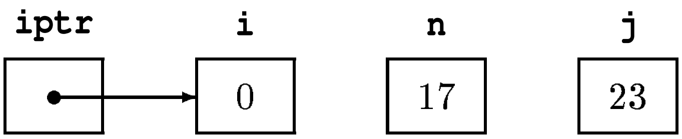
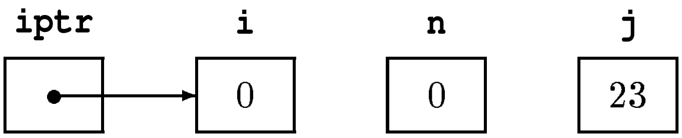
CHAPTER 9

POINTERS AND ARRAYS

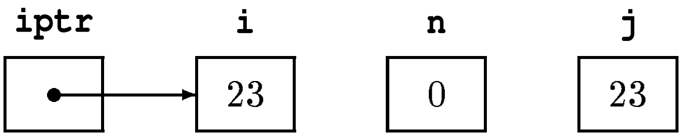
EXAMPLE POINTER OPERATIONS



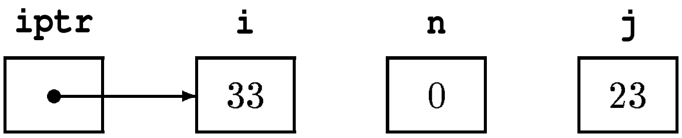
(a) `i = 0; iptr = &i; n = 17; j = 23;`



(b) `n = *iptr;`



(c) `*iptr = j;`



(d) `*iptr = *iptr + 10;`

Sample Output – ptrex

```
Addresses: &i=FFB6 &n=FFB8 &j=FFBA
Initial values: iptr=0000 i=0 n=17 j=23
Later values: iptr=FFB6 i=0 n=0 j=23
Final values: iptr=FFB6 i=33 n=0 j=23
```

ptrex.c

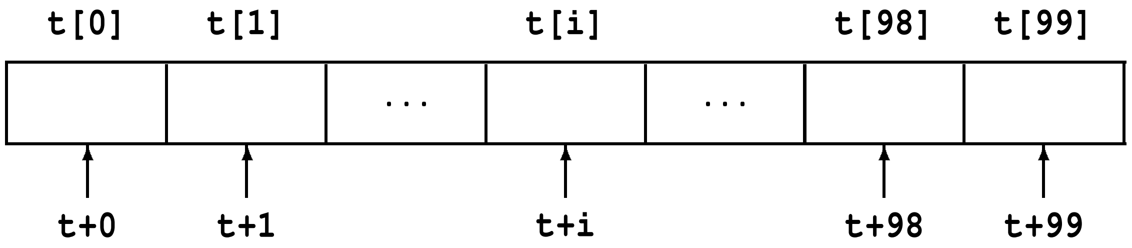
(Figure 9.2 – Page 206)

```
/*
 * Examples of assigning, following, and printing pointers.
 */
#include <stdio.h>

int main()
{
  int *iptr = NULL;
  int i = 0;                            /* some useful value */
  int n = 17;                           /* another value */
  int j = 23;                           /* still another value */

  printf("Addresses: ");                /* display addresses */
  printf("&i=%p", (void *) &i);
  printf(" &n=%p", (void *) &n);
  printf(" &j=%p\n", (void *) &j);

  printf("Initial values: ");           /* display initial values */
  printf("iptr=%p i=%i n=%i j=%i\n", (void *) iptr, i, n, j);
  iptr = &i;                            /* place i's address in iptr */
  n = *iptr;                            /* place i's value to n */

  printf("Later values: ");             /* display changed values */
  printf("iptr=%p i=%i n=%i j=%i\n", (void *) iptr, i, n, j);

  *iptr = j;                            /* place j's value in n */
  *iptr = *iptr + 10;                   /* update n */
  printf("Final values: ");
  printf("iptr=%p i=%i n=%i j=%i\n", (void *) iptr, i, n, j);
  return 0;
}
```

totavg.c
(Not In Text)

```
/*
 * Compute total and average of read in values.
 */
#include <stdio.h>

#define TO_READ 4      /* scores to read for student */

main()
{
  int obtain_scores(int scores_to_read, int *total_ptr, double *avg_ptr);
  int student_total;
  double student_avg;

  if (obtain_scores(TO_READ, &student_total, &student_avg) != -1)
    printf("Total is %i.\nAverage is %f\n", student_total, student_avg);
  else
    printf("Problem reading student.\n");
  return 0;
}

/*
 * Try to read a set of scores, filling in the total and average.
 */
int obtain_scores(int scores_to_read, int *total_ptr, double *avg_ptr)
{
  int scores_read;
  int next_score;
  int total_score = 0;

  for (scores_read = 0; scores_read < scores_to_read; scores_read++)
  {
    if (scanf("%i", &next_score) != 1)
      return -1;   /* failure */
    total_score += next_score;
  }
  *total_ptr = total_score;
  *avg_ptr = (scores_read != 0) ? (double) total_score / scores_read : 0.0;
  return 0;        /* success */
}
```

DECLARING AN ARRAY



Sample I/O – useavg

```
10 40 20 30 30 100 20 10 20 20
_ _ _ _ _
There are 10 values.
The average is 30.
There are 2 values above average.
There are 6 values below average.
```

useavg.c

(Figure 9.4 – Page 209)

```
/*
 * Compute average and print interesting counts.  Uses:
 *   table_fill - reads in table entries (same table_fill as before).
 *   table_avg_cnts - compute average statistics.
 */
#include <stdio.h>

#define MAXVALS 100

int main()
{
  int    table_fill(int a[], int max);
  void   table_avg_cnts(int a[], int n, double avg);
  double table_average(int a[], int n);
  int    t[MAXVALS];
  int    n   = table_fill(t, MAXVALS);
  double avg = table_average(t, n);

  printf("There are %i values.\n", n);
  printf("The average is %g.\n", avg);
  table_avg_cnts(t, n, avg);
  return 0;
}
```

useavg.c

(Figure 9.4 – Page 209)

```
int table_fill(int a[], int max)
{
  int count = 0;

  for (; count < max; count++)
    if (scanf("%i", &a[count]) != 1)
      break;                                 /* kick out on error */
  return count;
}

void table_avg_cnts(int a[], int n, double avg)
{
  int i;                                     /* index */
  int above = 0;                             /* above average */
  int below = 0;                             /* below average */

  for (i = 0; i < n; i++)
    if (a[i] > avg)
      above++;
    else if (a[i] < avg)
      below++;
  printf("There are %i values above average.\n", above);
  printf("There are %i values below average.\n", below);
}
```

avg.c

(Figure 9.5 – Page 210)

```
/*
 * Compute average of an array, array subscripting version.
 */
double table_average(int a[], int n)
{
  double sum = 0.0;                              /* running total */
  int    i;                                      /* count of items */

  for (i = 0; i < n; i++)
    sum += a[i];
  return (n != 0) ? sum / n : 0.0;
}
```

avg2.c

(Figure 9.6 – Page 211)

```
/*
 * Compute average of an array, pointer version.
 */
double table_average(int a[], int n)
{
  double sum = 0.0;                              /* running total */
  int    i;                                      /* count of items */
  int    *ptr;                                   /* traversing pointer */

  ptr = a;
  for (i = 0; i < n; i++)
  {
    sum += *ptr;
    ptr++;
  }
  return (n != 0) ? sum / n : 0.0;
}
```

avg3.c
(Figure 9.7 – Page 213)

```
/*
 * Compute average of an array, concise pointer version.
 */
double table_average(int a[], int n)
{
  double sum = 0.0;                  /* running total */
  int    *ptr;                       /* traversing pointer */
  int    *endptr = a + n;            /* pointer to just past end */

  for (ptr = a; ptr < endptr; ptr++)
    sum += *ptr;
  return (n != 0) ? sum / n : 0.0;
}
```

revtab.c

(Figure 9.8 – Page 214)

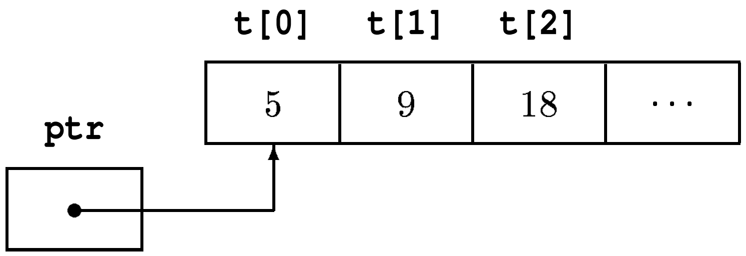
```
/*
 * Pointer version of input-reversal program.
 */
#define MAXVALS   100                      /* max values in table */

int main()
{
  int  table_fill(int a[], int max);
  void table_print_rev(int a[], int num);
  int  t[MAXVALS];                         /* input values*/
  int  n = table_fill(t, MAXVALS);

  table_print_rev(t, n);
  return 0;
}
```

table.c

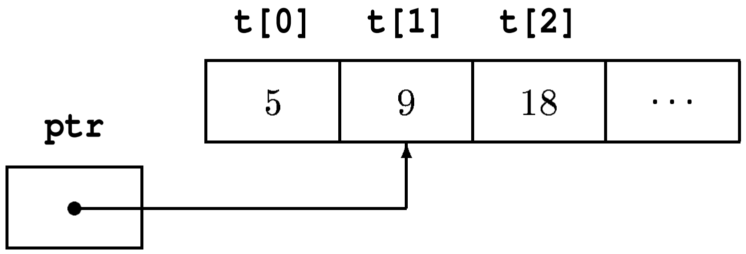
(Figure 9.9 – Page 214)

```
/*
 * Pointer versions of table-handling functions.
 *   table_fill - read values into table.
 *   table_print_rev - print values in reverse order.
 */
#include <stdio.h>

int table_fill(int a[], int max)
{
  int *ptr = a;                    /* pointer to first element */
  int *endptr = ptr + max;         /* pointer to just past last element */

  for (ptr = a; ptr < endptr; ptr++)
    if (scanf("%i", ptr) != 1)
      break;
  return ptr - a;                  /* # of values read successfully */
}

void table_print_rev(int a[], int num)
{
  int *ptr = a + num;              /* pointer to just past last element */

  while (ptr-- > a)
    printf("%i\n", *ptr);
}
```

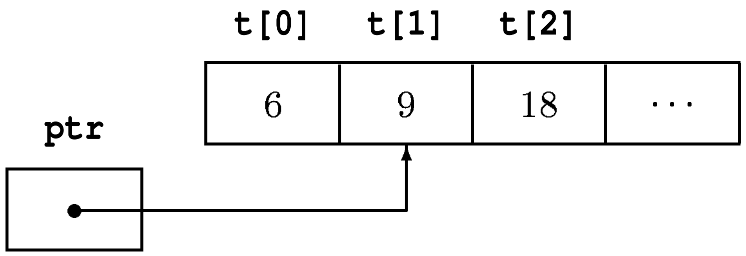
POINTER INCREMENT AND DECREMENT



(a) Initial assignment: **`ptr = &t[0]`** (or equivalently, **`ptr=t`**).



(b) **`*ptr++`** increments **`ptr`** and returns 5, so **`v = *ptr++`** sets **`v`** to 5.



(c) **`(*ptr)++`** increments **`t[1]`** and returns 9, so **`v = (*ptr)++`** sets **`v`** to 9.

drevtab.c

(Figure 9.11 – Page 216)

```
/*
 * Input-reversal program using pointers.  For doubles, not ints.
 */
#define MAXVALS 100

int main()
{
  int    dbl_table_fill(double d[], int max);
  void   dbl_table_print_rev(double d[], int num);
  double table[MAXVALS];
  int    n = dbl_table_fill(table, MAXVALS);

  dbl_table_print_rev(table, n);
  return 0;
}
```

dtable.c

(Figure 9.12 – Page 216)

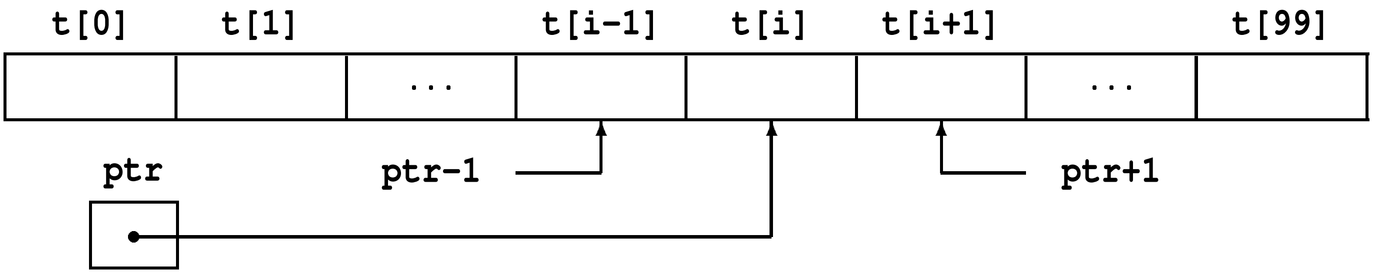
```
/*
 * Table-handling functions using pointers.  For doubles, not ints.
 *   dbl_table_fill - read values into table.
 *   dbl_table_print_rev - print table in reverse order.
 */
#include <stdio.h>

int dbl_table_fill(double a[], int max)
{
  double *ptr = a;               /* pointer to first element */
  double *endptr = ptr + max;    /* pointer to just after last element */

  for (ptr = a; ptr < endptr; ptr++)
    if (scanf("%lf", ptr) != 1)
      break;
  return ptr - a;                /* # of values read successfully */
}

void dbl_table_print_rev(double a[], int num)
{
  double *ptr = a + num;

  while (ptr-- > a)
    printf("%f\n", *ptr);
}
```

avg4.c

(Figure 9.13 – Page 217)

```
/*
 * Function to compute average of array, most concise version.
 */
double table_average(int *ptr, int n)
{
  double sum = 0.0;
  int  *endptr = ptr + n;   /* pointer to just past last element */

  while (ptr < endptr)
    sum += *ptr++;
  return (n != 0) ? sum / n : 0.0;
}
```

PASSING PART OF AN ARRAY



revtab2.c

(Figure 9.15 – Page 220)

```
/*
 * Pointer version of input-reversal program, using constants.
 */
#define MAXVALS   100                  /* max values in "table" */

int main()
{
  int  table_fill(int *ptr, int max);
  void table_print_rev(const int a[], int num);
  int  t[MAXVALS];                     /* table for input values*/
  int  n = table_fill(t, MAXVALS);

  table_print_rev(t, n);
  return 0;
}
```

table2.c

(Figure 9.16 – Page 220)

```
/*
 * Pointer versions of table-handling functions:
 *   table_fill - read values into table.
 *   table_print_rev - print values in reverse order.
 */
#include <stdio.h>

int table_fill(int *ptr, int max)
{
  int *const firstptr = ptr;       /* pointer to first element */
  int *const endptr = ptr + max;   /* pointer to just past end */

  for ( ; ptr < endptr; ptr++)
    if (scanf("%i", ptr) != 1)
      break;
  return ptr - firstptr;           /* # of values read successfully */
}

void table_print_rev(const int a[], int num)
{
  const int *const firstptr = a;
  const int *ptr;

  ptr = firstptr + num;
  while (ptr-- > firstptr)
    printf("%i\n", *ptr);
}
```

BYTE-ACCESSING FUNCTIONS

FUNCTION	WHAT IT DOES
`void *memcpy(d,s,n)`	Copy **`n`** bytes from the location pointed to by **`s2`** into the location pointed to by **`s1`**. Returns **`d`**. Shouldn't use it for overlapping memory locations.
`void *memmove(d,s,n)`	Exactly like **`memcpy`** except that the memory locations may overlap.
`int memcmp(s1,s2,n)`	Compare the first **`n`** bytes in **`s1`** with the first **`n`** bytes in **`s2`**. Returns a negative value if **`s1`** $<$ **`s2`**, zero if they're equal, and a positive value if **`s1`** $>$ **`s2`**.
`void *memchr(s,c,n)`	Find the first occurrence of **`c`** in the first **`n`** bytes of **`s`**. Returns a pointer to the character if found, or the null pointer otherwise.
`void *memset(d,c,n)`	Copy the byte **`c`** into the first **`n`** locations pointed to by **`d`**. Returns **`d`**.

memcpy.c

(Figure 9.17 – Page 222)

```
/*
 * One possible implementation of memcpy, along with a program that
 * uses it to copy one array into another.
 */
#include <stdio.h>
#define MAXVALS 10              /* number of items in array */

typedef unsigned char BYTE;    /* BYTE's a synonym for unsigned char */

int main()
{
  void ourmemcpy(void *xptr, const void *yptr, size_t n);
  double f[MAXVALS];
  double g[MAXVALS] = {7.5, 4.5, 5.4, 9.8, 9.5, 8.2, 9.1, 9.9, 4.5, 6.3};
  int    i;

  ourmemcpy(f, g, sizeof(g));          /* copy g into f */
  for (i = 0; i < MAXVALS; i++)        /* print f to see result */
    printf("f[%i]=%f\n", i, f[i]);
  return 0;
}

void ourmemcpy(void *xptr, const void *yptr, size_t n)
{
  BYTE       *destptr;
  const BYTE *srcptr;
  const BYTE *endptr;

  destptr = xptr;
  srcptr = yptr;
  endptr = srcptr + n;
  while (srcptr < endptr)
    *destptr++ = *srcptr++;
}
```

MEMORY-ALLOCATION FUNCTIONS

FUNCTION	WHAT IT DOES
`void *calloc(n,size)`	Allocates space for **`n`** items, each of **`size`** bytes. Sets the allocated space to zero. Returns a pointer to the beginning of the allocated space, or **`NULL`** if the space can't be allocated.
`void free(p)`	Allows allocated space pointed to by **`p`** to be allocated again.
`void *malloc(size)`	Allocate space for **`size`** bytes. Doesn't initialize the allocated space. Returns a pointer to the beginning of the allocated space, or **`NULL`** if the space can't be allocated.
`void *realloc(p,size)`	Changes the space allocated by **`p`** to be **`size`** bytes. Returns a pointer to the beginning of the allocated space, or **`NULL`** if the space can't be allocated. The contents of the space pointed to by **`p`** are unchanged.

Sample I/O – grprev

```
3 35.5 25.3 45.5
2 20.2 12.6
_ _ _ _ _
```

```
45.500000
25.300000
35.500000
12.600000
20.200000
```

grprev.c

(Figure 9.18 – Page 225)

```
/*
 * Read and reverse groups within the input.
 */
#include <stdio.h>
#include <stddef.h>              /* definition of NULL */
#include <stdlib.h>              /* definition of malloc */

int main()
{
  int     dbl_table_fill(double a[], int max);
  void    dbl_table_print_rev(double a[], int n);
  double *tptr;                  /* holds the input array */
  int     n;                     /* elements in group, table */
  int     res;                   /* scanf result */

  while ((res = scanf("%i", &n)) == 1)
  {
    tptr = malloc(n * sizeof(double));
    if (tptr == NULL)
    {
      printf("Couldn't allocate %i elements\n", n);
      return 1;   /* out of memory error, quit */
    }
    dbl_table_print_rev(tptr, dbl_table_fill(tptr, n));
    free(tptr);
  }
  if (res != EOF)
    printf("Couldn't read all elements\n");
  return res == EOF;
}
```

sets.h

(Figure 9.19 – Page 227)

```
/*
 * Improved definitions to use "sets" of integers.
 */
typedef void *SET;

extern SET   set_create(unsigned int n);
extern void  set_destroy(SET s);
extern void  set_add(SET s, int e);
extern void  set_delete(SET s, int e);
extern int   set_member(const SET s, int e);
```

sets.c

(Figure 9.20 – Page 228)

```
/*
 * Functions to handle dynamically allocated sets.
 */
#include <stdlib.h>
#include "sets.h"

#define US_BITS 16

typedef unsigned short USHORT;

static int word(int elem);
static int bit(int elem);

SET set_create(unsigned int n)          /* create an empty set */
{
  return calloc((n / US_BITS) + ((n % US_BITS) != 0), sizeof(USHORT));
}

void set_destroy(SET s)                 /* get rid of a set */
{
  free(s);
}
```

sets.c

(Figure 9.20 – Page 228)

```
void set_add(SET s, int elem)           /* add element to set */
{
  * ((USHORT *) s + word(elem)) |= 1 << bit(elem);
}

void set_delete(SET s, int elem)        /* delete element from set */
{
  * ((USHORT *) s + word(elem)) &= ~(1 << bit(elem));
}

int set_member(const SET s, int elem)   /* is item in set? */
{
  const unsigned int temp = * ((const USHORT *) s + word(elem));

  return (temp >> bit(elem)) & 01;
}

static int word(int elem)
{
  return elem / US_BITS;                /* word # containing element */
}

static int bit(int elem)
{
  return elem % US_BITS;                /* bit # representing element */
}
```

usesets.c

(Figure 9.21 – Page 229)

```
/*
 * Identify duplicates in the input.
 */
#include <stdio.h>
#include "sets.h"                         /* for SETs */

int main()
{
  void  set_print(const SET s, int max);
  void  set_switch(SET news, SET olds, int elem);
  const int MAXELEMS = 512;               /* # of set elements */
  SET   unique = set_create(MAXELEMS),    /* unique and */
        dup    = set_create(MAXELEMS);    /*   duplicate sets */
  int   r, inp;                           /* scanf return, input value */

  while ((r = scanf("%i", &inp)) != EOF)
  {
    if (r != 1)
    {
      printf("Value in error.\n");
      break;
    }
    if (inp < 0 || inp >= MAXELEMS)
    {
      printf("Value out of range.\n");
      continue;
    }
    if (set_member(unique, inp))
      set_switch(dup, unique, inp);
    else if (!set_member(dup, inp))
      set_add(unique, inp);
  }
  printf("Unique values\n");  set_print(unique, MAXELEMS);
  printf("Duplicate values\n");  set_print(dup, MAXELEMS);
  return 0;
}
```

usesets.c

(Figure 9.21 – Page 229)

```
void set_switch(SET new, SET old, int value)
{
  set_delete(old, value);  set_add(new, value);
}

void set_print(const SET set, int max)
{
  int i;                              /* next potential element */

  for (i = 0; i < max; i++)
    if (set_member(set, i))
      printf("%i\n", i);
}
```

REMINDERS

- Don’t dereference a pointer variable until you’ve assigned it an address.
- The only legal arithmetic operators on pointers are adding or subtracting an integer, or subtracting one pointer from another.
- Don’t compare pointers that don’t access the same array.
- Don’t assume that a pointer to the element just before the start of an array is legal.
- Don’t pass an array element to a function expecting an array parameter.
- Don’t assume **`malloc`** will always succeed.
- Don’t assume the storage **`malloc`** provides is initialized to zero.
- Don’t modify the pointer returned by **`malloc`**.
- Only **`free`** pointers obtained from **`malloc`**, and don’t access the storage after it’s been **`free`**d.

CHAPTER SUMMARY

- C provides pointers, a data type that can hold addresses. We obtain pointers with **`&`** and obtain the value they point to with **`*`**.

- There is a generic pointer type, **`void *`**, that can contain any type of pointer. But before we can make use of that pointer, we must cast it to another type.

- We can add or subtract an integer and a pointer, and we can subtract one pointer from another. The arithmetic takes place in units of the pointed-to type. We can also compare two pointers, so long as they point to the same array.

- To print a pointer, we cast it to **`void *`** and use **`printf`**'s **`%p`** formatting code.

- We can combine pointer arithmetic and comparisons to traverse arrays more efficiently than with array subscripting. Pointer arithmetic takes place in units of the pointed-to type.

- C has several library functions to manage dynamic memory allocation: **`malloc`**, which allocates a chunk of memory; **`calloc`**, which allocates and sets to zero a chunk of memory; and **`free`**, which returns a chunk of memory.