# BBM 102 – INTRODUCTION TO PROGRAMMING II

*SPRING 2018*

Structs & File IO

***Instructors:*** *Dr. Cumhur Yiğit Özcan, Dr. Ali Seydi Keçeli, Dr. Aydın Kaya*

**source: Deitel&Deitel – C How To Program*

# Structures

- Collections of related variables (aggregates) under one name
  - Can contain variables of different data types
- Commonly used to define records to be stored in files
- Combined with pointers, can create linked lists, stacks, queues, and trees

# Structure Definitions

Example 1:

```
struct card {
    char *face;
    char *suit;
};
```

- `struct` introduces the definition for structure `card`
- `card` is the structure name and is used to declare variables of the structure type
- `card` contains two members of type `char *`
  - These members are `face` and `suit`

# Structure Definitions

- A structure definition does not reserve space in memory
  - Instead creates a new data type used to define structure variables

- Variables can be defined as below:

```
struct card {
   char *face;
   char *suit;
} oneCard, deck[ 52 ], *cPtr;
```

- Or defined like other variables:

```
struct card {
   char *face;
   char *suit;
};
struct card oneCard, deck[ 52 ], *cPtr;
```

# Structure Definitions

Example 2:

```
struct point {
     int x;
     int y;
};

struct point pt; /* defines a variable pt which
                    is a structure of type
                    struct point */

pt.x = 15;
pt.y = 30;
printf("%d, %d", pt.x, pt.y);
```

# Structure Definitions

```
/* Structures can be nested.  One representation of
 a  rectangle is a pair of points that denote the
 diagonally opposite corners. */

struct rect {
     struct point pt1;
     struct point pt2;
};

struct rect screen;

/* Print the pt1 field of screen */
printf("%d, %d",screen.pt1.x,screen.pt1.y);

/* Print the pt2 field of screen */
printf("%d, %d",screen.pt2.x,screen.pt2.y);
```

# Structure Definitions

## Valid Operations

- Assigning a structure to a structure of the same type
- Taking the address (&) of a structure
- Accessing the members of a structure
- Using the sizeof operator to determine the size of a structure

# Initializing Structures

- Initializer lists
  - Example:

    ```
    struct card oneCard = { "Three", "Hearts" };
    ```

- Assignment statements
  - Example:

    ```
    struct card threeHearts = oneCard;
    ```

  - Could also define and initialize threeHearts as follows:

    ```
    struct card threeHearts;
    threeHearts.face = “Three”;
    threeHearts.suit = “Hearts”;
    ```

# Accessing Members of Structures

- Accessing structure members
  - Dot operator (`.`) used with structure variables

```
struct card myCard;
printf( "%s", myCard.suit );
```

  - Arrow operator (`->`) used with pointers to structure variables

```
struct card *myCardPtr = &myCard;
printf( "%s", myCardPtr->suit );
```

  - `myCardPtr->suit` is equivalent to

```
( *myCardPtr ).suit
```

```c
// Fig. 10.2: fig10_02.c
// Structure member operator and
// structure pointer operator
#include <stdio.h>

// card structure definition
struct card {
   char *face; // define pointer face
   char *suit; // define pointer suit
};

int main(void)
{
   struct card aCard; // define one struct card variable

   // place strings into aCard
   aCard.face = "Ace";
   aCard.suit = "Spades";

   struct card *cardPtr = &aCard; // assign address of aCard to cardPtr

```

**Fig. 10.2** | Structure member operator and structure pointer operator. (Part 1 of 2.)

```
22      printf("%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
23         cardPtr->face, " of ", cardPtr->suit,
24         (*cardPtr).face, " of ", (*cardPtr).suit);
25   }
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

**Fig. 10.2** | Structure member operator and structure pointer operator. (Part 2 of 2.)

# typedef

## typedef

- Creates synonyms (aliases) for previously defined data types
- Use **typedef** to create shorter type names

Example:

```
typedef struct point pixel;
```

- Defines a new type name **pixel** as a synonym for type **struct point**

```
typedef struct Card *CardPtr;
```

- Defines a new type name **CardPtr** as a synonym for type **struct Card ***

- **typedef** does not create a new data type
  - Only creates an alias

# Using Structures With Functions

- Passing structures to functions
  - Pass entire structure
    - Or, pass individual members
  - Both pass call by value
- To pass structures call-by-reference
  - Pass its address
  - Pass reference to it
- To pass arrays call-by-value
  - Create a structure with the array as a member
  - Pass the structure

# Using Structures with Functions 1

```
#include<stdio.h> /* Demonstrates passing a structure to a
  function */

struct data{
    int amount;
    char fname[30];
    char lname[30];
}rec;

void printRecord(struct data x){
   printf("\nDonor %s %s gave $%d", x.fname, x.lname, x.amount);
}

int main(void){
      printf("Enter the donor's first and last names\n");
      printf("separated by a space:  ");
      scanf("%s %s",rec.fname, rec.lname);
      printf("Enter the donation amount:   ");
      scanf("%d",&rec.amount);
      printRecord(rec);
      return 0;
}
```

# Using Structures with Functions 2

```
/* Make a point from x and y components. */
struct point makepoint (int x, int y)
{
        struct point temp;

        temp.x = x;
        temp.y = y;
        return (temp);

}

/* makepoint can now be used to initialize a structure */
struct rect screen;
struct point middle;

screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(50,100);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

```
/* add two points */

struct point addpoint (struct point p1, struct point p2)
{
     p1.x += p2.x;
     p1.y += p2.y;
     return p1;
}
```

Both arguments and the return value are structures in the function addpoint.

# Structures and Pointers

```
struct point *p;  /* p is a pointer to a structure
                     of type struct point */
struct point origin;

p = &origin;
printf("Origin is (%d, %d)\n", (*p).x, (*p).y);
```

- Parenthesis are necessary in `(*p).x` because the precedence of the structure member operator (dot) is higher than *.
- The expression `*p.x` ≡ `*(p.x)` which is illegal because `x` is not a pointer.

# Structures and Pointers

- Pointers to structures are so frequently used that an alternative is provided as a shorthand.
- If `p` is a pointer to a structure, then

```
p -> field_of_structure
```

  refers to a particular field.
- We could write

```
printf("Origin is (%d %d)\n", p->x, p->y);
```

# Assignments

```
struct student {
        char *last_name;
        int student_id;
        char grade;
};
struct student temp, *p = &temp;

temp.grade = ‘A’;
temp.last_name = “Casanova”;
temp.student_id = 590017;
```

| **Expression** | **Equiv. Expression** | **Value** |
|---|---|---|
| temp.grade | p -> grade | A |
| temp.last_name | p -> last_name | Casanova |
| temp.student_id | p -> student_id | 590017 |
| (*p).student_id | p -> student_id | 590017 |

# Structures and Pointers

- Both `.` and `->` associate from left to right
- Consider

```
struct rect r, *rp = &r;
```

- The following 4 expressions are equivalent.

```
r.pt1.x
rp -> pt1.x
(r.pt1).x
(rp->pt1).x
```

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

# Arrays of Structures

- Usually a program needs to work with more than one instance of data.
- For example, to maintain a list of phone #s in a program, you can define a structure to hold each person's name and number.

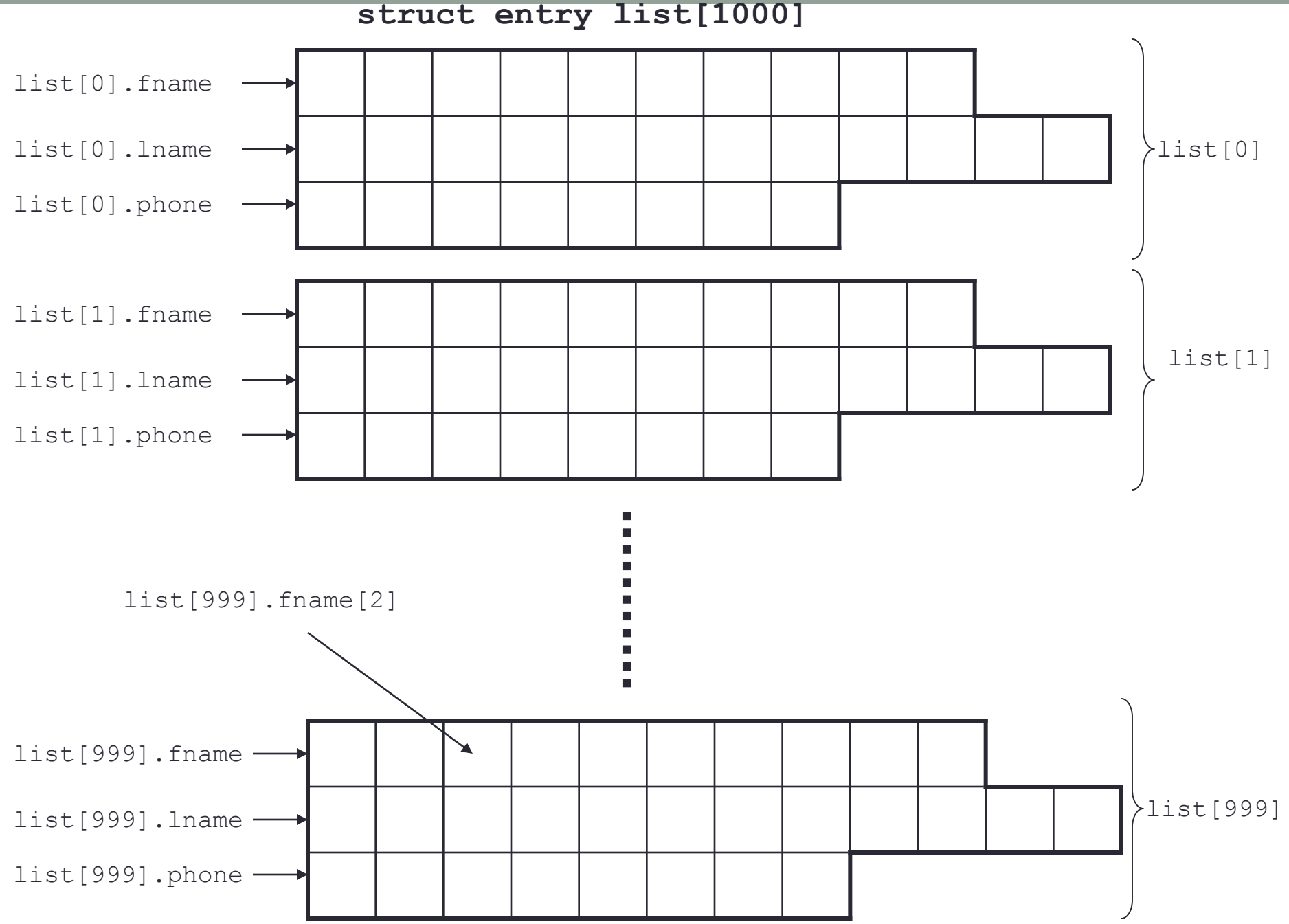
```
struct entry {
        char fname[10];
        char lname[12];
        char phone[8];
};
```

# Arrays of Structures

- A phone list has to hold many entries, so a single instance of the entry structure isn't of much use. What we need is an array of structures of type entry.

- After the structure has been defined, you can define the array as follows:

```
struct entry list[1000];
```


struct entry list[1000]
list[0].fname
list[0].lname
list[0].phone
list[0]
list[1].fname
list[1].lname
list[1].phone
list[1]
list[999].fname[2]
list[999].fname
list[999].lname
list[999].phone
list[999]

- To assign data in one element to another array element, you write

```
list[1] = list[5];
```

- To move data between individual structure fields, you write

```
strcpy(list[1].phone, list[5].phone);
```

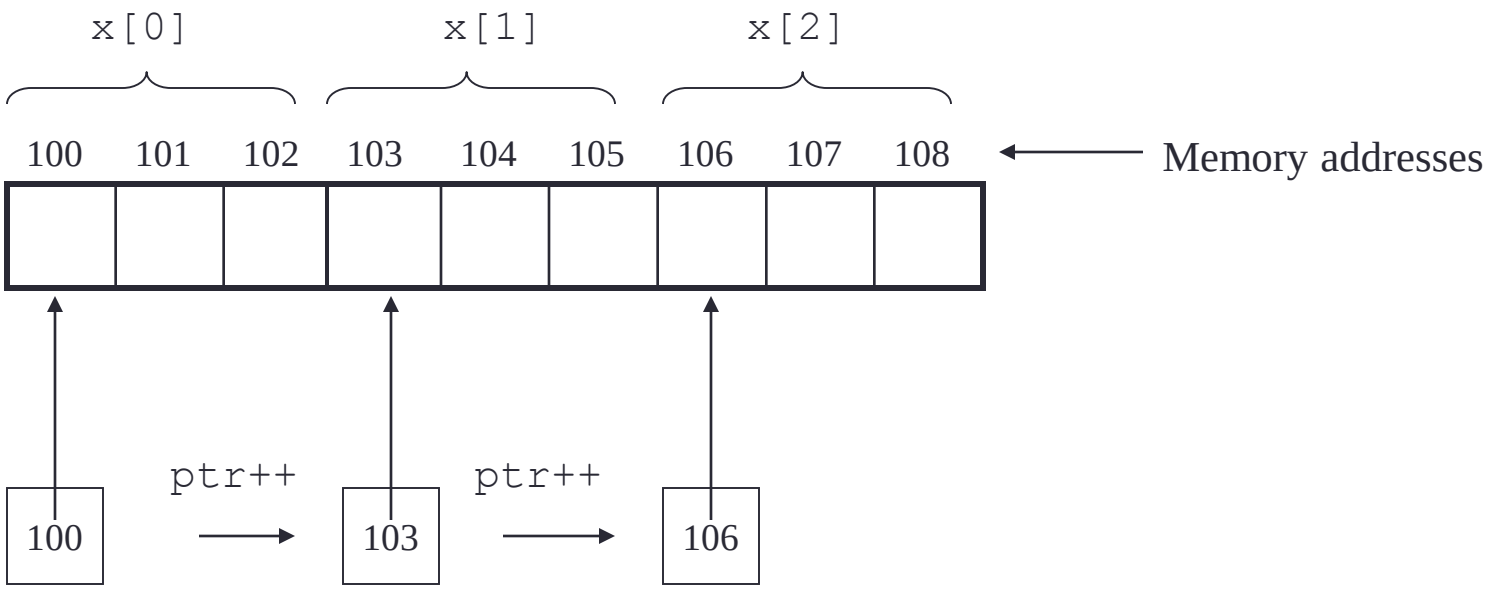
- To move data between individual elements of structure field arrays, you write

```
list[5].phone[1] = list[2].phone[3];
```

```
#define CLASS_SIZE 100
struct student {
        char *last_name;
        int student_id;
        char grade;
};

int main(void)
{
        struct student temp,
                class[CLASS_SIZE];

        ... /*Do some operation to fill class structure*/

        printf ("Number of A's in class: %d\n", countA(class));
}

int countA(struct student class[])
{
        int i, cnt = 0;
        for (i = 0; i < CLASS_SIZE; ++i)
                cnt += class[i].grade == 'A';
        return cnt;
}
```

• Arrays of structures can be very powerful programming tools, as can pointers to structures.

```
struct part {
       int number;
       char name [10];
};

struct part data[100];
struct part *p_part;

p_part = data;
printf("%d %s", p_part->number, p_part -> name);
```




- The above diagram shows an array named *x* that consists of 3 elements. The pointer *ptr* was initialized to point at x[0]. Each time *ptr* is incremented, it points at the next array element.

```
/* Array of structures */
#include <stdio.h>
#define MAX 4

struct part {
        int number;
        char name[10];
};

struct part data[MAX]= {1, “Smith”, 2, “Jones”, 3, “Adams”, 4, “Will”};

int main (void)
{
        struct part *p_part;
        int count;

        p_part = data;
        for (count = 0; count < MAX; count++) {
           printf(“\n %d %s”, p_part -> number, p_part -> name);
           p_part++;
        }
        return 0;
}
```

# Example: High-Performance Card Shuffling and Dealing Simulation

- The program in Fig. 10.3 is based on the card shuffling and dealing simulation discussed in Chapter 7.
- The program represents the deck of cards as an array of structures and uses high-performance shuffling and dealing algorithms.

```c
// Fig. 10.3: fig10_03.c
// Card shuffling and dealing program using structures
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define CARDS 52
#define FACES 13

// card structure definition
struct card {
   const char *face; // define pointer face
   const char *suit; // define pointer suit
};

typedef struct card Card; // new type name for struct card

// prototypes
void fillDeck(Card * const wDeck, const char * wFace[],
   const char * wSuit[]);
void shuffle(Card * const wDeck);
void deal(const Card * const wDeck);

```

**Fig. 10.3** | Card shuffling and dealing program using structures. (Part 1 of 4.)

```
24  int main(void)
25  {
26     Card deck[CARDS]; // define array of Cards
27
28     // initialize array of pointers
29     const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
30        "Six", "Seven", "Eight", "Nine", "Ten",
31        "Jack", "Queen", "King"};
32
33     // initialize array of pointers
34     const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};
35
36     srand(time(NULL)); // randomize
37
38     fillDeck(deck, face, suit); // load the deck with Cards
39     shuffle(deck); // put Cards in random order
40     deal(deck); // deal all 52 Cards
41  }
42
```

**Fig. 10.3** | Card shuffling and dealing program using structures. (Part 2 of 4.)

```
43  // place strings into Card structures
44  void fillDeck(Card * const wDeck, const char * wFace[],
45     const char * wSuit[])
46  {
47     // loop through wDeck
48     for (size_t i = 0; i < CARDS; ++i) {
49        wDeck[i].face = wFace[i % FACES];
50        wDeck[i].suit = wSuit[i / FACES];
51     }
52  }
53
54  // shuffle cards
55  void shuffle(Card * const wDeck)
56  {
57     // loop through wDeck randomly swapping Cards
58     for (size_t i = 0; i < CARDS; ++i) {
59        size_t j = rand() % CARDS;
60        Card temp = wDeck[i];
61        wDeck[i] = wDeck[j];
62        wDeck[j] = temp;
63     }
64  }
65
```

**Fig. 10.3** | Card shuffling and dealing program using structures. (Part 3 of 4.)

```
66  // deal cards
67  void deal(const Card * const wDeck)
68  {
69     // loop through wDeck
70     for (size_t i = 0; i < CARDS; ++i) {
71        printf("%5s of %-8s%s", wDeck[i].face  , wDeck[i].suit  ,
72           (i + 1) % 4 ? "  " : "\n");
73     }
74  }
```

**Fig. 10.3** | Card shuffling and dealing program using structures. (Part 4 of 4.)

```
Three of Hearts     Jack of Clubs     Three of Spades      Six of Diamonds
 Five of Hearts    Eight of Spades    Three of Clubs     Deuce of Spades
 Jack of Spades     Four of Hearts    Deuce of Hearts      Six of Clubs
Queen of Clubs     Three of Diamonds  Eight of Diamonds   King of Clubs
 King of Hearts    Eight of Hearts    Queen of Hearts    Seven of Clubs
Seven of Diamonds   Nine of Spades     Five of Clubs     Eight of Clubs
  Six of Hearts    Deuce of Diamonds   Five of Spades     Four of Clubs
Deuce of Clubs      Nine of Hearts    Seven of Hearts     Four of Spades
  Ten of Spades     King of Diamonds    Ten of Hearts     Jack of Diamonds
 Four of Diamonds    Six of Spades     Five of Diamonds    Ace of Diamonds
  Ace of Clubs      Jack of Hearts     Ten of Clubs      Queen of Diamonds
  Ace of Hearts      Ten of Diamonds   Nine of Clubs      King of Spades
  Ace of Spades     Nine of Diamonds  Seven of Spades    Queen of Spades
```

**Fig. 10.4** | Output for the high-performance card shuffling and dealing simulation.

# Unions

- `union`
  - Memory that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a `union` share space
  - Conserves storage
  - Only the last data member defined can be accessed

- `union` definitions
  - Same as struct

```
union Number {
  int x;
  float y;
};
union Number value;
```

# Unions

- Valid `union` operations
  - Assignment to `union` of same type: `=`
  - Taking address: &
  - Accessing union members: `.`
  - Accessing members using pointers: `->`

```
/* number union definition */
union number {
       int x;    /* define int x */
       double y; /* define double y */
}; /* end union number */

int main(){
        union number value; /* define union value */

        value.x = 100; /* put an integer into the union */

        printf("Put a value in the integer member.\n");
         printf(" int: %d\n double:%f\n\n", value.x, value.y );

         value.y = 100.0; /* put a double into the same union */

         printf("Put a value in the floating member.\n");
         printf(" int: %d\n double:%f\n\n", value.x, value.y );

         return 0; /* indicates successful termination */
} /* end main */
```

```
Put a value in the integer member.
int:   100
double:-
92559592117433136000000000000000000000000000000
00000000000000.000000

Put a value in the floating member.
int:   0
double: 100.000000
```

# FILE INPUT/OUTPUT

- In this chapter, you will learn:
  - To be able to create, read, write and update files.
  - To become familiar with sequential access file processing.
  - To become familiar with random-access file processing.

# Introduction

- Data files
  - Can be created, updated, and processed by C programs
  - Are used for permanent storage of large amounts of data
    - Storage of data in variables and arrays is only temporary

- When you use a file to store data for use by a program, that file usually consists of text (alphanumeric data) and is therefore called a **text file.**

# The Data Hierarchy

- Data Hierarchy:
  - Bit – smallest data item
    - Value of 0 or 1
  - Byte – 8 bits
    - Used to store a character
      - Decimal digits, letters, and special symbols
  - Field – group of characters conveying meaning
    - Example: your name
  - Record – group of related fields
    - Represented by a `struct` or a `class`
    - Example: In a payroll system, a record for a particular employee that contained his/her identification number, name, address, etc.

# The Data Hierarchy

- Data Hierarchy (continued):
  - File – group of related records
    - Example: payroll file
  - Database – group of related files

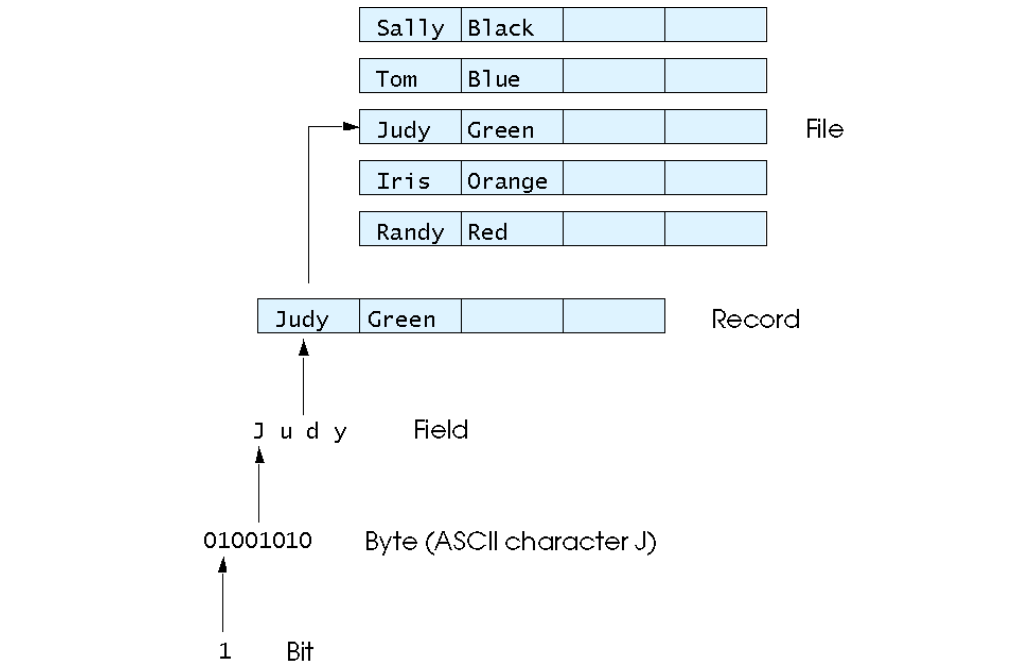



Fig. 11.1 The data hierarchy.

# Files and Streams

- C views each file as a sequence of bytes
  - File ends with the *end-of-file marker*
    - Or, file ends at a specified byte
- Stream created when a file is opened
  - Provide communication channel between files and programs
  - Opening a file returns a pointer to a `FILE` structure
    - Example file pointers:
    - `stdin` - standard input (keyboard)
    - `stdout` - standard output (screen)
    - `stderr` - standard error (screen)

# Files and Streams

- `FILE` structure
  - File descriptor
    - Index into operating system array called the open file table
  - File Control Block (FCB)
    - Operating system uses it to administer the file (file size, date, name, etc). This structure allows programs to open many files.

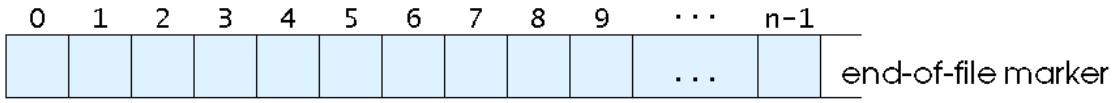



Fig. 11.2 C's view of a file of *n* bytes.

# Files and Streams

- Read/Write functions in standard library
  - `fscanf` / `fprintf`
    - File processing equivalents of `scanf` and `printf`
  - `fgetc`
    - Reads one character from a file
    - Takes a `FILE` pointer as an argument
    - `fgetc( stdin )` equivalent to `getchar()`
  - `fputc`
    - Writes one character to a file
    - Takes a `FILE` pointer and a character to write as an argument
    - `fputc( 'a', stdout )` equivalent to `putchar( 'a' )`
  - `fgets`
    - Reads a line from a file
  - `fputs`
    - Writes a line to a file

```c
/*
   Create a sequential file */
#include <stdio.h>

int main()
{
   int account;
   char name[ 30 ];
   double balance;
   FILE *cfPtr;   /* cfPtr = clients.dat file pointer */

   if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL )
      printf( "File could not be opened\n" );
   else {
      printf( "Enter the account, name, and balance.\n" );
      printf( "Enter EOF to end input.\n" );
      printf( "? " );
      scanf( "%d%s%lf", &account, name, &balance );

      while ( !feof( stdin ) ) {
         fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );

         printf( "? " );
         scanf( "%d%s%lf", &account, name, &balance );
      }

      fclose( cfPtr );
   }

   return 0;
}
```

# Program Output

```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

# Creating a Sequential Access File

- Creating a File
  - `FILE *myPtr;`
    - Creates a `FILE` pointer called `myPtr`
  - `myPtr = fopen(`*`filename, openmode`*`);`
    - Function `fopen` returns a `FILE` pointer to file specified
    - Takes two arguments – file to open and file open mode
    - If open fails, `NULL` returned

| Computer system | Key combination |
|---|---|
| UNIX systems | *<return> <ctrl> d* |
| IBM PC and compatibles | *<ctrl> z* |
| Macintosh | *<ctrl> d* |

Fig. 11.4 End-of-file key combinations for various popular computer systems.

# Creating a Sequential Access File

| Mode | Description |
|---|---|
| `r` | Open a file for reading. |
| `w` | Create a file for writing. If the file already exists, discard the current contents. |
| `a` | Append; open or create a file for writing at end of file. |
| `r+` | Open a file for update (reading and writing). |
| `w+` | Create a file for update. If the file already exists, discard the current contents. |
| `a+` | Append; open or create a file for update; writing is done at the end of the file. |
| `rb` | Open a file for reading in binary mode. |
| `wb` | Create a file for writing in binary mode. If the file already exists, discard the current contents. |
| `ab` | Append; open or create a file for writing at end of file in binary mode. |
| `rb+` | Open a file for update (reading and writing) in binary mode. |
| `wb+` | Create a file for update in binary mode. If the file already exists, discard the current contents. |
| `ab+` | Append; open or create a file for update in binary mode; writing is done at the end of the file. |

Fig. 11.6 File open modes.

# Creating a Sequential Access File

- `fprintf`
  - Used to print to a file
  - Like `printf`, except first argument is a `FILE` pointer (pointer to the file you want to print in)
- `feof( `*`FILE pointer`*` )`
  - Returns true if end-of-file indicator (no more data to process) is set for the specified file
- `fclose(` *`FILE pointer`* `)`
  - Closes specified file
  - Performed automatically when program ends
  - Good practice to close files explicitly
- Details
  - Programs may process no files, one file, or many files
  - Each file must have a unique name and should have its own pointer

# Reading Data from a File

- Reading a sequential access file
  - Create a `FILE` pointer, link it to the file to read

    ```
    myPtr = fopen( “myfile.dat", "r" );
    ```
  - Use `fscanf` to read from the file
    - Like `scanf`, except first argument is a `FILE` pointer

    ```
    fscanf( myPtr, "%d%s%f", &accounnt, name, &balance );
    ```
  - Data read from beginning to end
  - File position pointer
    - Indicates number of next byte to be read / written
    - Not really a pointer, but an integer value (specifies byte location)
    - Also called byte offset
  - `rewind( myPtr )`
    - Repositions file position pointer to beginning of file (byte 0)

```c

/* Reading and printing a sequential file */
#include <stdio.h>

int main()
{
   int account;
   char name[ 30 ];
   double balance;
   FILE *cfPtr;   /* cfPtr = clients.dat file pointer */

   if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL )
      printf( "File could not be opened\n" );
   else {
      printf( "%-10s%-13s%s\n", "Account", "Name", "Balance" );
      fscanf( cfPtr, "%d%s%lf", &account, name, &balance );

      while ( !feof( cfPtr ) ) {
         printf( "%-10d%-13s%7.2f\n", account, name, balance );
         fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
      }

      fclose( cfPtr );
   }

   return 0;
}
```

| Account | Name | Balance |
|---|---|---|
| 100 | Jones | 24.98 |
| 200 | Doe | 345.67 |
| 300 | White | 0.00 |
| 400 | Stone | -42.16 |
| 500 | Rich | 224.62 |

# Example: Merge two files

```
#include <stdio.h>
int main()
{   FILE *fileA,  /* first input file */
         *fileB,  /* second input file */
         *fileC; /*  output file to be created */
   int num1,  /* number to be read from first file */
       num2;  /* number to be read from second file */
   int f1, f2;

   /* Open files for processing */
   fileA = fopen("class1.txt","r");
   fileB = fopen("class2.txt","r");
   fileC = fopen("class.txt","w");
```

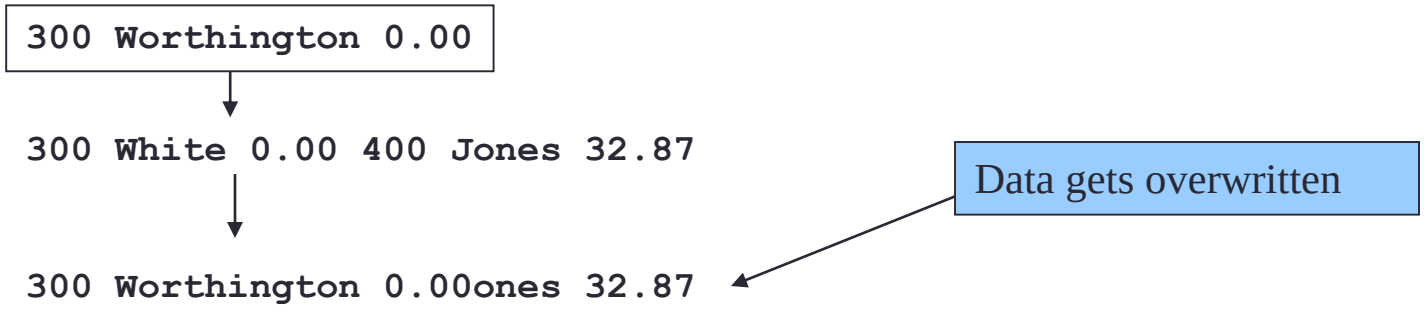
```
/* As long as there are numbers in both files, read and compare numbers one
by one. Write the smaller number to the output file and read the next number
in the file from which the smaller number is read. */

   f1 = fscanf(fileA, "%d", &num1);
   f2 = fscanf(fileB, "%d", &num2);

   while ((f1!=EOF) && (f2!=EOF)){
      if (num1 < num2){
         fprintf(fileC,"%d\n", num1);
         f1 = fscanf(fileA, "%d", &num1);
      }
      else if (num2 < num1) {
         fprintf(fileC,"%d\n", num2);
         f2 = fscanf(fileB, "%d", &num2);
      }
      else {  /* numbs are equal:read from both files */
         fprintf(fileC,"%d\n", num1);
         f1 = fscanf(fileA, "%d", &num1);
         f2 = fscanf(fileB, "%d", &num2);
      }
   }
```

```
   while (f1!=EOF){/* if reached end of second file, read
          the remaining numbers from first file and write to
          output file */
      fprintf(fileC,"%d\n", num1);
      f1 = fscanf(fileA, "%d", &num1);
   }
   while (f2!=EOF){ if reached the end of first file, read
          the remaining numbers from second file and write
          to output file */
      fprintf(fileC,"%d\n", num2);
      f2 = fscanf(fileB, "%d", &num2);
   }

   /* close files */
   fclose(fileA);
   fclose(fileB);
   fclose(fileC);
   return 0;
}  /* end of main */
```

# Reading Data from a Sequential Access File

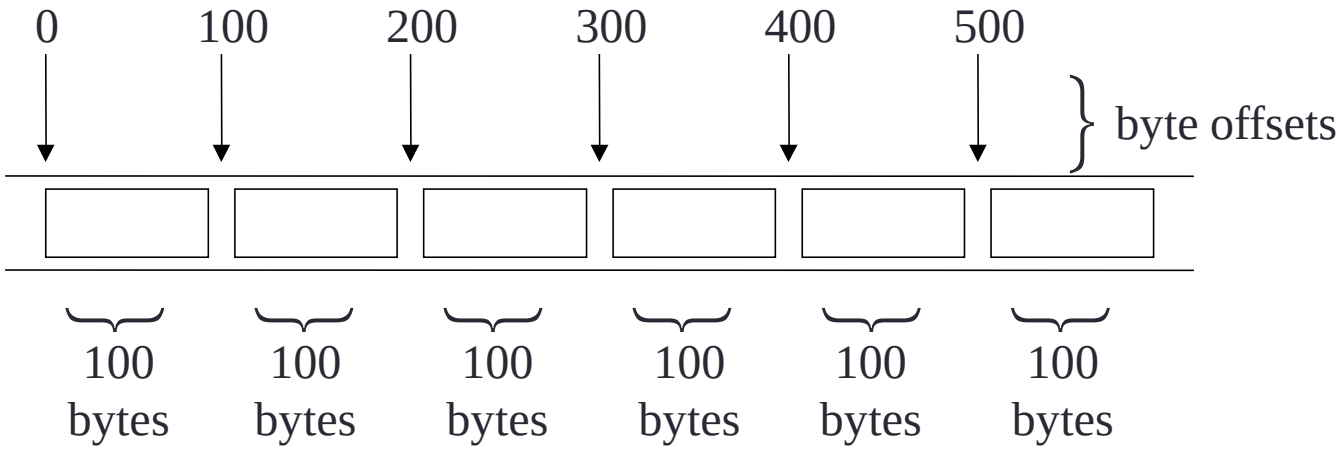
- Sequential access file
  - Cannot be modified without the risk of destroying other data
  - Fields can vary in size
    - Different representation in files and screen than internal representation
    - 1, 34, -890 are all `ints`, but have different sizes on disk

300 White 0.00 400 Jones 32.87 (old data in file)
If we want to change White's name to Worthington,



# Random-Access Files

- Random access files
  - Access individual records without searching through other records
  - Instant access to records in a file
  - Data can be inserted without destroying other data
  - Data previously stored can be updated or deleted without overwriting
- Implemented using fixed length records
  - Sequential files do not have fixed length records



# Creating a Randomly Accessed File

- Data in random access files
  - Unformatted (stored as "raw bytes")
    - All data of the same type (`int`s, for example) uses the same amount of memory
    - All records of the same type have a fixed length
    - Data not human readable

# Creating a Randomly Accessed File

- Unformatted I/O functions
  - `fwrite`
    - Transfer bytes from a location in memory to a file
  - `fread`
    - Transfer bytes from a file to a location in memory
  - Example:

    ```
    fwrite( &number, sizeof( int ), 1, myPtr );
    ```

    - `&number` – Location to transfer bytes from
    - `sizeof( int )` – Number of bytes to transfer
    - `1` – For arrays, number of elements to transfer
      - In this case, "one element" of an array is being transferred
    - `myPtr` – File to transfer to or from

# Creating a Randomly Accessed File

- Writing `structs`

```
fwrite( &myObject, sizeof (struct myStruct), 1, myPtr );
```

  - `sizeof` – returns size in bytes of object in parentheses
- To write several array elements
  - Pointer to array as first argument
  - Number of elements to write as third argument

```c
/* Fig. 11.11: fig11_11.c
   Creating a randomly accessed file sequentially */
#include <stdio.h>

/* clientData structure definition */
struct clientData {
   int acctNum;            /* account number */
   char lastName[ 15 ];  /* account last name */
   char firstName[ 10 ]; /* account first name */
    double balance;        /* account balance */
}; /* end structure clientData */

int main()
{
   int i; /* counter */

   /* create clientData with no information */
   struct clientData blankClient = { 0, "sevil", "sen", 5000.0 };

   FILE *cfPtr; /* credit.dat file pointer */
```

```
22     /* fopen opens the file; exits if file cannot be opened */
23     if ( ( cfPtr = fopen( "credit.dat", "wb" ) ) == NULL ) {
24        printf( "File could not be opened.\n" );
25     } /* end if */
26     else {
27
28        /* output 100 blank records to file */
29        for ( i = 1; i <= 100; i++ ) {
30           fwrite( &blankClient, sizeof( struct clientData ), 1, cfPtr );
31        } /* end for */
32
33        fclose ( cfPtr ); /* fclose closes the file */
34     } /* end else */
35
36     return 0; /* indicates successful termination */
37
38  } /* end main */
```

# Writing Data Randomly to a Randomly Accessed File

- `fseek`
  - Sets file position pointer to a specific position
  - `fseek( pointer, offset, symbolic_constant );`
    - *pointer* – pointer to file
    - *offset* – file position pointer (0 is first location)
    - *symbolic_constant* – specifies where in file we are reading from
    - `SEEK_SET` – seek starts at beginning of file
    - `SEEK_CUR` – seek starts at current location in file
    - `SEEK_END` – seek starts at end of file

```c
/* Fig. 11.12: fig11_12.c
   Writing to a random access file */
#include <stdio.h>

/* clientData structure definition */
struct clientData {
   int acctNum;           /* account number */
   char lastName[ 15 ];  /* account last name */
   char firstName[ 10 ]; /* account first name */
    double balance;       /* account balance */
}; /* end structure clientData */

int main()
{
   FILE *cfPtr; /* credit.dat file pointer */

   /* create clientData with no information */
   struct clientData client = { 0, "", "", 0.0 };

   /* fopen opens the file; exits if file cannot be opened */
   if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
      printf( "File could not be opened.\n" );
   } /* end if */
   else {

```

```
26        /* require user to specify account number */
27        printf( "Enter account number"
28                " ( 1 to 100, 0 to end input )\n? " );
29        scanf( "%d", &client.acctNum );
30
31        /* user enters information, which is copied into file */
32        while ( client.acctNum != 0 ) {
33
34           /* user enters last name, first name and balance */
35           printf( "Enter lastname, firstname, balance\n? " );
36
37           /* set record lastName, firstName and balance value */
38           fscanf( stdin, "%s%s%lf", client.lastName,
39                   client.firstName, &client.balance );
40
41           /* seek position in file of user-specified record */
42           fseek( cfPtr, ( client.acctNum - 1 ) *
43                  sizeof( struct clientData ), SEEK_SET );
44
45           /* write user-specified information in file */
46           fwrite( &client, sizeof( struct clientData ), 1, cfPtr );
47
48           /* enable user to specify another account number */
49           printf( "Enter account number\n? " );
50           scanf( "%d", &client.acctNum );
```

```
51        } /* end while */
52
53        fclose( cfPtr ); /* fclose closes the file */
54     } /* end else */
55
56     return 0; /* indicates successful termination */
57
58  } /* end main */
```

```
Enter account number ( 1 to 100, 0 to end input )
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

# Writing Data Randomly to a Randomly Accessed File

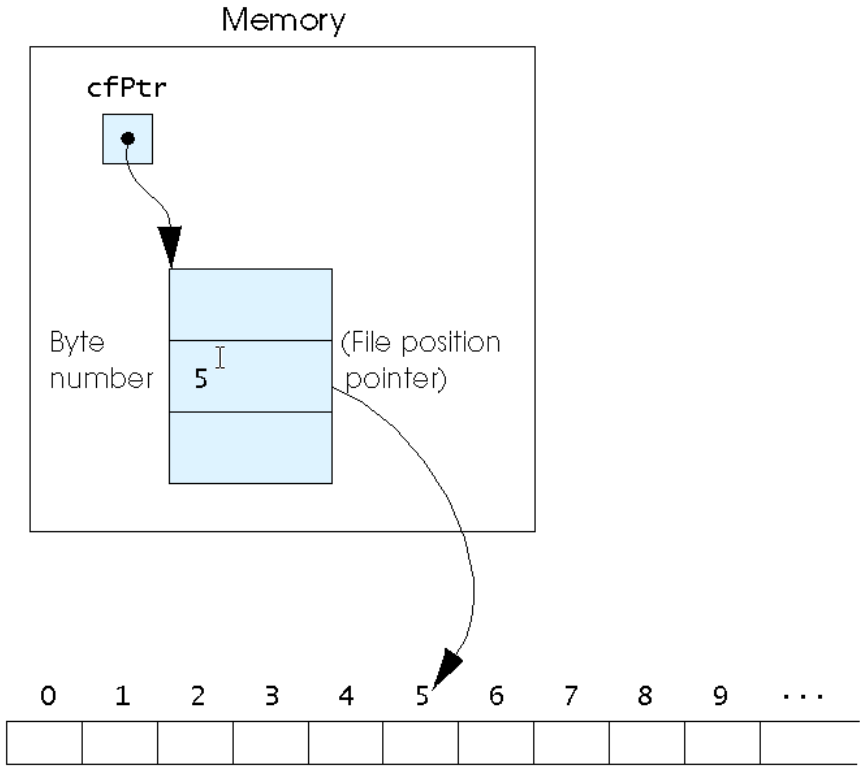



Fig. 11.14 The file position pointer indicating an offset of 5 bytes from the beginning of the file.

# Reading Data Randomly from a Randomly Accessed File

- `fread`
  - Reads a specified number of bytes from a file into memory

```
fread( &client, sizeof (struct clientData), 1, myPtr );
```

  - Can read several fixed-size array elements
    - Provide pointer to array
    - Indicate number of elements to read
  - To read multiple elements, specify in third argument

```c
/* Fig. 11.15: fig11_15.c
   Reading a random access file sequentially */
#include <stdio.h>

/* clientData structure definition */
struct clientData {
   int acctNum;           /* account number */
   char lastName[ 15 ];   /* account last name */
   char firstName[ 10 ]; /* account first name */
    double balance;        /* account balance */
}; /* end structure clientData */

int main()
{
   FILE *cfPtr; /* credit.dat file pointer */

   /* create clientData with no information */
   struct clientData client = { 0, "", "", 0.0 };

   /* fopen opens the file; exits if file cannot be opened */
   if ( ( cfPtr = fopen( "credit.dat", "rb" ) ) == NULL ) {
      printf( "File could not be opened.\n" );
   } /* end if */
```

```
24    else {
25       printf( "%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
26               "First Name", "Balance" );
27
28       /* read all records from file (until eof) */
29       while ( !feof( cfPtr ) ) {
30          fread( &client, sizeof( struct clientData ), 1, cfPtr );
31
32          /* display record */
33          if ( client.acctNum != 0 ) {
34             printf( "%-6d%-16s%-11s%10.2f\n",
35                     client.acctNum, client.lastName,
36                     client.firstName, client.balance );
37          } /* end if */
38
39       } /* end while */
40
41       fclose( cfPtr );/* fclose closes the file*/
42    } /* end else */
43
44    return 0;
45
46 } /* end main */
```

```
Acct  Last Name       First Name    Balance
29    Brown           Nancy          -24.54
33    Dunn            Stacey         314.33
37    Barker          Doug             0.00
88    Smith           Dave           258.34
96    Stone           Sam             34.98
```