "%lf", va_arg(argptr, double *));
                break;
      case 'i': sscanf(line, "%i", va_arg(argptr, int *));
                break;
    }
  }
  va_end(argptr);
  return cptr - control;              /* successful reads */
}
```

Sample Output – debug

```
DEBUGGING main: x=10, y=20
```

debug.c

(Figure 14.16 – Page 356)

```
/*
 * Illustrates using "printf"-like debugging function.
 */
#include <stdio.h>
#include <stdarg.h>

int main()
{
  void debug(const char *funcname, const char *format, ...);
  int x = 10, y = 20;

  debug("main", "x=%i, y=%i\n", x, y);
  return 0;
}

/* Print function name, and values of variables */

void debug(const char *funcname, const char *format, ...)
{
  va_list argptr;

  printf("DEBUGGING %s: ", funcname);
  va_start(argptr, format);
  vprintf(format, argptr);   /* print various arguments */
  va_end(argptr);
}
```

emp.h

(Figure 14.17 – Page 358)

```
/*
 * Definition of an employee record.  Differs from Chapter 11's
 * EMPLOYEEs in that we now store a salary rather than an age.
 */
#include <stdio.h>
#include <stddef.h>

typedef struct
{
  long   number;
  char   *name;
  char   *phone;
  double salary;
} EMPLOYEE;
```

empsrch1.c

(Figure 14.18 – Page 358)

```
/*
 * Employee table-searching functions.
 *   search_empno - find matching employee by number.
 *   search_name - find matching employee by name.
 */
#include <string.h>
#include "emp.h"

EMPLOYEE *search_empno(const EMPLOYEE *ptr, int n, long target)
{
  const EMPLOYEE *const endptr = ptr + n;

  for (; ptr < endptr; ptr++)
    if (ptr->number == target)
      return (EMPLOYEE *) ptr;
  return NULL;
}

EMPLOYEE *search_name(const EMPLOYEE *ptr, int n, char *target)
{
  const EMPLOYEE *const endptr = ptr + n;

  for (; ptr < endptr; ptr++)
    if (strcmp(ptr->name, target) == 0)
      return (EMPLOYEE *) ptr;
  return NULL;
}
```

empmain.c

(Figure 14.19 – Page 359)

```
/*
 * Use standard search on an array of structures.
 */
#include <string.h>
#include "emp.h"

int main()
{
  EMPLOYEE *search_empno(const EMPLOYEE tab[], int n, long target);
  EMPLOYEE *search_name(const EMPLOYEE tab[], int n, char *target);
  EMPLOYEE table[] =
            {{1001, "Daphne Borromeo",    "213-555-2134", 8.78},
             {1011, "Tammy Merriweather", "213-555-1212", 4.50},
             {1140, "Doris Perl",         "213-555-1215", 5.60},
             {2045, "Barbara Wong",       "213-555-1219", 7.80},
             {2945, "Diane Goebel",       "213-555-1245", 9.00},
             {3300, "Irene Borromeo",     "213-555-1210", 3.97},
             {4011, "Veronica Ockert",    "213-555-1212", 4.50},
             {4140, "Tony Quilici",       "213-555-1215", 5.60},
             {5045, "Brian Hight",        "213-555-1219", 7.80},
             {5945, "Gisele Pham",        "213-555-1245", 9.00}};
  const int entries = sizeof(table)/sizeof(table[0]);
  EMPLOYEE *matchptr;

  if ((matchptr = search_empno(table, entries, 1045)) != NULL)
    printf("1045 is in record %i\n", matchptr - table);
  else
    printf("1045 wasn't found\n");
  if ((matchptr = search_name(table, entries, "Tony Quilici")) != NULL)
    printf("Tony Quilici is in record %i\n", matchptr - table);
  else
    printf("Tony Quilici wasn't found.\n");
  return 0;
}
```

empsrch2.c

(Figure 14.20 – Page 361)

```
/*
 * New versions of our search functions.
 *   search_empno - find matching employee by number.
 *   empnocmp - employee-number comparison function.
 *   search_name - find matching employee by name.
 *   namecmp - employee-name comparison function.
 *   empsearch - actual function to search employee table.
 */
#include <string.h>
#include "emp.h"

static EMPLOYEE *empsearch(const EMPLOYEE *ptr,
                           int n,
                           const void *tarptr,
                           int (*fptr)(const void *, const EMPLOYEE *));
static int empnocmp(const void *tarptr, const EMPLOYEE *entryptr);
static int namecmp(const void *tarptr, const EMPLOYEE *entryptr);

EMPLOYEE *search_empno(EMPLOYEE *ptr, int n, long num)
{
  return empsearch(ptr, n, &num, empnocmp);
}
```

empsrch2.c

(Figure 14.20 – Page 361)

```
static int empnocmp(const void *tarptr, const EMPLOYEE *entryptr)
{
  return * (long *) tarptr != entryptr->number;
}

EMPLOYEE *search_name(const EMPLOYEE *ptr, int n, char *name)
{
  return empsearch(ptr, n, name, namecmp);
}

static int namecmp(const void *tarptr, const EMPLOYEE *entryptr)
{
  return strcmp((char *) tarptr, entryptr->name);
}

static EMPLOYEE *empsearch(const EMPLOYEE *ptr,
                           int n,
                           const void *tarptr,
                           int (*funcptr)(const void *, const EMPLOYEE *))
{
  const EMPLOYEE *const endptr = ptr + n;

  for (; ptr < endptr; ptr++)
    if ((*funcptr)(tarptr, ptr) == 0)
      return (EMPLOYEE *) ptr;
  return NULL;
}
```

Sample Output – fact

```
24
Here's a trace.
Computing 4 factorial
Computing 3 factorial
Computing 2 factorial
Computing 1 factorial
Computed 1 factorial = 1
Computed 2 factorial = 2
Computed 3 factorial = 6
Computed 4 factorial = 24
24
```

fact.c

(Figure 14.21 – Page 364)

```
/*
 * Use factorial function to compute 4!.
 */
#include <stdio.h>

int main()
{
  long fact(int n);                    /* compute factorial */
  long trace_fact(int n);              /* provides tracing info */

  printf("%li\n", fact(4));            /* test it */
  printf("Here's a trace.\n");
  printf("%li\n", trace_fact(4));      /* with trace */
  return 0;
}

long fact(int n)                       /* compute N! for N >= 0 */
{
  return (n <= 1) ? 1 : n * fact(n-1);
}

long trace_fact(int n)                 /* also output tracing info */
{
  long temp;

  printf("Computing %i factorial\n", n);
  temp = (n <= 1) ? 1 : n * trace_fact(n-1);
  printf("Computed %i factorial = %li\n", n, temp);
  return temp;
}
```

Sample Output – permute

```
alex alxe aelx aexl axel axle
laex laxe leax lexa lxea lxae
elax elxa ealx eaxl exal exla
xlea xlae xela xeal xael xale
```

permute.c

(Figure 14.23 – Page 365)

```
/*
 * Produce permutations, assuming sufficient memory.
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int Len;                            /* length of string to permute */

main(int argc, char *argv[])
{
  void permute(const char *s, int pos);
  int i;

  for (i = 1; i < argc; i++)
  {
    Len = strlen(argv[i]);
    permute(argv[i], 0);
  }
  return 0;
}
```

permute.c

(Figure 14.23 – Page 365)

```
/* Generate all permutations of "s" */

void permute(const char *s, int changepos)
{
  void swap(char *ptr1, char *ptr2);
  char *scopy;
  int  i;

  if (changepos < Len)
    for (i = changepos; i < Len; i++)
    {
      scopy = malloc(Len + 1);    /* assumes malloc succeeds */
      strcpy(scopy, s);
      swap(&scopy[i], &scopy[changepos]);
      permute(scopy, changepos + 1);
      free(scopy);
    }
  else
    printf("%s\n", s);
}

/* Exchange two character variables in caller */

void swap(char *p, char *q)
{
  char temp;

  temp = *p;  *p = *q;  *q = temp;
}
```

binsrch.c

(Figure 14.24 – Page 367)

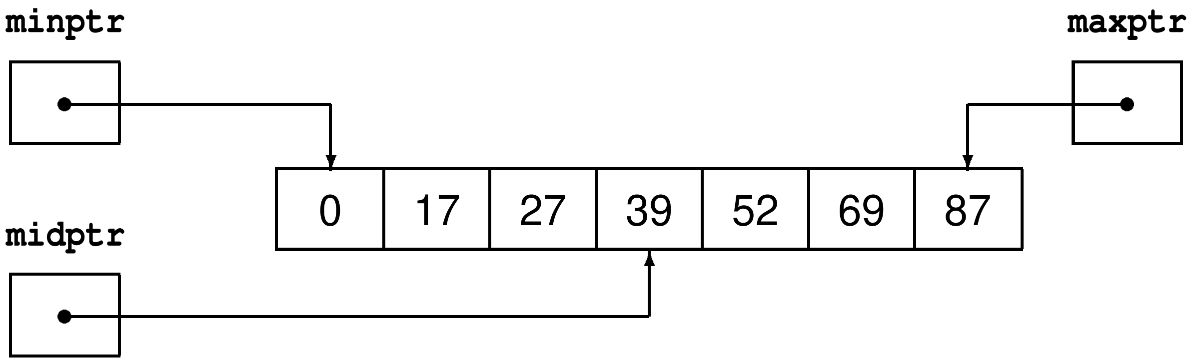
```
/*
 * Program using binary search for values given to it as arguments.
 */
#include <stdio.h>
#include <stddef.h>

#define MAX    100

int main(int argc, char *argv[])
{
  long *binsearch(const long *min, const long *max, long t);
  long table[MAX];                      /* table to search */
  long t;                               /* target */
  long *fndptr;                         /* pointer to item in array */
  int i;                                /* array index */

  for (i = 0; i < MAX; i++)             /* initialize table with */
    table[i] = i;                       /*    0,1,2,3,... */

  for (i = 1; i < argc; i++)
   if (sscanf(argv[i], "%li", &t) == 1)
   {
     fndptr = binsearch(&table[0], &table[MAX - 1], t);
     if (fndptr != NULL)
       printf("Found %li as element %i.\n", t, fndptr - table);
     else
       printf("Didn't find %li.\n", t);
    }
  return 0;
}
```

binsrch.c

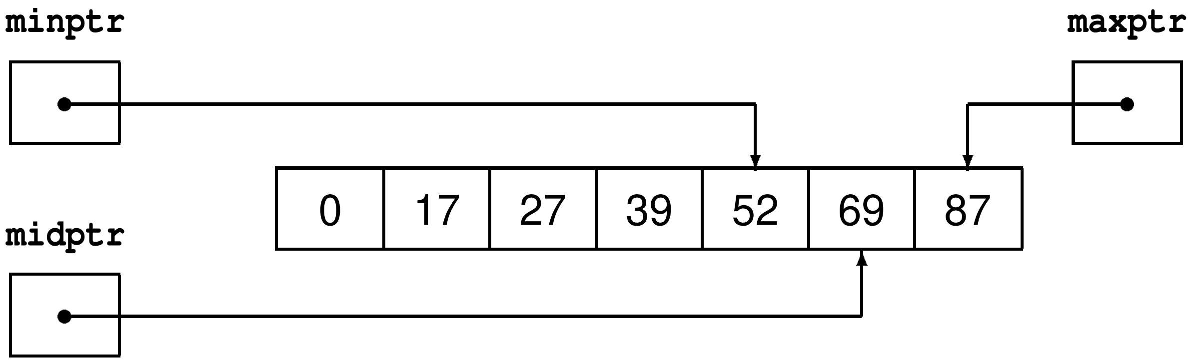
(Figure 14.24 – Page 367)

```
/* Use binary search to search a table of longs */

long *binsearch(const long *minptr, const long *maxptr, long target)
{
  const long *midptr = minptr + (maxptr - minptr) / 2;

  if (target == *midptr)               /* middle is target? */
    return (long *) midptr;
  if (maxptr <= minptr)                /* value's not in table? */
    return NULL;

  return (target < *midptr)            /* search correct half */
           ? binsearch(minptr, midptr - 1, target)
           : binsearch(midptr + 1, maxptr, target);
}
```

SUCCESSFUL BINARY SEARCH

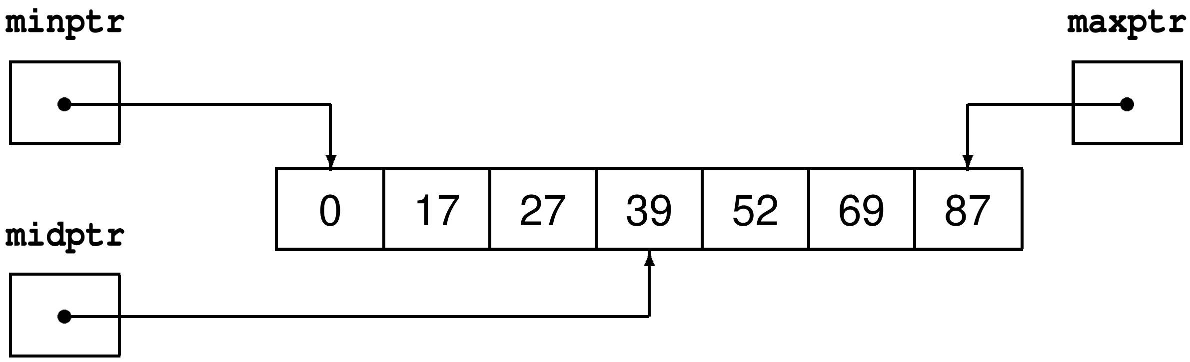


(a) At start: 39 is less than 69, so search top half of table.

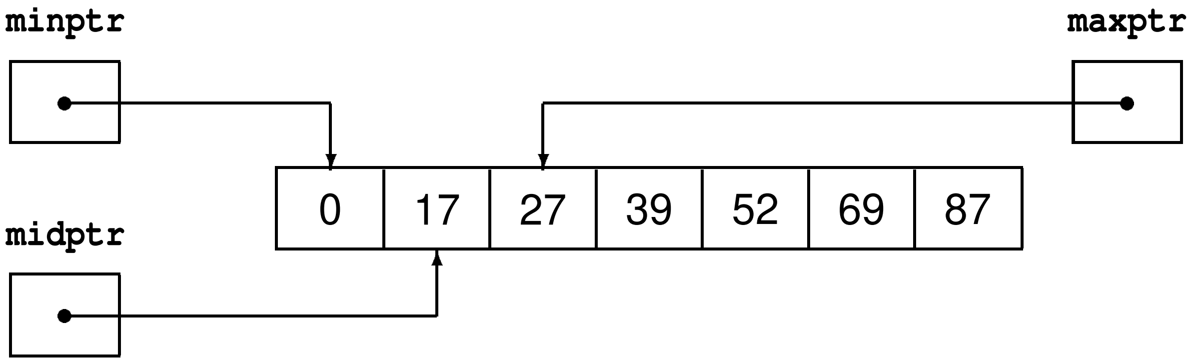


(b) After calling **`binsearch`** once, found it.

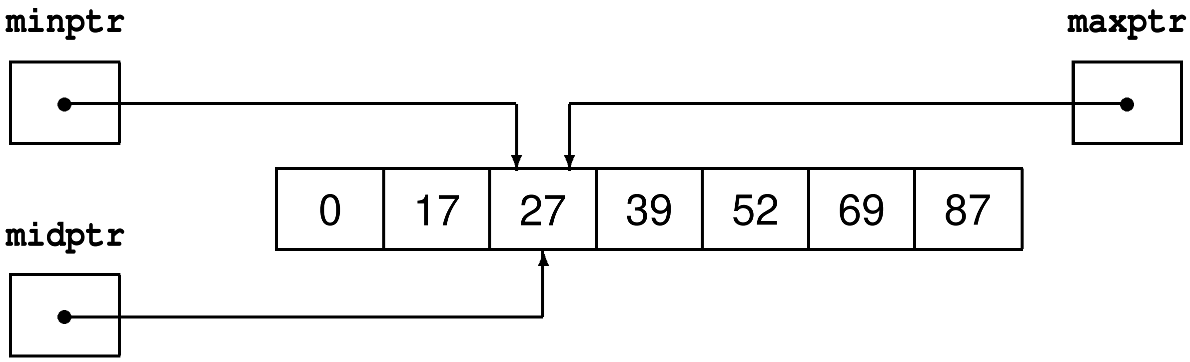
FAILED BINARY SEARCH



(a) At start: 39 is greater than 28, so search lower half of table.



(b) After calling **binsearch** once, 28 greater than 17.



(c) Call **binsearch** again, 28 not equal to 27, so we stop.

REMINDERS

- Don't fail to pass a pointer to a function that expects one.
- Don't forget to dereference pointer parameters when you need to access their values.
- Don't pass a pointer's address instead of the pointer itself.
- With old-style declarations and definitions, type mismatches usually go undetected at compile time.
- Be sure to call **`va_start`** before trying to access the variable arguments.

CHAPTER SUMMARY

- We can simulate call-by-reference parameter passing by using pointers. We pass the equivalent of a call-by-reference parameter by passing its address, which the function then dereferences to access and change its value.

- We can write functions with variable numbers of arguments using the *stdargs* package.

- We can write generic functions by using pointers to functions. These generic functions save us repeating the same code over and over when all that's really different is the type of data we're manipulating.

- We're allowed to have recursive functions—functions that call themselves.

- ANSI-C actually provides two styles for function declarations and definitions. The older style doesn't provide automatic type checking or conversions and is provided for backward compatibility.