# Chapter 11

# Structures and Unions

In Chap. 9 we studied the array, which is a data structure whose elements are all of the same data type. We now turn our attention to the *structure*, in which the individual elements can differ in type. Thus, a single structure might contain integer elements, floating-point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as *members*.

This chapter is concerned with the use of structures within a C program. We will see how structures are defined, and how their individual members are accessed and processed within a program. The relationships between structures and pointers, arrays and functions will also be examined.

Closely associated with the structure is the *union*, which also contains multiple members. Unlike a structure, however, the members of a union share the same storage area, even though the individual members may differ in type. Thus, a union permits several different data items to be stored in the same portion of the computer's memory at different times. We will see how unions are defined and utilized within a C program.

## 11.1 DEFINING A STRUCTURE

Structure declarations are somewhat more complicated than array declarations, since a structure must be defined in terms of its individual members. In general terms, the composition of a structure may be defined as

```
struct  tag  {
        member 1;
        member 2;
        . . . . .
        member m;
};
```

In this declaration, `struct` is a required keyword; *tag* is a name that identifies structures of this type (i.e., structures having this composition); and *member 1*, *member 2*, . . . , *member m* are individual member declarations. (*Note*: There is no formal distinction between a structure *definition* and a structure *declaration*; the terms are used interchangeably.)

The individual members can be ordinary variables, pointers, arrays, or other structures. The member names within a particular structure must be distinct from one another, though a member name can be the same as the name of a variable that is defined outside of the structure. A storage class, however, cannot be assigned to an individual member, and individual members cannot be initialized within a structure type declaration.

Once the composition of the structure has been defined, individual structure-type variables can be declared as follows:

*storage-class* `struct` *tag variable 1, variable 2, . . ., variable n*;

where *storage-class* is an optional storage class specifier, `struct` is a required keyword, *tag* is the name that appeared in the structure declaration, and *variable 1, variable 2, . . ., variable n* are structure variables of type *tag*.

**EXAMPLE 11.1** A typical structure declaration is shown below.

```
struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
};
```

This structure is named `account` (i.e., the tag is `account`). It contains four members: an integer quantity (`acct_no`), a single character (`acct_type`), an 80-element character array (`name[80]`), and a floating-point quantity (`balance`). The composition of this account is illustrated schematically in Fig. 11.1.

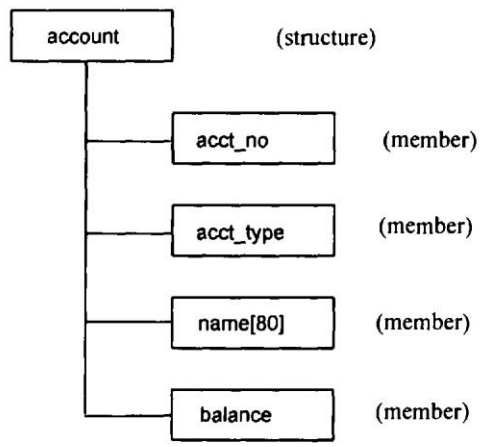



**Fig. 11.1**

We can now declare the structure variables `oldcustomer` and `newcustomer` as follows.

```
struct  account  oldcustomer, newcustomer;
```

Thus, `oldcustomer` and `newcustomer` are variables of type `account`. In other words, `oldcustomer` and `newcustomer` are structure-type variables whose composition is identified by the tag `account`.

It is possible to combine the declaration of the structure composition with that of the structure variables, as shown below.

```
storage-class  struct  tag  {
          member 1;
          member 2;
          . . . . .
          member m;
}  variable 1,  variable 2,  . . .,  variable n;
```

The *tag* is optional in this situation.

**EXAMPLE 11.2** The following single declaration is equivalent to the two declarations presented in the previous example.

```
struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
}  oldcustomer, newcustomer;
```

Thus, `oldcustomer` and `newcustomer` are structure variables of type `account`.

Since the variable declarations are now combined with the declaration of the structure type, the tag (i.e., `account`) need not be included. Thus, the above declaration can also be written as

```
struct  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
}  oldcustomer, newcustomer;
```

A structure variable may be defined as a member of another structure. In such situations, the declaration of the embedded structure must appear before the declaration of the outer structure.

**EXAMPLE 11.3** A C program contains the following structure declarations.

```
struct date  {
     int month;
     int day;
     int year;
};

struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
struct date lastpayment;
}  oldcustomer, newcustomer;
```

The second structure (`account`) now contains another structure (`date`) as one of its members. Note that the declaration of `date` precedes the declaration of `account`. The composition of `account` is shown schematically in Fig. 11.2.

The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. The initial values must appear in the order in which they will be assigned to their corresponding structure members, enclosed in braces and separated by commas. The general form is

*storage-class* `struct` *tag variable* = {*value 1*, *value 2*, . . ., *value m*};

where *value 1* refers to the value of the first member, *value 2* refers to the value of the second member, and so on. A structure variable, like an array, can be initialized only if its storage class is either `external` or `static`.

**EXAMPLE 11.4** This example illustrates the assignment of initial values to the members of a structure variable.

```
struct date  {
     int month;
     int day;
     int year;
};
```

```
struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
     struct date lastpayment;
};

static  struct  account  customer = {12345, 'R', "John W. Smith", 586.30, 5, 24, 90};
```

Thus, `customer` is a static structure variable of type `account`, whose members are assigned initial values. The first member (`acct_no`) is assigned the integer value `12345`, the second member (`acct_type`) is assigned the character `'R'`, the third member (`name[80]`) is assigned the string `"John W. Smith"`, and the fourth member (`balance`) is assigned the floating-point value `586.30`. The last member is itself a structure that contains three integer members (`month`, `day` and `year`). Therefore, the last member of `customer` is assigned the integer values `5`, `24` and `90`.

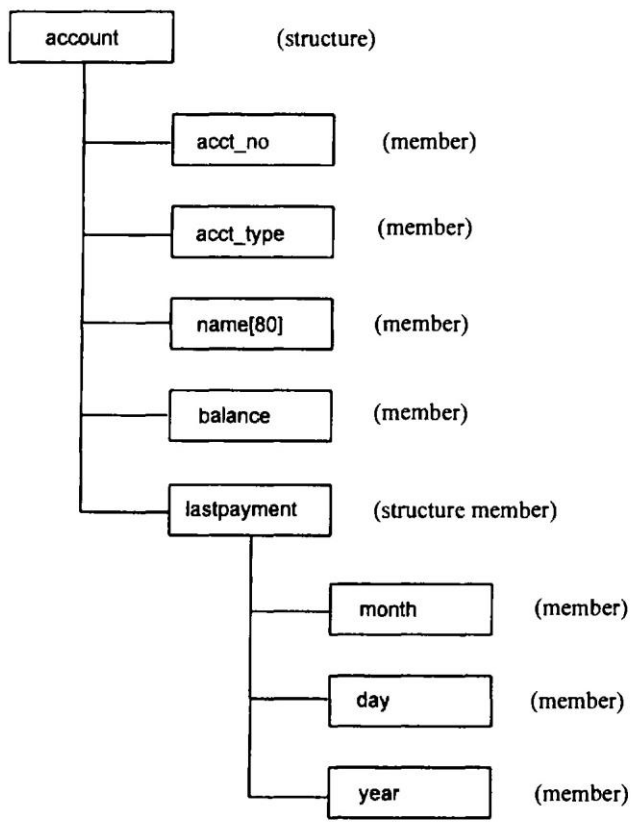



**Fig. 11.2**

It is also possible to define an array of structures; i.e., an array in which each element is a structure. The procedure is illustrated in the following example.

**EXAMPLE 11.5** A C program contains the following structure declarations.

```
struct date  {
     int month;
     int day;
     int year;
};

struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
     struct date lastpayment;
}  customer[100];
```

In this declaration `customer` is a 100-element array of structures. Hence, each element of `customer` is a separate structure of type `account` (i.e., each element of `customer` represents an individual customer record).

Note that each structure of type `account` includes an array (`name[80]`) and another structure (`date`) as members. Thus, we have an array and a structure embedded within another structure, which is itself an element of an array.

It is, of course, also permissible to define `customer` in a separate declaration, as shown below.

```
struct date  {
     int month;
     int day;
     int year;
};

struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
     struct date lastpayment;
};

struct  account  customer[100];
```

An array of structures can be assigned initial values just as any other array. Remember that each array element is a structure that must be assigned a corresponding set of initial values, as illustrated below.

**EXAMPLE 11.6** A C program contains the following declarations.

```
struct date  {
     char name[80];
     int month;
     int day;
     int year;
};

static  struct  date  birthday[] = {"Amy", 12, 30, 73,
                                    "Gail", 5, 13, 66,
                                    "Marc", 7, 15, 72,
                                    "Marla", 11, 29, 70,
                                    "Megan", 2, 4, 77,
                                    "Sharon", 12, 29, 63,
                                    "Susan", 4, 12, 69};
```

In this example `birthday` is an array of structures whose size is unspecified. The initial values will define the size of the array, and the amount of memory required to store the array.

Notice that each row in the variable declaration contains four constants. These constants represent the initial values, i.e., the name, month, day and year, for one array element. Since there are 7 rows (7 sets of constants), the array will contain 7 elements, numbered 0 to 6.

Some programmers may prefer to embed each set of constants within a separate pair of braces, in order to delineate the individual array elements more clearly. This is entirely permissible. Thus, the array declaration can be written

```
static  struct  date  birthday[] = {
                                      {"Amy", 12, 30, 73},
                                      {"Gail", 5, 13, 66},
                                      {"Marc", 7, 15, 72},
                                      {"Marla", 11, 29, 70},
                                      {"Megan", 2, 4, 77},
                                      {"Sharon", 12, 29, 63},
                                      {"Susan", 4, 12, 69}
                                    };
```

Remember that each structure is a self-contained entity with respect to member definitions. Thus, the same member name can be used in different structures to represent different data. In other words, the scope of a member name is confined to the particular structure within which it is defined.

**EXAMPLE 11.7** Two different structures, called `first` and `second`, are declared below.

```
struct first  {
     float a;
     int b;
     char c;
};

struct second  {
     char a;
     float b, c;
};
```

Notice that the individual member names `a`, `b` and `c` appear in both structure declarations, but the associated data types are different. Thus, `a` represents a floating-point quantity in `first` and a character in `second`. Similarly, `b` represents an integer quantity in `first` and a floating-point quantity in `second`, whereas `c` represents a character in `first` and a floating-point quantity in `second`. This duplication of member names is permissible, since the scope of each set of member definitions is confined to its respective structure. Within each structure the member names are distinct, as required.

## 11.2 PROCESSING A STRUCTURE

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

*variable.member*

where *variable* refers to the name of a structure-type variable, and *member* refers to the name of a member within the structure. Notice the period (.) that separates the variable name from the member name. This period is an operator; it is a member of the highest precedence group, and its associativity is left to right (see Appendix C).

**EXAMPLE 11.8** Consider the following structure declarations.

```
struct date  {
     int month;
     int day;
     int year;
};

struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
     struct date lastpayment;
}  customer;
```

In this example `customer` is a structure variable of type `account`. If we wanted to access the customer's account number, we would write

```
customer.acct_no
```

Similarly, the customer's name and the customer's balance can be accessed by writing

```
customer.name
```

and

```
customer.balance
```

Since the period operator is a member of the highest precedence group, this operator will take precedence over the unary operators as well as the various arithmetic, relational, logical and assignment operators. Thus, an expression of the form ++*variable.member* is equivalent to ++(*variable.member*); i.e., the ++ operator will apply to the structure member, not the entire structure variable. Similarly, the expression &*variable.member* is equivalent to &(*variable.member*); thus, the expression accesses the address of the structure member, not the starting address of the structure variable.

**EXAMPLE 11.9** Consider the structure declarations given in Example 11.8; i.e.,

```
struct date  {
     int month;
     int day;
     int year;
};

struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
     struct date lastpayment;
}  customer;
```

Several expressions involving the structure variable `customer` and its members are given below.

| *Expression* | *Interpretation* |
|---|---|
| `++customer.balance` | Increment the value of `customer.balance` |
| `customer.balance++` | Increment the value of `customer.balance` after accessing its value |
| `--customer.acct_no` | Decrement the value of `customer.acct_no` |
| `&customer` | Access the beginning address of `customer` |
| `&customer.acct_no` | Access the address of `customer.acctno` |

More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing

*variable.member.submember*

where *member* refers to the name of the member within the outer structure, and *submember* refers to the name of the member within the embedded structure. Similarly, if a structure member is an array, then an individual array element can be accessed by writing

*variable.member*[ *expression*]

where *expression* is a nonnegative value that indicates the array element.

**EXAMPLE 11.10** Consider once again the structure declarations presented in Example 11.8.

```
struct date  {
     int month;
     int day;
     int year;
};

struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
     struct date lastpayment;
}  customer;
```

The last member of `customer` is `customer.lastpayment`, which is itself a structure of type `date`. To access the month of the last payment, we would therefore write

```
customer.lastpayment.month
```

Moreover, this value can be incremented by writing

```
++customer.lastpayment.month
```

Similarly, the third member of `customer` is the character array `customer.name`. The third character within this array can be accessed by writing

```
customer.name[2]
```

This character's address can be obtained as

```
&customer.name[2]
```

The use of the period operator can be extended to arrays of structures, by writing

*array*[*expression*].*member*

where *array* refers to the array name, and *array*[*expression*] is an individual array element (a structure variable). Therefore *array*[*expression*].*member* will refer to a specific member within a particular structure.

**EXAMPLE 11.11** Consider the following structure declarations, which were originally presented in Example 11.5.

```
struct date  {
     int month;
     int day;
     int year;
};

struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
     struct date lastpayment;
}  customer[100];
```

In this example `customer` is an array that may contain as many as 100 elements. Each element is a structure of type `account`. Thus, if we wanted to access the account number for the 14th customer (i.e., `customer[13]`, since the subscripts begin with 0), we would write `customer[13].acct_no`. Similarly, this customer's balance can be accessed by writing `customer[13].balance`. The corresponding address can be obtained as `&customer[13].balance`.

The 14th customer's name can be accessed by writing `customer[13].name`. Moreover, we can access an individual character within the name by specifying a subscript. For example, the 8th character within the customer's name can be accessed by writing `customer[13].name[7]`. In a similar manner we can access the month, day and year of the 14th customer's last payment by specifying the individual members of `customer[13].lastpayment`, i.e., `customer[13].lastpayment.month`, `customer[13].lastpayment.day`, `customer[13].lastpayment.year`. Moreover, the expression `++customer[13].lastpayment.day` causes the value of the day to be incremented.

Structure members can be processed in the same manner as ordinary variables of the same data type. Single-valued structure members can appear in expressions, they can be passed to functions, and they can be returned from functions, as though they were ordinary single-valued variables. Complex structure members are processed in the same way as ordinary data items of that same type. For example, a structure member that is an array can be processed in the same manner as an ordinary array, and with the same restrictions. Similarly, a structure member that is itself a structure can be processed on a member-by-member basis (the members here refer to the embedded structure), the same as any other structure.

**EXAMPLE 11.12** Several statements or groups of statements that access individual structure members are shown below. All of the structure members conform to the declarations given in Example 11.8.

```
customer.balance = 0;

customer.balance -= payment;

customer.lastpayment.month = 12;

printf("Name: %s\n", customer.name);

if (customer.acct_type == 'P')
   printf("Preferred account no.: %d\n", customer.acct_no);
else
   printf("Regular account no.: %d\n", customer.acct_no);
```

The first statement assigns a value of zero to `customer.balance`, whereas the second statement causes the value of `customer.balance` to be decreased by the value of `payment`. The third statement causes the value 12 to be assigned to `customer.lastpayment.month`. Note that `customer.lastpayment.month` is a member of the embedded structure `customer.lastpayment`.

The fourth statement passes the array `customer.name` to the `printf` function, causing the customer name to be displayed. Finally, the last example illustrates the use of structure members in an `if - else` statement. Also, we see a situation in which the structure member `customer.acct_no` is passed to a function as an argument.

In some older versions of C, structures must be processed on a member-by-member basis. With this restriction, the only permissible operation on an entire structure is to take its address (more about this later). However, the current ANSI standard permits entire structures to be assigned to one another provided the structures have the same composition.

**EXAMPLE 11.13** Suppose `oldcustomer` and `newcustomer` are structure variables having the same composition; i.e.,

```
struct date  {
     int month;
     int day;
     int year;
};

struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
     struct date lastpayment;
}  oldcustomer, newcustomer;
```

as declared in Example 11.8. Let us assume that all of the members of `oldcustomer` have been assigned individual values. In most newer versions of C, it is possible to copy these values to `newcustomer` simply by writing

```
newcustomer = oldcustomer;
```

On the other hand, some older versions of C may require that the values be copied individually, member by member; for example,

```
newcustomer.acct_no = oldcustomer.acct_no;
newcustomer.acct_type = oldcustomer.acct_type;

. . . . .

newcustomer.lastpayment.year = oldcustomer.lastpayment.year;
```

It is also possible to pass entire structures to and from functions, though the way this is done varies from one version of C to another. Older versions of C allow only pointers to be passed, whereas the ANSI standard allows passing of the structures themselves. We will discuss this further in Sec. 11.5. Before moving on to the relationship between structures and pointers and the methods for passing structures to functions, however, let us consider a more comprehensive example that involves the processing of structure members.

**EXAMPLE 11.14 Updating Customer Records** To illustrate further how the individual members of a structure can be processed, consider a very simple customer billing system. In this system the customer records will be stored within an array of structures. Each record will be stored as an individual structure (i.e., as an array element) containing the customer's name, street address, city and state, account number, account status (current, overdue or delinquent), previous balance, current payment, new balance and payment date.

The overall strategy will be to enter each customer record into the computer, updating it as soon as it is entered, to reflect current payments. All of the updated records will then be displayed, along with the current status of each account. The account status will be based upon the size of the current payment relative to the customer's previous balance.

The structure declarations are shown below.

```
struct date  {
     int month;
     int day;
     int year;
};

struct account  {
     char name[80];
     char street[80];
     char city[80];
     int acct_no;
     char acct_type;
     float oldbalance;
     float newbalance;
     float payment;
     struct date lastpayment;
} customer[100];
```

Notice that `customer` is a 100-element array of structures. Thus, each array element (each structure) will represent one customer record. Each structure includes three members that are character-type arrays (`name`, `street` and `city`), and one member that is another structure (`lastpayment`).

The status of each account will be determined in the following manner:

1. If the current payment is greater than zero but less than 10 percent of the previous outstanding balance, the account will be overdue.
2. If there is an outstanding balance and the current payment is zero, the account will be delinquent.
3. Otherwise, the account will be current.

The overall program strategy will be as follows.

1. Specify the number of customer accounts (i.e., the number of structures) to be processed.
2. For each customer, read in the following items.
   (*a*) name
   (*b*) street
   (*c*) city
   (*d*) account number
   (*e*) previous balance
   (*f*) current payment
   (*g*) payment date
3. As each customer record is read into the computer, update it in the following manner.
   (*a*) Compare the current payment with the previous balance and determine the appropriate account status.
   (*b*) Calculate a new account balance by subtracting the current payment from the previous balance (a negative balance will indicate a credit).
4. After all of the customer records have been entered and processed, write out the following information for each customer.
   (*a*) name
   (*b*) account number
   (*c*) street
   (*d*) city
   (*e*) old balance
   (*f*) current payment
   (*g*) new balance
   (*h*) account status

Let us write the program in a modular manner, with one function to enter and update each record and another function to display the updated data. Ideally, we would like to pass every customer record (i.e., every array element) to each of these functions. Since each customer record is a structure, however, and we have not yet discussed how to pass a

structure to or from a function, we will define the array of structures as an external array. This will allow us to access the array elements, and the individual structure members, directly from all of the functions.

The individual program modules are straightforward, though some care is required in reading the individual structure members into the computer. Here is the entire program.

```
/* update a series of customer accounts (simplified billing system) */
/* maintain the customer accounts as an external array of structures */

#include <stdio.h>

void readinput(int i);
void writeoutput(int i);

struct date  {
     int month;
     int day;
     int year;
};

struct account  {
     char name[80];
     char street[80];
     char city[80];
     int acct_no;              /* (positive integer) */
     char acct_type;           /* C (current), O (overdue), or D (delinquent) */
     float oldbalance;         /* (nonnegative quantity) */
     float newbalance;         /* (nonnegative quantity) */
     float payment;            /* (nonnegative quantity) */
     struct date lastpayment;
} customer[100];               /* maintain as many as 100 customers */

main()

{
    int i, n;

    printf("CUSTOMER BILLING SYSTEM\n\n");
    printf("How many customers are there? ");
    scanf("%d", &n);

    for (i = 0; i < n; ++i)   {
        readinput(i);

        /* determine account status */

        if (customer[i].payment > 0)
           customer[i].acct_type =
              (customer[i].payment < 0.1 * customer[i].oldbalance) ? 'O' : 'C';
        else
           customer[i].acct_type =
              (customer[i].oldbalance > 0) ? 'D' : 'C';

        /* adjust account balance */

        customer[i].newbalance = customer[i].oldbalance - customer[i].payment;
    };

    for (i = 0; i < n; ++i)
        writeoutput(i);
}
```

```
void readinput(int i)

/* read input data and update record for each customer */

{
    printf("\nCustomer no. %d\n", i + 1);
    printf("   Name: ");
    scanf(" %[^\n]", customer[i].name);
    printf("   Street: ");
    scanf(" %[^\n]", customer[i].street);
    printf("   City: ");
    scanf(" %[^\n]", customer[i].city);
    printf("   Account number: ");
    scanf("%d", &customer[i].acct_no);
    printf("   Previous balance: ");
    scanf("%f", &customer[i].oldbalance);
    printf("   Current payment: ");
    scanf("%f", &customer[i].payment);
    printf("   Payment date (mm/dd/yyyy): ");
    scanf("%d/%d/%d", &customer[i].lastpayment.month,
                      &customer[i].lastpayment.day,
                      &customer[i].lastpayment.year);
    return;
}


void writeoutput(int i)

/* display current information for each customer */

{
    printf("\nName:   %s", customer[i].name);
    printf("     Account number: %d\n", customer[i].acct_no);
    printf("Street: %s\n", customer[i].street);
    printf("City:   %s\n\n", customer[i].city);
    printf("Old balance: %7.2f", customer[i].oldbalance);
    printf("   Current payment: %7.2f", customer[i].payment);
    printf("   New balance: %7.2f\n\n", customer[i].newbalance);
    printf("Account status: ");

    switch (customer[i].acct_type)   {
    case 'C':
                printf("CURRENT\n\n");
                break;
    case 'O':
                printf("OVERDUE\n\n");
                break;
    case 'D':
                printf("DELINQUENT\n\n");
                break;
    default:
                printf("ERROR\n\n");
    }
    return;
}
```

Now suppose the program is used to process four fictitious customer records. The input dialog is shown below, with the user's responses underlined.

```
CUSTOMER BILLING SYSTEM

How many customers are there? 4

Customer no. 1
    Name: Steve Johnson
    Street: 123 Mountainview Drive
    City: Denver, CO
    Account number: 4208
    Previous balance: 247.88
    Current payment: 25.00
    Payment date (mm/dd/yyyy): 6/14/1998

Customer no. 2
    Name: Susan Richards
    Street: 4383 Alligator Blvd
    City: Fort Lauderdale, FL
    Account number: 2219
    Previous balance: 135.00
    Current payment: 135.00
    Payment date (mm/dd/yyyy): 8/10/2000

Customer no. 3
    Name: Martin Peterson
    Street: 1787 Pacific Parkway
    City: San Diego, CA
    Account number: 8452
    Previous balance: 387.42
    Current payment: 35.00
    Payment date (mm/dd/yyyy): 9/22/1999

Customer no. 4
    Name: Phyllis Smith
    Street: 1000 Great White Way
    City: New York, NY
    Account number: 711
    Previous balance: 260.00
    Current payment: 0
    Payment date (mm/dd/yyyy): 11/27/2001
```

The program will then generate the following output data:

```
Name:    Steve Johnson      Account number: 4208
Street: 123 Mountainview Drive
City:    Denver, CO

Old balance:  247.88   Current payment:   25.00   New balance:  222.88

Account status: CURRENT
```

```
Name:   Susan Richards      Account number: 2219
Street: 4383 Alligator Blvd
City:   Fort Lauderdale, FL

Old balance:  135.00   Current payment:  135.00   New balance:    0.00

Account status: CURRENT


Name:   Martin Peterson     Account number: 8452
Street: 1787 Pacific Parkway
City:   San Diego, CA

Old balance:  387.42   Current payment:   35.00   New balance:  352.42

Account status: OVERDUE


Name:   Phyllis Smith      Account number: 711
Street: 1000 Great White Way
City:   New York, NY

Old balance:  260.00   Current payment:    0.00   New balance:  260.00

Account status: DELINQUENT
```

You should understand that this example is unrealistic from a practical standpoint, for two reasons. First, the array of structures (`customer`) is defined to be external to all of the functions within the program. It would be preferrable to declare `customer` within `main`, and then pass it to or from `readinput` or `writeouput` as required. We will learn how to do this in Sec. 11.5.

A more serious problem is the fact that a real customer billing system will store the customer records within a data file on an auxiliary memory device, such as a hard disk or a magnetic tape. To update a record we would access the record from the data file, change the data where necessary, and then write the updated record back to the data file. The use of data files will be discussed in Chap. 12. Since the present example does not make use of data files, we must reenter all of the customer records whenever the program is run. This is rather contrived, though it does provide a simple example illustrating the manner in which structures can be processed on a member-by-member basis.

It is sometimes useful to determine the number of bytes required by an array or a structure. This information can be obtained through the use of the `sizeof` operator, originally discussed in Sec. 3.2. For example, the size of a structure can be determined by writing either `sizeof` *`variable`* or `sizeof (struct` *`tag`*`)`.

**EXAMPLE 11.15** An elementary C program is shown below.

```
#include <stdio.h>

main()   /* determine the size of a structure */
{
    struct date  {
         int month;
         int day;
         int year;
    };
```

```
    struct account  {
         int acct_no;
         char acct_type;
         char name[80];
         float balance;
         struct date lastpayment;
    }  customer;

    printf("%d\n", sizeof customer);
    printf("%d", sizeof (struct account));
}
```

This program makes use of the `sizeof` operator to determine the number of bytes associated with the structure variable `customer` (or equivalently, the structure `account`). The two `printf` statements illustrate different ways to utilize the `sizeof` operator. Both `printf` statements will produce the same output.

Execution of the program will result in the following output.

```
93
93
```

Thus, the structure variable `customer` (or the structure `account`) will occupy 93 bytes. This value is obtained as follows.

| *Structure member* | *Number of bytes* |
|---|---|
| `acct_no` | 2 |
| `acct_type` | 1 |
| `name` | 80 |
| `balance` | 4 |
| `lastpayment` | 6 |
| Total | 93 |

Some compilers may assign two bytes to `acct_type` in order to maintain an even number of bytes. Hence, the total byte count may be 94 rather than 93.

## 11.3 USER-DEFINED DATA TYPES (`typedef`)

The `typedef` feature allows users to define new data-types that are equivalent to existing data types. Once a user-defined data type has been established, then new variables, arrays, structures, etc. can be declared in terms of this new data type.

In general terms, a new data type is defined as

```
typedef  type  new-type;
```

where *type* refers to an existing data type (either a standard data type, or previous user-defined data type), and *new-type* refers to the new user-defined data type. It should be understood, however, that the new data type will be new in name only. In reality, this new data type will not be fundamentally different from one of the standard data types.

**EXAMPLE 11.16** Here is a simple declaration involving the use of `typedef`.

```
typedef  int  age;
```

In this declaration `age` is a user-defined data type, which is equivalent to type `int`. Hence, the variable declaration

```
age  male, female;
```

is equivalent to writing

```
int  male, female;
```

In other words, `male` and `female` are regarded as variables of type `age`, though they are actually integer-type variables.

Similarly, the declarations

```
typedef  float  height[100];
height  men, women;
```

define `height` as a 100-element, floating-point array type—hence, `men` and `women` are 100-element, floating-point arrays. Another way to express this is

```
typedef  float  height;
height  men[100], women[100];
```

though the former declaration is somewhat simpler.

The `typedef` feature is particularly convenient when defining structures, since it eliminates the need to repeatedly write `struct` *tag* whenever a structure is referenced. Hence, the structure can be referenced more concisely. In addition, the name given to a user-defined structure type often suggests the purpose of the structure within the program.

In general terms, a user-defined structure type can be written as

```
typedef  struct  {
     member 1;
     member 2;
     . . . . .
     member m;
}  new-type;
```

where *new-type* is the user-defined structure type. Structure variables can then be defined in terms of the new data type.

**EXAMPLE 11.17** The following declarations are comparable to the structure declarations presented in Examples 11.1 and 11.2. Now, however, we introduce a user-defined data type to describe the structure.

```
typedef  struct  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
}  record;

record oldcustomer, newcustomer;
```

The first declaration defines `record` as a user-defined data type. The second declaration defines `oldcustomer` and `newcustomer` as structure variables of type `record`.

The `typedef` feature can be used repeatedly, to define one data type in terms of other user-defined data types.

**EXAMPLE 11.18** Here are some variations of the structure declarations presented in Example 11.5.

```
typedef struct  {
        int month;
        int day;
        int year;
}  date;

typedef struct  {
        int acct_no;
        char acct_type;
        char name[80];
        float balance;
        date lastpayment;
}  record;

record customer[100];
```

In this example `date` and `record` are user-defined stucture types, and `customer` is a 100-element array whose elements are structures of type `record`. (Recall that `date` was a tag rather than an actual data type in Example 11.5.) The individual members within the *i*th element of `customer` can be written as `customer[i].acct_no`, `customer[i].name`, `customer[i].lastpayment.month`, etc., as before.

There are, of course, variations on this theme. Thus, an alternate declaration can be written as

```
typedef struct  {
        int month;
        int day;
        int year;
}  date;

typedef struct  {
        int acct_no;
        char acct_type;
        char name[80];
        float balance;
        date lastpayment;
}  record[100];

record customer;
```

or simply

```
typedef struct  {
        int month;
        int day;
        int year;
}  date;

struct  {
        int acct_no;
        char acct_type;
        char name[80];
        float balance;
        date lastpayment;
}  customer[100];
```

All three sets of declarations are equivalent.

## 11.4 STRUCTURES AND POINTERS

The beginning address of a structure can be accessed in the same manner as any other address, through the use of the address (&) operator. Thus, if *variable* represents a structure-type variable, then *&variable* represents the starting address of that variable. Moreover, we can declare a pointer variable for a structure by writing

```
type  *ptvar;
```

where *type* is a data type that identifies the composition of the structure, and *ptvar* represents the name of the pointer variable. We can then assign the beginning address of a structure variable to this pointer by writing

```
ptvar = &variable;
```

**EXAMPLE 11.19** Consider the following structure declaration, which is a variation of the declaration presented in Example 11.1.

```
typedef struct  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
}  account;

account  customer, *pc;
```

In this example `customer` is a structure variable of type `account`, and `pc` is a pointer variable whose object is a structure variable of type `account`. Thus, the beginning address of `customer` can be assigned to `pc` by writing

```
pc = &customer;
```

The variable and pointer declarations can be combined with the structure declaration by writing

```
struct  {
     member 1;
     member 2;
     . . . . .
     member m;
}  variable,  *ptvar;
```

where *variable* again represents a structure-type variable, and *ptvar* represents the name of a pointer variable.

**EXAMPLE 11.20** The following single declaration is equivalent to the two declarations presented in the previous example.

```
struct  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
}  customer, *pc;
```

The beginning address of `customer` can be assigned to `pc` by writing

```
pc = &customer;
```

as in the previous example.

An individual structure member can be accessed in terms of its corresponding pointer variable by writing

*ptvar->member*

where *ptvar* refers to a structure-type pointer variable and the operator -> is comparable to the period (.) operator discussed in Sec. 11.2. Thus, the expression

*ptvar->member*

is equivalent to writing

*variable.member*

where *variable* is a structure-type variable, as discussed in Sec. 11.2. The operator -> falls into the highest precedence group, like the period operator (.). Its associativity is left to right (see Appendix C).

The -> operator can be combined with the period operator to access a submember within a structure (i.e., to access a member of a structure that is itself a member of another structure). Hence, a submember can be accessed by writing

*ptvar->member.submember*

Similarly, the -> operator can be used to access an element of an array that is a member of a structure. This is accomplished by writing

*ptvar->member*[*expression*]

where *expression* is a nonnegative integer that indicates the array element.

**EXAMPLE 11.21** Here is a variation of the declarations shown in Example 11.8.

```
typedef struct  {
     int month;
     int day;
     int year;
}  date;

struct  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
     date lastpayment;
}  customer, *pc = &customer;
```

Notice that the pointer variable pc is initialized by assigning it the beginning address of the structure variable `customer`. In other words, pc will point to `customer`.

If we wanted to access the customer's account number, we could write any of the following:

```
customer.acct_no        pc->acct_no        (*pc).acct_no
```

The parentheses are required in the last expression because the period operator has a higher precedence than the indirection operator (*). Without the parentheses the compiler would generate an error, because pc (a pointer) is not directly compatible with the dot operator.

Similarly, the customer's balance can be accessed by writing any of the following:

```
customer.balance      pc->balance      (*pc).balance
```

and the month of the last payment can be accessed by writing any of the following:

```
customer.lastpayment.month      pc->lastpayment.month      (*pc).lastpayment.month
```

Finally, the customer's name can be accessed by writing any of the following:

```
customer.name          pc->name          (*pc).name
```

Therefore, the third character of the customer's name can be accessed by writing any of the following (see Sec. 10.4).

```
customer.name[2]          pc->name[2]          (*pc).name[2]
*(customer.name + 2)      pc->(name + 2)       *((*pc).name + 2)
```

A structure can also include one or more pointers as members. Thus, if *ptmember* is both a pointer and a member of *variable*, then **variable.ptmember* will access the value to which *ptmember* points. Similarly, if *ptvar* is a pointer variable that points to a structure and *ptmember* is a member of that structure, then **ptvar->ptmember* will access the value to which *ptmember* points.

**EXAMPLE 11.22** Consider the simple C program shown below.

```
#include <stdio.h>

main()

{
    int n = 3333;
    char t = 'C';
    float b = 99.99;

    typedef struct  {
         int month;
         int day;
         int year;
    }  date;

    struct  {
         int *acct_no;
         char *acct_type;
         char *name;
         float *balance;
         date lastpayment;
    }  customer, *pc = &customer;

    customer.acct_no = &n;
    customer.acct_type = &t;
    customer.name = "Smith";
    customer.balance = &b;

    printf("%d %c %s %.2f\n", *customer.acct_no, *customer.acct_type,
                              customer.name, *customer.balance);
    printf("%d %c %s %.2f", *pc->acct_no, *pc->acct_type,
                            pc->name, *pc->balance);
}
```

Within the second structure, the members `acct_no`, `acct_type`, `name` and `balance` are written as pointers. Thus, the value to which `acct_no` points can be accessed by writing either `*customer.acct_no` or `*pc->acct_no`. The same is true for `acct_type` and `balance`. Moreover, recall that a string can be assigned directly to a character-type pointer. Therefore, if `name` points to the beginning of a string, then the string can be accessed by writing either `customer.name` or `pc->name`.

Execution of this simple program results in the following two lines of output.

```
3333 C Smith 99.99
3333 C Smith 99.99
```

The two lines of output are identical, as expected.

Since the `->` operator is a member of the highest precedence group, it will be given the same high priority as the period (`.`) operator, with left-to-right associativity. Moreover, this operator, like the period operator, will take precedence over any unary, arithmetic, relational, logical or assignment operators that may appear in an expression. We have already discussed this point, as it applies to the period operator, in Sec. 11.2. However, some additional consideration should be given to certain unary operators, such as ++, as they apply to structure-type pointer variables.

We already know that expressions such as *++ptvar->member* and *++ptvar->member.submember* are equivalent to *++(ptvar->member)* and *++(ptvar->member.submember)*, respectively. Thus, such expressions will cause the value of the member or the submember to be incremented, as discussed in Sec. 11.2. On the other hand, the expression *++ptvar* will cause the value of *ptvar* to increase by whatever number of bytes is associated with the structure to which *ptvar* points. (The number of bytes associated with a particular structure can be determined through the use of the `sizeof` operator, as illustrated in Example 11.15.) Hence, the address represented by *ptvar* will change as a result of this expression. Similarly, the expression *(++ptvar).member* will cause the value of *ptvar* to increase by this number of bytes before accessing *member*. There is some danger in attempting operations like these, because *ptvar* may no longer point to a structure variable once its value has been altered.

**EXAMPLE 11.23** Here is a variation of the simple C program shown in Example 11.15.

```
#include <stdio.h>

main()
{
    typedef struct  {
         int month;
         int day;
         int year;
    }  date;

    struct  {
         int acct_no;
         char acct_type;
         char name[80];
         float balance;
         date lastpayment;
    }  customer, *pt = &customer;

    printf("Number of bytes (dec): %d\n", sizeof *pt);
    printf("Number of bytes (hex): %x\n\n", sizeof *pt);
    printf("Starting address (hex): %x\n", pt);
    printf("Incremented address (hex): %x", ++pt);
}
```

Notice that pt is a pointer variable whose object is the structure variable customer.

The first printf statement causes the number of bytes associated with customer to be displayed as a decimal quantity. The second printf statement displays this same value as a hexadecimal quantity. The third printf statement causes the value of pt (i.e., the starting address of customer) to be displayed in hexadecimal, whereas the fourth printf statement shows what happens when pt is incremented.

Execution of the program causes the following output to be generated.

```
Number of bytes (dec): 93
Number of bytes (hex): 5d

Starting address (hex): f72
Incremented address (hex): fcf
```

Thus, we see that customer requires 93 decimal bytes, which is 5d in hexadecimal. The initial value assigned to pt (i.e., the starting address of customer) is f72, in hexadecimal. When pt is incremented, its value increases by 5d hexadecimal bytes, to fcf.

It is interesting to alter this program by replacing the character array member name[80] with the character pointer *name, and then execute the program. What do you think will happen?

## 11.5 PASSING STRUCTURES TO FUNCTIONS

There are several different ways to pass structure-type information to or from a function. Structure members can be transferred individually, or entire structures can be transferred. The mechanics for carrying out the transfers vary, depending on the type of transfer (individual members or complete structures) and the particular version of C.

Individual structure members can be passed to a function as arguments in the function call, and a single structure member can be returned via the return statement. To do so, each structure member is treated the same as an ordinary single-valued variable.

**EXAMPLE 11.24** The skeletal outline of a C program is shown below. This outline makes use of the structure declarations presented earlier.

```
float adjust(char name[], int acct_no, float balance);      /* funct prototype */

main()
{
    typedef struct  {                   /* structure declaration */
         int month;
         int day;
         int year;
    }  date;

    struct  {                           /* structure declaration */
         int acct_no;
         char acct_type;
         char name[80];
         float balance;
         date lastpayment;
    }  customer;

    . . . . .

    customer.balance = adjust(customer.name, customer.acct_no, customer.balance);

    . . . . .
}
```

```
float adjust(char name[], int acct_no, float balance)

{
    float newbalance;                    /* local variable declaration */

    . . . . .

    newbalance = . . . . .;              /* adjust value of balance */

    . . . . .

    return(newbalance);
}
```

This program outline illustrates the manner in which structure members can be passed to a function. In particular, `customer.name`, `customer.acct_no` and `customer.balance` are passed to the function `adjust`. Within `adjust`, the value assigned to `newbalance` presumably makes use of the information passed to the function. This value is then returned to `main`, where it is assigned to the structure member `customer.balance`.

Notice the function declaration in `main`. This declaration could also have been written without the argument names, as follows:

```
float adjust(char [], int, float);
```

Some programmers prefer this form, since it avoids the specification of dummy argument names for data items that are actually structure members. We will continue to utilize full function prototypes, however, to take advantage of the resulting error checking.

A complete structure can be transferred to a function by passing a structure-type pointer as an argument. In principle, this is similar to the procedure used to transfer an array to a function. However, we must use explicit pointer notation to represent a structure that is passed as an argument.

You should understand that a structure passed in this manner will be passed by *reference* rather than by *value*. Hence, if any of the structure members are altered within the function, the alterations will be recognized outside of the function. Again, we see a direct analogy with the transfer of arrays to a function.

**EXAMPLE 11.25** Consider the simple C program shown below.

```
#include <stdio.h>

typedef struct {
     char *name;
     int acct_no;
     char acct_type;
     float balance;
} record;

main()   /* transfer a structure-type pointer to a function */
{
    void adjust(record *pt);        /* function declaration */

    static record customer = {"Smith", 3333, 'C', 33.33};

    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
                              customer.acct_type, customer.balance);
    adjust(&customer);
    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
                              customer.acct_type, customer.balance);
}
```

```
void adjust(record *pt)          /* function definition */

{
    pt->name = "Jones";
    pt->acct_no = 9999;
    pt->acct_type = 'R';
    pt->balance = 99.99;
    return;
}
```

This program illustrates the transfer of a structure to a function by passing the structure's address (a pointer) to the function. In particular, `customer` is a static structure of type `record`, whose members are assigned an initial set of values. These initial values are displayed when the program begins to execute. The structure's address is then passed to the function `adjust`, where different values are assigned to the members of the structure.

Within `adjust`, the formal argument declaration defines `pt` as a pointer to a structure of type `record`. Also, notice the empty `return` statement; i.e., nothing is explicitly returned from `adjust` to `main`.

Within `main`, we see that the current values assigned to the members of `customer` are displayed a second time, after `adjust` has been accessed. Thus, the program illustrates whether or not the changes made in `adjust` carry over to the calling portion of the program.

Execution of the program results in the following output.

```
Smith 3333 C 33.33
Jones 9999 R 99.99
```

Thus, the values assigned to the members of `customer` within `adjust` are recognized within `main`, as expected.

A pointer to a structure can be returned from a function to the calling portion of the program. This feature may be useful when several structures are passed to a function, but only one structure is returned.

**EXAMPLE 11.26 Locating Customer Records** Here is a simple C program that illustrates how an array of structures is passed to a function, and how a pointer to a particular structure is returned.

Suppose we specify an account number for a particular customer and then locate and display the complete record for that customer. Each customer record will be maintained in a structure, as in the last example. Now, however, the entire set of customer records will be stored in an array called `customer`. Each element of `customer` will be an independent structure.

The basic strategy will be to enter an account number, and then transfer both the account number and the array of records to a function called `search`. Within `search`, the specified account number will be compared with the account number that is stored within each customer record until a match is found, or until the entire list of records has been searched. If a match is found, a pointer to that array element (the structure containing the desired customer record) is returned to `main`, and the contents of the record are displayed.

If a match is not found after searching the entire array, then the function returns a value of `NULL` (zero) to `main`. The program then displays an error message requesting that the user reenter the account number. This overall search procedure will continue until a value of zero is entered for the account number.

The complete program is shown below. Within this program, `customer` is an array of structures of type `record`, and `pt` is a pointer to a structure of this same type. Also, `search` is a function that accepts two arguments and returns a pointer to a structure of type `record`. The arguments are an array of structures of type `record` and an integer quantity, respectively. Within `search`, the quantity returned is either the address of an array element, or `NULL` (zero).

```
/* find a customer record that corresponds to a specified account number */

#include <stdio.h>

#define N  3
#define NULL  0
```

```
typedef struct {
     char *name;
     int acct_no;
     char acct_type;
     float balance;
} record;

record *search(record table[], int acctn);     /* function prototype */

main()

{
    static record customer[N] = {
                                    {"Smith", 3333, 'C', 33.33},
                                    {"Jones", 6666, 'O', 66.66},
                                    {"Brown", 9999, 'D', 99.99}
                                };                  /* array of structures */

    int acctn;                                      /* variable declaration */
    record *pt;                                     /* pointer declaration */

    printf("Customer Account Locator\n");
    printf("To END, enter 0 for the account number\n");
    printf("\nAccount no.: ");              /* enter first account number */
    scanf("%d", &acctn);

    while (acctn != 0)   {
        pt = search(customer, acctn);

        if (pt != NULL)   {          /* found a match */
           printf("\nName: %s\n", pt->name);
           printf("Account no.: %d\n", pt->acct_no);
           printf("Account type: %c\n", pt->acct_type);
           printf("Balance: %.2f\n", pt->balance);
        }
        else
           printf("\nERROR - Please try again\n");

        printf("\nAccount no.: ");         /* enter next account number */
        scanf("%d", &acctn);
    }
}


record *search(record table[N], int acctn)      /* function definition */

/* accept an array of structures and an account number,
   return a pointer to a particular structure (an array element)
   if the account number matches a member of that structure */

{
    int count;

    for (count = 0; count < N; ++count)
        if (table[count].acct_no == acctn)              /* found a match */
           return(&table[count]);      /* return pointer to array element */

    return(NULL);
}
```

The array size is expressed in terms of the symbolic constant N. For this simple example we have selected a value of N = 3. That is, we are storing only three sample records within the array. In a more realistic example, N would have a much greater value.

Finally, it should be mentioned that there are much better ways to search through a set of records than examining each record sequentially. We have selected this simple though inefficient procedure in order to concentrate on the mechanics of transferring structures between main and its subordinate function search.

Shown below is a typical dialog that might result from execution of the program. The user's responses are underlined, as usual.

```
Customer Account Locator
To END, enter 0 for the account number

Account no.: 3333

Name: Smith
Account no.: 3333
Account type: C
Balance: 33.33

Account no.: 9999

Name: Brown
Account no.: 9999
Account type: D
Balance: 99.99

Account no.: 666

ERROR - Please try again

Account no.: 6666

Name: Jones
Account no.: 6666
Account type: O
Balance: 66.66

Account no.: 0
```

Newer versions of C permit an entire structure to be transferred directly to a function as an argument, and returned directly from a function via the return statement. (Notice the contrast with arrays, which *cannot* be returned via the return statement.) These features are included in the current ANSI standard.

When a structure is passed directly to a function, the transfer is by value rather than by reference. This is consistent with other direct (nonpointer) transfers in C. Therefore, if any of the structure members are altered within the function, the alterations *will not* be recognized outside of the function. However, if the altered structure is returned to the calling portion of the program, then the changes *will* be recognized within this broader scope.

**EXAMPLE 11.27** In Example 11.25 we saw a program that transferred a structure-type pointer to a function. Two different printf statements within main illustrated the fact that transfers of this type are by reference; i.e., alterations made to the structure within the function are recognized within main. A similar program is shown below. However, the present program transfers a complete structure, rather than a structure-type pointer, to the function.

```
#include <stdio.h>

typedef struct {
     char *name;
     int acct_no;
     char acct_type;
     float balance;
}  record;

void adjust(record customer);          /* function prototype*/

main()   /* transfer a structure to a function */

{
    static record customer = {"Smith", 3333, 'C', 33.33};

    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
                              customer.acct_type, customer.balance);
    adjust(customer);
    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
                              customer.acct_type, customer.balance);
}


void adjust(record cust)         /* function definition */

{
    cust.name = "Jones";
    cust.acct_no = 9999;
    cust.acct_type = 'R';
    cust.balance = 99.99;
    return;
}
```

Notice that the function `adjust` now accepts a structure of type `record` as an argument, rather than a pointer to a structure of type `record`, as in Example 11.25. Nothing is returned from `adjust` to `main` in either program.

When the program is executed, the following output is obtained.

```
Smith 3333 C 33.33
Smith 3333 C 33.33
```

Thus, the new assignments made within `adjust` are not recognized within `main`. This is expected, since the transfer of the structure `customer` from `main` to `adjust` is by value rather than by reference. (Compare with the output shown in Example 11.25.)

Now suppose we modify this program so that the altered structure is returned from `adjust` to `main`. Here is the modified program.

```
#include <stdio.h>

typedef struct {
     char *name;
     int acct_no;
     char acct_type;
     float balance;
}  record;

    record adjust(record customer);        /* function prototype */
```

```
main()   /* transfer a structure to a function and return the structure */
{
    static record customer = {"Smith", 3333, 'C', 33.33};

    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
                              customer.acct_type, customer.balance);
    customer = adjust(customer);
    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
                              customer.acct_type, customer.balance);
}


record adjust(record cust)                /* function definition */
{
    cust.name = "Jones";
    cust.acct_no = 9999;
    cust.acct_type = 'R';
    cust.balance = 99.99;
    return(cust);
}
```

Notice that `adjust` now returns a structure of type `record` to `main`. The `return` statement is modified accordingly.

Execution of this program results in the following output.

```
Smith 3333 C 33.33
Jones 9999 R 99.99
```

Thus, the alterations that were made within `adjust` are now recognized within `main`. This is expected, since the altered structure is now returned directly to the calling portion of the program. (Compare with the output shown in Example 11.25 as well as the output shown earlier in this example.)

Most versions of C allow complicated data structures to be transferred freely between functions. We have already seen examples involving the transfer of individual structure members, entire structures, pointers to structures and arrays of structures. As a practical matter, however, there are some limitations on the complexity of data structures that can easily be transferred to or from a function. In particular, some compilers may have difficulty executing programs that involve complex data structure transfers, because of certain memory restrictions. Beginning programmers should be aware of these limitations, though the details of this topic are beyond the scope of the current text.

**EXAMPLE 11.28 Updating Customer Records** Example 11.14 presented a simple customer billing system illustrating the use of structures to maintain and update customer records. In that example the customer records were stored within a global (external) array of structures. We now consider two variations of that program. In each program the array of structures is maintained locally, within `main`. The individual array elements (i.e., individual customer records) are transferred back and forth between functions, as required.

In the first program, complete structures are transferred between the functions. In particular, the function `readinput` allows information defining each customer record to be entered into the computer. When an entire record has been entered, the corresponding structure is returned to `main`, where it is stored within the 100-element array called `customer` and adjusted for the proper account type. After all the records have been entered and adjusted, they are transferred individually to the function `writeoutput`, where certain information is displayed for each customer.

The entire program is shown below.

```
/* update a series of customer accounts (simplified billing system) */

#include <stdio.h>

/* maintain the customer accounts as an array of structures,
   transfer complete structures to and from functions              */

typedef struct  {
     int month;
     int day;
     int year;
}  date;

typedef struct  {
     char name[80];
     char street[80];
     char city[80];
     int acct_no;          /* (positive integer) */
     char acct_type;       /* C (current), O (overdue), or D (delinquent) */
     float oldbalance;     /* (nonnegative quantity) */
     float newbalance;     /* (nonnegative quantity) */
     float payment;        /* (nonnegative quantity) */
     date lastpayment;
}  record;

record readinput(int i);              /* function prototype */
void writeoutput(record customer);    /* function prototype */

main()

/* read customer accounts, process each account, and display output */

{
    int i, n;                  /* variable declarations */
    record customer[100];      /* array declaration (array of structures) */

    printf("CUSTOMER BILLING SYSTEM\n\n");
    printf("How many customers are there? ");
    scanf("%d", &n);

    for (i = 0; i < n; ++i)   {
        customer[i] = readinput(i);

        /* determine account status */

        if (customer[i].payment > 0)
            customer[i].acct_type =
              (customer[i].payment < 0.1 * customer[i].oldbalance) ? 'O' : 'C';
        else
            customer[i].acct_type =
              (customer[i].oldbalance > 0) ? 'D' : 'C';

        /* adjust account balance */

        customer[i].newbalance = customer[i].oldbalance - customer[i].payment;
    }

    for (i = 0; i < n; ++i)
        writeoutput(customer[i]);
}
```

```
record readinput(int i)

/* read input data for a customer */

{
    record customer;

    printf("\nCustomer no. %d\n", i + 1);
    printf("   Name: ");
    scanf(" %[^\n]", customer.name);
    printf("   Street: ");
    scanf(" %[^\n]", customer.street);
    printf("   City: ");
    scanf(" %[^\n]", customer.city);
    printf("   Account number: ");
    scanf("%d", &customer.acct_no);
    printf("   Previous balance: ");
    scanf("%f", &customer.oldbalance);
    printf("   Current payment: ");
    scanf("%f", &customer.payment);
    printf("   Payment date (mm/dd/yyyy): ");
    scanf("%d/%d/%d", &customer.lastpayment.month,
                      &customer.lastpayment.day,
                      &customer.lastpayment.year);
    return(customer);
}

void writeoutput(record customer)

/* display current information for a customer */

{
    printf("\nName:   %s", customer.name);
    printf("     Account number: %d\n", customer.acct_no);
    printf("Street: %s\n", customer.street);
    printf("City:   %s\n\n", customer.city);
    printf("Old balance: %7.2f", customer.oldbalance);
    printf("   Current payment: %7.2f", customer.payment);
    printf("   New balance: %7.2f\n\n", customer.newbalance);
    printf("Account status: ");

    switch (customer.acct_type)   {
       case 'C':
                 printf("CURRENT\n\n");
                 break;
       case 'O':
                 printf("OVERDUE\n\n");
                 break;
       case 'D':
                 printf("DELINQUENT\n\n");
                 break;
       default:
                 printf("ERROR\n\n");
    }
    return;
}
```

The next program is very similar to the previous program. Now, however, the transfers involve pointers to structures rather than the structures themselves. Thus, the structures are now transferred by reference, whereas they were transferred by value in the previous program.

For brevity, this program is outlined rather than listed in its entirety. The missing blocks are identical to the corresponding portions of the previous program.

```
/* update a series of customer accounts (simplified billing system) */

#include <stdio.h>

/* maintain the customer accounts as an array of structures,
   transfer pointers to structures to and from functions    */

/* (structure definitions) */

record *readinput(int i);          /* function prototype */
void writeoutput(record *cust);    /* function prototype */

main()

/* read customer accounts, process each account, and display output */

{
    int i, n;                  /* variable declarations */
    record customer[100];      /* array declaration (array of structures) */

    . . . . .

    for (i = 0; i < n; ++i)   {
        customer[i] = *readinput(i);

        /* determine account status */

        . . . . .

        /* adjust account balance */

        . . . . .
    }

    for (i = 0; i < n; ++i)
        writeoutput(&customer[i]);
}


record *readinput(int i)

/* read input data for a customer */

{
    record customer;

    /* enter input data */

    return(&customer);
}
```

```
void writeoutput(record *pt)

/* display current information for a customer */

{
    record customer;

    customer = *pt;

    /* display output data */

    return;
}
```

Both of these programs will behave in the same manner as the program given in Example 11.14 when executed. Because of the complexity of the data structure (i.e., the array of structures, where each structure contains embedded arrays and embedded structures), however, the compiled programs may not be executable with certain compilers. In particular, a stack overflow condition (a type of inadequate memory condition) may be experienced with some compilers.

This problem would not exist if the program were more realistic; i.e., if the customer records were stored within a file on an auxiliary memory device, rather than in an array that is stored within the computer's memory. We will discuss this problem in Chap. 12, where we consider the use of data files for situations such as this.

## 11.6 SELF-REFERENTIAL STRUCTURES

It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type. In general terms, this can be expressed as

```
struct  tag  {
     member 1;
     member 2;
     . . . . .
     struct  tag  *name;
};
```

where *name* refers to the name of a pointer variable. Thus, the structure of type *tag* will contain a member that points to another structure of type *tag*. Such structures are known as *self-referential* structures.

**EXAMPLE 11.29** A C program contains the following structure declaration.

```
struct  list_element  {
          char item[40];
          struct  list_element  *next;
};
```

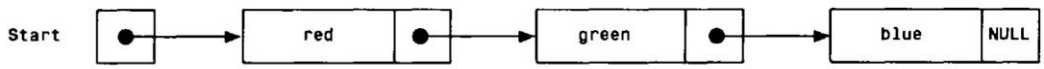
This is a structure of type `list_element`. The structure contains two members: a 40-element character array, called `item`, and a pointer to a structure of the same type (i.e., a pointer to a structure of type `list_element`), called `next`. Therefore this is a self-referential structure.

Self-referential structures are very useful in applications that involve linked data structures, such as lists and trees. We will see a comprehensive example illustrating the processing of a linked list in Example 11.32. First, however, we present a brief summary of linked data structures.

The basic idea of a linked data structure is that each component within the structure includes a pointer indicating where the next component can be found. Therefore, the relative order of the components can easily be changed simply by altering the pointers. In addition, individual components can easily be added or deleted,

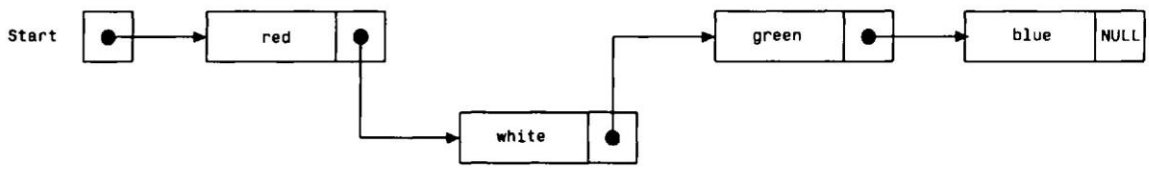
again by altering the pointers. As a result, a linked data structure is not confined to some maximum number of components. Rather, the data structure can expand or contract in size as required.

**EXAMPLE 11.30** Figure 11.3(*a*) illustrates a linked list containing three components. Each component consists of two data items: a string, and a pointer that references the next component within the list. Thus, the first component contains the string `red`, the second contains `green` and the third contains `blue`. The beginning of the list is indicated by a separate pointer, which is labeled `start`. Also, the end of the list is indicated by a special pointer, called `NULL`.

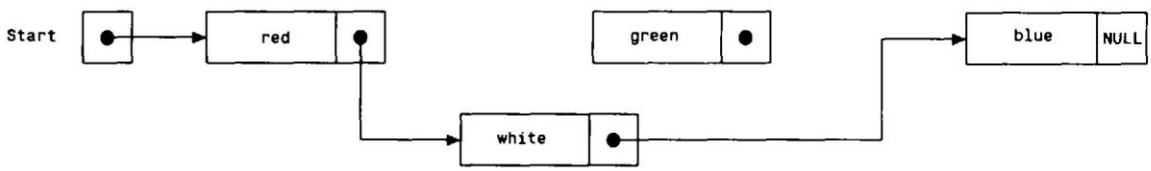



(*a*)

Now let us add another component, whose value is `white`, between `red` and `green`. To do so we merely change the pointers, as illustrated in Fig. 11.3(*b*). Similarly, if we choose to delete the component whose value is `green`, we simply change the pointer associated with the second component, as shown in Fig. 11.3(*c*).




(*b*)




(*c*)

**Fig. 11.3**

There are several different kinds of linked data structures, including *linear* linked lists, in which the components are all linked together sequentially; linked lists with *multiple pointers*, which permit forward and backward traversal within the list; *circular* linked lists, which have no beginning and no ending; and *trees*, in which the components are arranged in a hierarchical structure. We have already seen an illustration of a linear linked list in Example 11.30. Other kinds of linked lists are illustrated in the next example.

**EXAMPLE 11.31** Figure 11.4 shows a linear linked list that is similar to that shown in Fig. 11.3(*a*). Now, however, we see that there are *two* pointers associated with each component: a forward pointer, and a backward pointer. This double set of pointers allows us to traverse the list in either direction, i.e., from beginning to end, or from end to beginning.

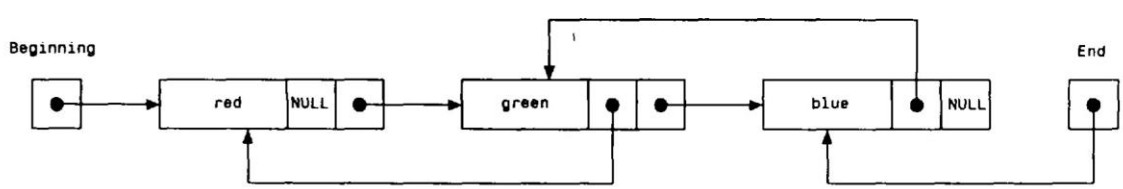



**Fig. 11.4**

Now consider the list shown in Fig. 11.5. This list is similar to that shown in Fig. 11.3(*a*), except that the last data item (`blue`) points to the first data item (`red`). Hence, this list has no beginning and no ending. Such lists are referred to as *circular lists*.

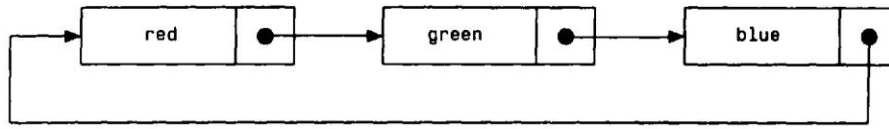
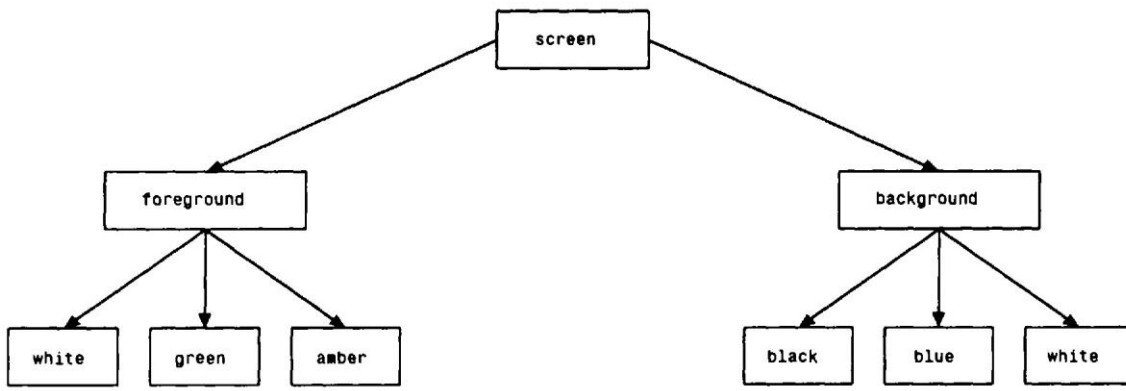



**Fig. 11.5**

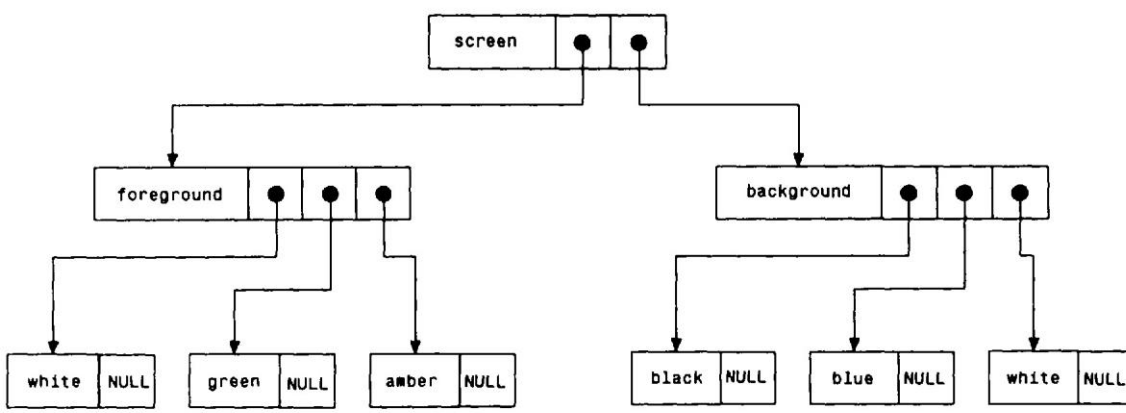
Finally, in Fig. 11.6(*a*) we see an example of a *tree*. Trees consist of nodes and branches, arranged in some hierarchical manner which indicates a corresponding hierarchical structure within the data. (A *binary tree* is a tree in which every node has no more than two branches.)

In Fig. 11.6(*a*) the *root node* has the value `screen`, and the associated branches lead to the nodes whose values are `foreground` and `background`, respectively. Similarly, the branches associated with `foreground` lead to the nodes whose values are `white`, `green` and `amber`, and the branches associated with `background` lead to the nodes whose values are `black`, `blue` and `white`.

Figure 11.6(*b*) illustrates the manner in which pointers are used to construct the tree.




(*a*)




(*b*)

**Fig. 11.6**

Self-referential structures are ideally suited for applications involving linked data structures. Each structure will represent a single component (i.e., one node) within the linked data structure. The self-referential pointer will indicate the location of the next component.

**EXAMPLE 11.32 Processing a Linked List** We now present an interactive C program that allows us to create a linear linked list, add a new component to the linked list, or delete an existing component from the linked list. Each component will consist of a string, and a pointer to the next component. The program will be menu-driven to facilitate its use by nonprogrammers. We will include a provision to display the list after the selection of any menu item (i.e., after any change has been made to the list).

This program is somewhat more complex than the preceding example programs. It utilizes both *recursion* (see Sec. 7.6) and *dynamic memory allocation* (see Sec. 10.5, and Examples 10.15, 10.22, 10.24 and 10.26).

The entire program is shown below. Following the program listing, the individual functions are discussed in detail.

```
/* menu-driven program to process a linked list of strings */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define  NULL  0

struct list_element  {
          char item[40];                  /* data item for this node */
          struct list_element *next;      /* pointer to the next node */
};

typedef struct list_element node;         /* structure type declaration */

int menu(void);                           /* function prototype */
void create(node *pt);                    /* function prototype */
node *insert(node *pt);                   /* function prototype */
node *remove(node *pt);                   /* function prototype */
void display(node *pt);                   /* function prototype */


main()

{
    node *start;                          /* pointer to beginning of list*/
    int choice;                           /* local variable declaration */

    do   {
           choice = menu();
           switch (choice)  {

           case 1:     /* create the linked list */
                start = (node *) malloc(sizeof(node));   /* allocate space, 1st node */
                create(start);
                printf("\n");
                display(start);
                continue;

           case 2:     /* add one component */
                start = insert(start);
                printf("\n");
                display(start);
                continue;
```

```
            case 3:     /* delete one component */
                 start = remove(start);
                 printf("\n");
                 display(start);
                 continue;

            default:    /* terminate computation */
                 printf("End of computation\n");
            }
    }  while (choice != 4);
}


int menu(void)     /* generate the main menu */

{
    int choice;

    do   {
           printf("\nMain menu:\n");
           printf("  1 - CREATE the linked list\n");
           printf("  2 - ADD a component\n");
           printf("  3 - DELETE a component\n");
           printf("  4 - END\n");
           printf("Please enter your choice (1, 2, 3 or 4) -> ");
           scanf("%d", &choice);
           if (choice < 1 || choice > 4)
              printf("\nERROR - Please try again\n");
    }  while (choice < 1 || choice > 4);
    printf("\n");
    return(choice);
}


void create(node *record)     /* create a linked list */

/* argument points to the current node */

{
    printf("Data item (type \'END\' when finished): ");
    scanf(" %[^\n]", record->item);

    if (strcmp(record->item, "END") == 0)
       record->next = NULL;
    else   {
       /* allocate space for next node */
       record->next = (node *) malloc(sizeof(node));

       /* create the next node */
       create(record->next);
    }
    return;
}


void display(node *record)     /* display the linked list */

/* argument points to the current node */
```

```
{
    if (record->next != NULL)   {
       printf("%s\n", record->item);          /* display this data item */
       display(record->next);                 /* get the next data item */
    }
    return;
}


node *insert(node *first)     /* add one component to the linked list
                                 return a pointer to beginning of the modified list */
/* argument points to the first node */
{
    node *locate(node*, char[]);    /* function declaration */
    node *newrecord;                /* pointer to new node */
    node *tag;                      /* pointer to node BEFORE target node */
    char newitem[40];               /* new data item */
    char target[40];                /* data item following the new entry */

    printf("New data item: ");
    scanf(" %[^\n]", newitem);
    printf("Place before (type \'END\' if last): ");
    scanf(" %[^\n]", target);

    if (strcmp(first->item, target) == 0)   {
       /* new node is first in list */

       /* allocate space for the new node */
       newrecord = (node *) malloc(sizeof(node));

       /* assign the new data item to newrecord->item */
       strcpy(newrecord->item, newitem);

       /* assign the current pointer to newrecord->next */
       newrecord->next = first;

       /* new pointer becomes the beginning of the list */
       first = newrecord;
    }

    else   {
       /* insert new node after an existing node */

       /* locate the node PRECEDING the target node */
       tag = locate(first, target);

       if (tag == NULL)
          printf("\nMatch not found - Please try again\n");
       else   {
          /* allocate space for the new node */
          newrecord = (node *) malloc(sizeof(node));

          /* assign the new data item to newrecord->item */
          strcpy(newrecord->item, newitem);

          /* assign the next pointer to newrecord->next */
          newrecord->next = tag->next;
```

```
            /* assign the new pointer to tag->next */
            tag->next = newrecord;
        }
    }
    return(first);
}

node *locate(node *record, char target[])      /* locate a node */

/* return a pointer to the node BEFORE the target node
   The first argument points to the current node
   The second argument is the target string           */

{
    if (strcmp(record->next->item, target) == 0)       /* found a match */
        return(record);
    else
        if (record->next->next == NULL)                /* end of list */
            return(NULL);
        else
            locate(record->next, target);              /* try next node */
}

node *remove(node *first)    /* remove (delete) one component from the linked list
                                return a pointer to beginning of the modified list */

/* argument points to the first node */

{
    node *locate(node*, char[]);      /* function declaration */
    node *tag;                        /* pointer to node BEFORE target node */
    node *temp;                       /* temporary pointer */
    char target[40];                  /* data item to be deleted */

    printf("Data item to be deleted: ");
    scanf(" %[^\n]", target);

    if (strcmp(first->item, target) == 0)   {
        /* delete the first node */

        /* mark the node following the target node */
        temp = first->next;

        /* free space for the target node */
        free(first);

        /* adjust the pointer to the first node */
        first = temp;
    }

    else   {
        /* delete a data item other than the first */

        /* locate the node PRECEDING the target node */
        tag = locate(first, target);

        if (tag == NULL)
            printf("\nMatch not found - Please try again\n");
```

```
            else   {
               /* mark the node following the target node */
               temp = tag->next->next;

               /* free space for the target node */
               free(tag->next);

               /* adjust the link to the next node */
               tag->next = temp;
            }
         }
         return(first);
}
```

The program begins with the usual `#include` statements and a definition of the symbolic constant `NULL` to represent the value `0`. Following these statements is a declaration for the self-referential structure `list_element`. This structure declaration is the same as that shown in Example 11.29. Thus, `list_element` identifies a structure consisting of two members: a 40-element character array (`item`), and a pointer (`next`) to another structure of the same type. The character array will represent a string, and the pointer will identify the location of the next component in the linked list.

The data type `node` is then defined, identifying structures having composition `list_element`. This definition is followed by the function prototypes. Within the function prototypes, notice that `start` is a pointer to a structure of type `node`. This pointer will indicate the beginning of the linked list. The remaining function prototypes identify several additional functions that are called from `main`. Note that these declarations and function prototypes are external. They will therefore be recognized throughout the program.

The `main` function consists of a `do - while` loop that permits repetitious execution of the entire process. This loop calls the function `menu`, which generates the main menu, and returns a value for `choice`, indicating the user's menu selection. A `switch` statement then calls the appropriate functions, in accordance with the user's selection. Notice that the program will stop executing if `choice` is assigned a value of 4.

If `choice` is assigned a value of 1, indicating that a new linked list will be created, a block of memory must be allocated for the first data item before calling the function `create`. This is accomplished using the library function `malloc`, as discussed in Sec. 10.5. Thus, memory allocation statement

```
start = (node *) malloc(sizeof(node));
```

reserves a block of memory whose size (in bytes) is sufficient for one node. The statement returns a pointer to a structure of type `node`. This pointer indicates the beginning of the linked list. Thus, it is passed to `create` as an argument.

Note that the type cast `(node *)` is required as a part of the memory allocation statement. Without it, the `malloc` function would return a pointer to a `char` rather than a pointer to a structure of type `node`.

Now consider the function `menu`, which is used to generate the main menu. This function accepts a value for `choice` after the menu has been generated. The only permissible values for `choice` are 1, 2, 3 or 4. An error trap, in the form of a `do - while` statement, causes an error message to be displayed and a new menu to be generated if a value other than 1, 2, 3 or 4 is entered in response to the menu.

The linked list is created by the function `create`. This is a recursive function that accepts a pointer to the current node (i.e., the node that is being created) as an argument. The pointer variable is called `record`.

The `create` function begins by prompting for the current data item; i.e., the string that is to reside in the current node. If the user enters the string `END` (in either upper- or lowercase), then `NULL` is assigned to the pointer that indicates the location of the next node and the recursion stops. If the user enters any string other than `END`, however, memory is allocated for the next node via the `malloc` function and the function calls itself recursively. Thus, the recursion will continue until the user has entered `END` for one of the data items.

Once the linked list has been created, it is displayed via the function `display`. This function is called from `main`, after the call to `create`. Notice that `display` accepts a pointer to the current node as an argument. The function then executes recursively, until it receives a pointer whose value is `NULL`. The recursion therefore causes the entire linked list to be displayed.

Now consider the function `insert`, which is used to add a new component (i.e., a new node) to the linked list. This function asks the user where the insertion is to occur. Note that the function accepts a pointer to the beginning of the list as an argument, and then returns a pointer to the beginning of the list, after the insertion has been made. These two pointers will be the same, unless the insertion is made at the beginning of the list.

The `insert` function does not execute recursively. It first prompts for the new data item (`newitem`), followed by a prompt for the existing data item that will follow the new data item (the existing data item is called `target`). If the insertion is to be made at the *beginning of the list*, then memory is allocated for the new node, `newitem` is assigned to the first member, and the pointer originally indicating the beginning of the linked list (`first`) is assigned to the second member. The pointer returned by `malloc`, which indicates the beginning of the new node, is then assigned to `first`. Hence, the beginning of the new node becomes the beginning of the entire list.

If the insertion is to be made *after an existing node*, then function `locate` is called to determine the location of the insertion. This function returns a pointer to the node *preceding* the target node. The value returned is assigned to the pointer `tag`. Hence, `tag` points to the node that will precede the new node. If `locate` cannot find a match between the value entered for `target` and an existing data item, it will return `NULL`.

If a match is found by `locate`, then the insertion is made in the following manner: memory is allocated for the new node, `newitem` is assigned to the first member of `newrecord` (i.e., to`newrecord->item`), and the pointer to the target node (i.e., `tag->next`) is assigned to the second member of `newrecord` (i.e., `newrecord->next`). The pointer returned by `malloc`, which indicates the beginning of the new node, is then assigned to `tag->next`. Hence, the pointer in the preceding node will point to the new node, and the pointer in the new node will point to the target node.

Now consider the function `locate`. this is a simple recursive function that accepts a pointer to the current node and the target string as arguments, and returns a pointer to the node that *precedes* the current node. Therefore, if the data item in the node following the current node matches the target string, the function will return the pointer to the current node. Otherwise, one of two possible actions will be taken. If the pointer in the node following the current node is `NULL`, indicating the end of the linked list, a match has not been found. Therefore, the function will return `NULL`. But, if the pointer in the node following the current node is something other than `NULL`, the function will call itself recursively, thus testing the next node for a match.

Finally, consider the function `remove`, which is used to delete an existing component (i.e., an existing node) from the linked list. This function is similar to `insert`, though somewhat simpler. It accepts a pointer to the beginning of the linked list as an argument, and returns a pointer to the beginning of the linked list after the deletion has been made.

The `remove` function begins by prompting for the data item to be deleted (`target`). If this is the first data item, then the pointers are adjusted as follows: The pointer indicating the location of the second node is temporarily assigned to the pointer variable `temp`; the memory utilized by the first node is freed, using the library function `free`; and the location of the second node (which is now the first node, because of the deletion) is assigned to `first`. Hence, the beginning of the (former) second node becomes the beginning of the entire list.

If the data item to be deleted is *not* the first data item in the list, then `locate` is called to determine the location of the deletion. This function will return a pointer to the node *preceding* the target node. The value returned is assigned to the pointer variable `tag`. If this value is `NULL`, a match cannot be found. An error message is then generated, requesting that the user try again.

If `locate` returns a value other than `NULL`, the target node is deleted in the following manner: The pointer to the node following the target node is temporarily assigned to the pointer variable `temp`; the memory utilized by the target node is then freed, using the library function `free`; and the value of `temp` is then assigned to `tag->next`. Hence, the pointer in the preceding node will point to the node following the target node.

Let us now utilize this program to create a linked list containing the following cities: Boston, Chicago, Denver, New York, Pittsburgh, San Francisco. We will then add several cities and delete several cities, thus illustrating all of the program's features. We will maintain the list of cities in alphabetical order throughout the exercise. (We could, of course, have the computer do the sorting for us, though this would further complicate an already complex program.)

The entire interactive session is shown below. As usual, the user's responses have been underlined.

```
Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
```

```
Please enter your choice (1, 2, 3 or 4) -> 1

Data item (type 'END' when finished): BOSTON
Data item (type 'END' when finished): CHICAGO
Data item (type 'END' when finished): DENVER
Data item (type 'END' when finished): NEW YORK
Data item (type 'END' when finished): PITTSBURGH
Data item (type 'END' when finished): SAN FRANCISCO
Data item (type 'END' when finished): END

BOSTON
CHICAGO
DENVER
NEW YORK
PITTSBURGH
SAN FRANCISCO

Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 2

New data item: ATLANTA
Place before (type 'END' if last): BOSTON

ATLANTA
BOSTON
CHICAGO
DENVER
NEW YORK
PITTSBURGH
SAN FRANCISCO

Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 2

New data item: SEATTLE
Place before (type 'END' if last): END

ATLANTA
BOSTON
CHICAGO
DENVER
NEW YORK
PITTSBURGH
SAN FRANCISCO
SEATTLE
```

```
Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 3

Data item to be deleted: NEW YORK

ATLANTA
BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE

Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 2

New data item: WASHINGTON
Place before (type 'END' if last): WILLIAMSBURG

Match not found - Please try again

ATLANTA
BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE

Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 2

New data item: WASHINGTON
Place before (type 'END' if last): END

ATLANTA
BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO
```

```
SEATTLE
WASHINGTON

Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 3

Data item to be deleted: ATLANTA

BOSTON
CHICAGO
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE
WASHINGTON

Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 2

New data item: DALLAS
Place before (type 'END' if last): DENVER

BOSTON
CHICAGO
DALLAS
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE
WASHINGTON

Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 3

Data item to be deleted: MIAMI

Match not found - Please try again

BOSTON
CHICAGO
DALLAS
```

```
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE
WASHINGTON

Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 3

Data item to be deleted: WASHINGTON

BOSTON
CHICAGO
DALLAS
DENVER
PITTSBURGH
SAN FRANCISCO
SEATTLE

Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 5

ERROR - Please try again

Main menu:
  1 - CREATE the linked list
  2 - ADD a component
  3 - DELETE a component
  4 - END
Please enter your choice (1, 2, 3 or 4) -> 4

End of computation
```

## 11.7 UNIONS

Unions, like structures, contain members whose individual data types may differ from one another. However, the members within a union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory. They are useful for applications involving multiple members, where values need not be assigned to all of the members at any one time.

Within a union, the bookkeeping required to store members whose data types are different (having different memory requirements) is handled automatically by the compiler. However, the user must keep track of what type of information is stored at any given time. An attempt to access the wrong type of information will produce meaningless results.

In general terms, the composition of a union may be defined as

```
union  tag  {
    member 1;
    member 2;
    . . . . .
    member m;
};
```

where `union` is a required keyword and the other terms have the same meaning as in a structure definition (see Sec. 11.1). Individual union variables can then be declared as

*storage-class* `union` *tag variable 1, variable 2, . . ., variable n*;

where *storage-class* is an optional storage class specifier, `union` is a required keyword, *tag* is the name that appeared in the union definition, and *variable 1, variable 2, . . ., variable n* are union variables of type *tag*.

The two declarations may be combined, just as we did with structures. Thus, we can write

```
storage-class  union  tag  {
            member 1;
            member 2;
            . . . . .
            member m;
}  variable 1,  variable 2,  . . .,  variable n;
```

The *tag* is optional in this type of declaration.

**EXAMPLE 11.33** A C program contains the following union declaration.

```
union  id  {
    char color[12];
    int size;
}  shirt, blouse;
```

Here we have two union variables, `shirt` and `blouse`, of type `id`. Each variable can represent either a 12-character string (`color`) or an integer quantity (`size`) at any one time.

The 12-character string will require more storage area within the computer's memory than the integer quantity. Therefore, a block of memory large enough for the 12-character string will be allocated to each union variable. The compiler will automatically distinguish between the 12-character array and the integer quantity within the given block of memory, as required.

A union may be a member of a structure, and a structure may be a member of a union. Moreover, structures and unions may be freely mixed with arrays.

**EXAMPLE 11.34** A C program contains the following declarations.

```
union  id  {
    char color[12];
    int size;
};
struct  clothes  {
    char manufacturer[20];
    float cost;
    union id description;
}  shirt, blouse;
```

Now `shirt` and `blouse` are structure variables of type `clothes`. Each variable will contain the following members: a string (`manufacturer`), a floating-point quantity (`cost`), and a union (`description`). The union may represent either a string (`color`) or an integer quantity (`size`).

Another way to declare the structure variables `shirt` and `blouse` is to combine the preceding two declarations, as follows.

```
struct clothes {
    char manufacturer[20];
    float cost;
    union {
        char color[12];
        int size;
    } description;
} shirt, blouse;
```

This declaration is more concise, though perhaps less straightforward, than the original declarations.

An individual union member can be accessed in the same manner as an individual structure member, using the operators . and -> . Thus, if *variable* is a union variable, then *variable.member* refers to a member of the union. Similarly, if *ptvar* is a pointer variable that points to a union, then *ptvar->member* refers to a member of that union.

**EXAMPLE 11.35** Consider the simple C program shown below.

```
#include <stdio.h>

main()

{
    union id {
        char color;
        int size;
    };

    struct {
        char manufacturer[20];
        float cost;
        union id description;
    } shirt, blouse;

    printf("%d\n", sizeof(union id));

    /* assign a value to color */
    shirt.description.color = 'w';
    printf("%c %d\n", shirt.description.color, shirt.description.size);

    /* assign a value to size */
    shirt.description.size = 12;
    printf("%c %d\n", shirt.description.color, shirt.description.size);
}
```

This program contains declarations similar to those shown in Example 11.34. Notice, however, that the first member of the union is now a single character rather than the 12-character array shown in the previous example. This change is made to simplify the assignment of appropriate values to the union members.

Following the declarations and the initial `printf` statement, we see that the character `'w'` is assigned to the union member `shirt.description.color`. Note that the other union member, `shirt.description.size`, will not have a meaningful value. The values of both union members are then displayed.

We then assign the value 12 to `shirt.description.size`, thus overwriting the single character previously assigned to `shirt.description.color`. The values of both union members are then displayed once more.

Execution of the program results in the following output.

```
2
w -24713
@ 12
```

The first line indicates that the union is allocated two bytes of memory, to accommodate an integer quantity. In line 2, the first data item (`w`) is meaningful, but the second (`-24713`) is not. In line 3, the first data item (`@`) is meaningless, but the second data item (`12`) has meaning. Thus, each line of output contains one meaningful value, in accordance with the assignment statement preceding each `printf` statement.

A union variable can be initialized provided its storage class is either `external` or `static`. Remember, however, that *only one member of a union can be assigned a value at any one time.* Most compilers will accept an initial value for only one union member, and they will assign this value to the first member within the union.

**EXAMPLE 11.36** Shown below is a simple C program that includes the assignment of initial values to a structure variable.

```
#include <stdio.h>

main()

{
    union id  {
        char color[12];
        int size;
    };

    struct clothes {
        char manufacturer[20];
        float cost;
        union id description;
    };

    static struct clothes shirt = {"American", 25.00, "white"};

    printf("%d\n", sizeof(union id));
    printf("%s %5.2f ", shirt.manufacturer, shirt.cost);
    printf("%s %d\n", shirt.description.color, shirt.description.size);

    shirt.description.size = 12;
    printf("%s %5.2f ", shirt.manufacturer, shirt.cost);
    printf("%s %d\n", shirt.description.color, shirt.description.size);
}
```

Notice that `shirt` is a static structure variable of type `clothes`. One of its members is `description`, which is a union of type `id`. This union consists of two members: a 12-character array and an integer quantity.

The structure variable declaration includes the assignment of the following initial values: `"American"` is assigned to the array member `shirt.manufacturer`; `25.00` is assigned to the integer member `shirt.cost`, and `"white"` is assigned to the union member `shirt.description.color`. Notice that the second union member within the structure, i.e., `shirt.description.size`, remains unspecified.

The program first displays the size of the memory block allocated to the union, and the value of each member of `shirt`. Then `12` is assigned to `shirt.description.size`, and the value of each member of `shirt` is again displayed.

When the program is executed, the following output is generated.

```
12
American 25.00 white 26743
American 25.00 - 12
```

The first line indicates that 12 bytes of memory are allocated to the union, in order to accommodate the 12-character array. The second line shows the values initially assigned to `shirt.manufacturer`, `shirt.cost` and `shirt.description.color`. The value shown for `shirt.description.size` is meaningless. In the third line we see that `shirt.manufacturer` and `shirt.cost` are unchanged. Now, however, the reassignment of the union members causes `shirt.description.color` to have a meaningless value, but `shirt.description.size` shows the newly assigned value of 12.

In all other respects, unions are processed in the same manner, and with the same restrictions, as structures. Thus, individual union members can be processed as though they were ordinary variables of the same data type, and pointers to unions can be passed to or from functions (by reference). Moreover, most C compilers permit an entire union to be assigned to another, provided both unions have the same composition. These compilers also permit entire unions to be passed to or from functions (by value), in accordance with the ANSI standard.

**EXAMPLE 11.37 Raising a Number to a Power** This example is a bit contrived, though it does illustrate how a union can be used to pass information to a function. The problem is to raise a number to a power. Thus, we wish to evaluate the formula $y = x^n$, where $x$ and $y$ are floating-point values, and $n$ can be either integer or floating point.

If $n$ is an integer, then $y$ can be evaluated by multiplying $x$ by itself an appropriate number of times. For example, the quantity $x^3$ could be expressed in terms of the product $(x)(x)(x)$. On the other hand, if $n$ is a floating-point value, we can write $\log y = n \log x$, or $y = e^{(n \log x)}$. In the latter case $x$ must be a positive quantity, since we cannot take the log of zero or a negative quantity.

Now let us introduce the following declarations:

```
typedef  union  {
    float fexp;        /* floating-point exponent */
    int nexp;          /* integer exponent */
} nvals;

typedef  struct  {
float x;               /* value to be raised to a power */
    char flag;         /* 'f' if exponent is floating-point,
                          'i' if exponent is integer */
    nvals exp;         /* union containing exponent */
} values;

values a;
```

Thus, `nvals` is a user-defined union type, consisting of the floating-point member `fexp` and the integer member `nexp`. These two members represent the two possible types of exponents in the expression $y = x^n$. Similarly, `values` is a user-defined structure type, consisting of a floating-point member `x`, a character member `flag` and a union of type `nvals` called `exp`. Note that `flag` indicates the type of exponent currently represented by the union. If `flag` represents `'i'`, the union will represent an integer exponent (`nexp` will currently be assigned a value); and if `flag` represents `'f'`, the union will represent a floating-point exponent (`fexp` will currently be assigned a value). Finally, we see that `a` is a structure variable of type `values`.

With these declarations, it is easy to write a function that will evaluate the formula $y = x^n$, as follows.

```
float power(values a)     /* carry out the exponentiation */

{
    int i;
    float y = a.x;

    if (a.flag == 'i')  {      /* integer exponent */
       if (a.exp.nexp == 0)
          y = 1.0;             /* zero exponent */
       else  {
          for (i = 1; i < abs(a.exp.nexp); ++i)
             y *= a.x;
          if (a.exp.nexp < 0)
             y = 1./y;         /* negative integer exponent */
       }
    }
    else                       /* floating-point exponent */
       y = exp(a.exp.fexp * log(a.x));

    return(y);
}
```

This function accepts a structure variable (a) of type `values` as an argument. The method used to carry out the calculations depends on the value assigned to `a.flag`. If `a.flag` is assigned the character `'i'`, then the exponentiation is carried out by multiplying `a.x` by itself an appropriate number of times. Otherwise, the exponentiation is carried out using the formula $y = e^{(n \log x)}$. Notice that the function contains corrections to accommodate a zero exponent ($y = 1.0$), and for a negative integer exponent.

Let us add a `main` function which prompts for the values of $x$ and $n$, determines whether or not $n$ is an integer (by comparing $n$ with its truncated value), assigns appropriate values to `a.flag` and `a.exp`, calls `power`, and then writes out the result. We also include a provision for generating an error message if $n$ is a floating-point exponent and the value of $x$ is less than or equal to zero.

Here is the entire program.

```
/* program to raise a number to a power */

#include <stdio.h>
#include <math.h>

typedef  union  {
    float fexp;           /* floating-point exponent */
    int nexp;             /* integer exponent */
} nvals;

typedef  struct  {
    float x;              /* value to be raised to a power */
    char flag;            /* 'f' if exponent is floating-point,
                            'i' if exponent is integer */
    nvals exp;            /* union containing exponent */
} values;

float power(values a);    /* function prototype */
```

```
main()

{
    values a;                   /* structure containing x, flag and fexp/nexp */
    int i;
    float n, y;

    /* enter input data */
    printf("y = x^n\n\n\nEnter a value for x: ");
    scanf("%f", &a.x);
    printf("Enter a value for n: ");
    scanf("%f", &n);

    /* determine type of exponent */
    i = (int) n;
    a.flag = (i == n) ? 'i' : 'f';
    if (a.flag == 'i')
       a.exp.nexp = i;
    else
       a.exp.fexp = n;

    /* raise x to the appropriate power and display the result */
    if (a.flag == 'f' && a.x <= 0.0)   {
       printf("\nERROR - Cannot raise a non-positive number to a ");
       printf("floating-point power");
    }
    else  {
       y = power(a);
       printf("\ny = %.4f", y);
    }
}


float power(values a)      /* carry out the exponentiation */

{
    int i;
    float y = a.x;

    if (a.flag == 'i')  {       /* integer exponent */
       if (a.exp.nexp == 0)
          y = 1.0;              /* zero exponent */
       else  {
          for (i = 1; i < abs(a.exp.nexp); ++i)
             y *= a.x;
          if (a.exp.nexp < 0)
             y = 1./y;          /* negative integer exponent */
       }
    }
    else                        /* floating-point exponent */
       y = exp(a.exp.fexp * log(a.x));

    return(y);
}
```

Notice that the union and structure declarations are external to the program functions, but the structure variable a is defined locally within each function.

The program does not execute repetitiously. Several typical dialogs, each representing a separate program execution, are shown below. As usual, the user's responses are underlined.

```
Enter a value for x: 2
Enter a value for n: 3

y = 8.0000

Enter a value for x: -2
Enter a value for n: 3

y = -8.0000

Enter a value for x: 2.2
Enter a value for n: 3.3

y = 13.4895

Enter a value for x: -2.2
Enter a value for n: 3.3

ERROR - cannot raise a non-positive number to a floating-point power
```

It should be pointed out that most C compilers include the library function pow, which is used to raise a number to a power. We have used pow in several earlier programming examples (see Examples 5.2, 5.4, 6.21, 8.13 and 10.30). The present program is not meant to replace pow; it is presented only to illustrate the use of a union in a representative programming situation.

## *Review Questions*

**11.1** What is a structure? How does a structure differ from an array?

**11.2** What is a structure member? What is the relationship between a structure member and a structure?

**11.3** Describe the syntax for defining the composition of a structure. Can individual members be initialized within a structure type declaration?

**11.4** How can structure variables be declared? How do structure variable declarations differ from structure type declarations?

**11.5** What is a tag? Must a tag be included in a structure type definition? Must a tag be included in a structure variable declaration? Explain fully.

**11.6** Can a structure variable be defined as a member of another structure? Can an array be included as a member of a structure? Can an array have structures as elements?

**11.7** How are the members of a structure variable assigned initial values? What restrictions apply to the structure's storage class when initial values are assigned?

**11.8** How is an array of structures initialized?

**11.9** What is the scope of a member name? What does this imply with respect to the naming of members within different structures?

**11.10** How is a structure member accessed? How can a structure member be processed?

**11.11** What is the precedence of the period (.) operator? What is its associativity?

**11.12** Can the period operator be used with an array of structures? Explain.

**11.13** What is the only operation that can be applied to an entire structure in some older versions of C? How is this rule modified in newer versions that conform to the current ANSI standard?

**11.14** How can the size of a structure be determined? In what units is the size reported?

**11.15** What is the purpose of the `typedef` feature? How is this feature used in conjuction with structures?

**11.16** How is a structure-type pointer variable declared? To what does this type of variable point?

**11.17** How can an individual structure member be accessed in terms of its corresponding pointer variable?

**11.18** What is the precedence of the `->` operator? What is its associativity? Compare with the answers to question 11.11.

**11.19** Suppose a pointer variable points to a structure that contains another structure as a member. How can a member of the embedded structure be accessed?

**11.20** Suppose a pointer variable points to a structure that contains an array as a member. How can an element of the embedded array be accessed?

**11.21** Suppose a member of a structure is a pointer variable. How can the object of the pointer be accessed, in terms of the structure variable name and the member name?

**11.22** What happens when a pointer to a structure is incremented? What danger is associated with this type of operation?

**11.23** How can an entire structure be passed to a function? Describe fully, both for older and newer versions of C.

**11.24** How can an entire structure be returned from a function? Describe fully, both for older and newer versions of C.

**11.25** What is a self-referential structure? For what kinds of applications are self-referential structures useful?

**11.26** What is the basic idea behind a linked data structure? What advantages are there in the use of linked data structures?

**11.27** Summarize several types of commonly used linked data structures.

**11.28** What is a union? How does a union differ from a structure?

**11.29** For what kinds of applications are unions useful?

**11.30** In what sense can unions, structures and arrays be intermixed?

**11.31** How is a union member accessed? How can a union member be processed? Compare with your answers to question 11.10.

**11.32** How is a member of a union variable assigned an initial value? In what way does the initialization of a union variable differ from the initialization of a structure variable?

**11.33** Summarize the rules that apply to processing unions. Compare with the rules that apply to processing structures.

## Problems

**11.34** Define a structure consisting of two floating-point members, called `real` and `imaginary`. Include the tag `complex` within the definition.

**11.35** Declare the variables `x1`, `x2` and `x3` to be structures of type `complex`, as described in the preceding problem.

**11.36** Combine the structure definition and the variable declarations described in Probs. 11.34 and 11.35 into one declaration.

**11.37** Declare a variable `x` to be a structure variable of type `complex`, as described in Prob. 11.34. Assign the initial values 1.3 and −2.2 to the members `x.real` and `x.imaginary`, respectively.

**11.38** Declare a pointer variable, `px`, which points to a structure of type `complex`, as described in Prob. 11.34. Write expressions for the structure members in terms of the pointer variable.

**11.39** Declare a one-dimensional, 100-element array called `cx` whose elements are structures of type `complex`, as described in Prob. 11.34.

**11.40** Combine the structure definition and the array declaration described in Probs. 11.34 and 11.39 into one declaration.

**11.41** Suppose that `cx` is a one-dimensional, 100-element array of structures, as described in Prob. 11.39. Write expressions for the members of the 18*th* array element (i.e., element number 17).

**11.42** Define a structure that contains the following three members:

(*a*) an integer quantity called `won`

(*b*) an integer quantity called `lost`

(*c*) a floating-point quantity called `percentage`

Include the user-defined data type `record` within the definition.

**11.43** Define a structure that contains the following two members:

(*a*) a 40-element character array called `name`

(*b*) a structure named `stats`, of type `record`, as defined in Prob. 11.42

Include the user-defined data type `team` within the definition.

**11.44** Declare a variable `t` to be a structure variable of type `team`, as described in Prob. 11.43. Write an expression for each member and submember of `t`.

**11.45** Declare a variable `t` to be a structure variable of type `team`, as in the previous problem. Now, however, initialize `t` as follows.

```
name : Chicago Bears
won : 14
lost : 2
percentage : 87.5
```

**11.46** Write a statement that will display the size of the memory block associated with the variable `t` which was described in Prob. 11.44.

**11.47** Declare a pointer variable `pt`, which points to a structure of type `team`, as described in Prob. 11.43. Write an expression for each member and submember within the structure.

**11.48** Declare a one-dimensional, 48-element array called `league` whose elements are structures of type `team`, as described in Prob. 11.43. Write expressions for the name and percentage of the 5th team in the league (i.e., team number 4).

**11.49** Define a self-referential structure containing the following three members:

(*a*) a 40-element character array called `name`

(*b*) a structure named `stats`, of type `record`, as defined in Prob. 11.42

(*c*) a pointer to another structure of this same type, called `next`

Include the tag `team` within the structure definition. Compare your solution with that of Prob. 11.43.

**11.50** Declare `pt` to be a pointer to a structure whose composition is described in the previous problem. Then write a statement that will allocate an appropriate block of memory, with `pt` pointing to the beginning of the memory block.

**11.51** Define a structure of type `hms` containing three integer members, called `hour`, `minute` and `second`, respectively. Then define a union containing two members, each a structure of type `hms`. Call the union members `local` and `home`, respectively. Declare a pointer variable called `time` that points to this union.

**11.52** Define a union of type `ans` which contains the following three members:

(*a*) an integer quantity called `ians`

(*b*) a floating-point quantity called `fans`

(*c*) a double-precision quantity called `dans`

Then define a structure which contains the following four members:

(*a*) a union of type `ans`, called `answer`

(*b*) a single character called `flag`

(*c*) integer quantities called `a` and `b`

Finally, declare two structure variables, called `x` and `y`, whose composition is as described above.

**11.53** Declare a structure variable called `v` whose composition is as described in Prob. 11.52. Assign the following initial values within the declaration:

```
answer : 14
flag : 'i'
a : -2
b : 5
```

**11.54** Modify the structure definition described in Prob. 11.52 so that it contains an additional member, called `next`, which is a pointer to another structure of the same type. (Note that the structure will now be self-referential.) Add a declaration of two variables, called `x` and `px`, where `x` is a structure variable and `px` is a pointer to a structure variable. Assign the starting address of `x` to `px` within the declaration.

**11.55** Describe the output generated by each of the following programs. Explain any differences between the programs.

(*a*)

```
#include <stdio.h>

typedef struct {
     char *a;
     char *b;
     char *c;
} colors;

void funct(colors sample);

main()
{
    static colors sample = {"red", "green", "blue"};

    printf("%s %s %s\n", sample.a, sample.b, sample.c);
    funct(sample);
    printf("%s %s %s\n", sample.a, sample.b, sample.c);
}

void funct(colors sample)
{
    sample.a = "cyan";
    sample.b = "magenta";
    sample.c = "yellow";
    printf("%s %s %s\n", sample.a, sample.b, sample.c);
    return;
}
```

(*b*)

```
#include <stdio.h>

typedef struct {
     char *a;
     char *b;
     char *c;
} colors;

void funct(colors *pt);

main()
{
    static colors sample = {"red", "green", "blue"};

    printf("%s %s %s\n", sample.a, sample.b, sample.c);
    funct(&sample);
    printf("%s %s %s\n", sample.a, sample.b, sample.c);
}
```

```
void funct(colors *pt)
{
    pt->a = "cyan";
    pt->b = "magenta";
    pt->c = "yellow";
    printf("%s %s %s\n", pt->a, pt->b, pt->c);
    return;
}
```

(*c*)

```
#include <stdio.h>

typedef struct {
     char *a;
     char *b;
     char *c;
} colors;

colors funct(colors sample);

main()
{
    static colors sample = {"red", "green", "blue"};

    printf("%s %s %s\n", sample.a, sample.b, sample.c);
    sample = funct(sample);
    printf("%s %s %s\n", sample.a, sample.b, sample.c);
}

colors funct(colors sample)
{
    sample.a = "cyan";
    sample.b = "magenta";
    sample.c = "yellow";
    printf("%s %s %s\n", sample.a, sample.b, sample.c);
    return(sample);
}
```

**11.56** Describe the output generated by the following program. Distinguish between meaninful and meaningless output.

```
#include <stdio.h>

main()
{
    union {
        int i;
        float f;
        double d;
    } u;

    printf("%d\n", sizeof u);
    u.i = 100;
    printf("%d %f %f\n", u.i, u.f, u.d);
    u.f = 0.5;
    printf("%d %f %f\n", u.i, u.f, u.d);
    u.d = 0.0166667;
    printf("%d %f %f\n", u.i, u.f, u.d);
}
```

**11.57** Describe the output generated by the following programs. Explain any differences between them.

(*a*)
```
#include <stdio.h>

typedef union {
     int i;
     float f;
} udef;

void funct(udef u);


main()
{
    udef u;

    u.i = 100;
    u.f = 0.5;
    funct(u);
    printf("%d %f\n", u.i, u.f);
}


void funct(udef u)
{
    u.i = 200;
    printf("%d %f\n", u.i, u.f);
    return;
}
```

(*b*)
```
#include <stdio.h>

typedef union {
    int i;
    float f;
} udef;

void funct(udef u);


main()
{
    udef u;

    u.i = 100;
    u.f = 0.5;
    funct(u);
    printf("%d %f\n", u.i, u.f);
}


void funct(udef u)
{
    u.f = -0.3;
    printf("%d %f\n", u.i, u.f);
    return;
}
```

(*c*)
```
#include <stdio.h>

typedef union {
    int i;
    float f;
} udef;

udef funct(udef u);

main()
{
    udef u;

    u.i = 100;
    u.f = 0.5;
    u = funct(u);
    printf("%d %f\n", u.i, u.f);
}

udef funct(udef u)
{
    u.f = -0.3;
    printf("%d %f\n", u.i, u.f);
    return(u);
}
```

## Programming Problems

**11.58** Answer the following questions as they pertain to your particular C compiler or interpreter.

(*a*) Can an entire structure variable (or union variable) be assigned to another structure (union) variable, provided both variables have the same composition?

(*b*) Can an entire structure variable (or union variable) be passed to a function as an argument?

(*c*) Can an entire structure variable (or union variable) be returned from a function to its calling routine?

(*d*) Can a pointer to a structure (or a union) be passed to a function as an argument?

(*e*) Can a pointer to a structure (or a union) be returned from a function to its calling routine?

**11.59** Modify the program given in Example 11.26 (locating customer records) so that the function `search` returns a complete structure rather than a pointer to a structure. (Do not attempt this problem if your version of C does not support the return of entire structures from a function.)

**11.60** Modify the billing program shown in Example 11.28 so that any of the following reports can be displayed:

(*a*) Status of all customers (now generated by the program)

(*b*) Status of overdue and delinquent customers only

(*c*) Status of delinquent customers only

Include a provision for generating a menu when the program is executed, from which the user may choose which report will be generated. Have the program return to the menu after printing each report, thus allowing for the possibility of generating several different reports.

**11.61** Modify the billing program shown in Example 11.28 so that the structure of type `record` now includes a union containing the members `office_address` and `home_address`. Each union member should itself be a structure consisting of two 80-character arrays, called `street` and `city`, respectively. Add another member to the primary structure (of type `record`), which is a single character called `flag`. This member should be assigned a character (e.g., `'o'` or `'h'`) to indicate which type of address is currently stored in the union.

Modify the remainder of the program so that the user is asked which type of address will be supplied for each customer. Then display the appropriate address, with a corresponding label, along with the rest of the output.

**11.62** Modify the program given in Example 11.37 so that a number raised to a floating-point power can be executed in either single precision or double precision, as specified by the user in response to a prompt. The union type `nvals` should now contain a third member, which should be a double-precision quantity called `dexp`.

**11.63** Rewrite each of the following C programs so that it makes use of structure variables.

(*a*) The depreciation program presented in Example 7.20.
(*b*) The program given in Example 10.28 for displaying the day of the year
(*c*) The program for determining the future value of monthly deposits, given in Example 10.31

**11.64** Modify the piglatin generator presented in Example 9.14 so that it will accept multiple lines of text. Represent each line of text with a separate structure. Include the following three members within each structure:

(*a*) The original line of text
(*b*) The number of words within the line
(*c*) The modified line of text (i.e., the piglatin equivalent of the original text)

Include the enhancements described in Prob. 9.36 (i.e., provisions for punctuation marks, uppercase letters and double-letter sounds).

**11.65** Write a C program that reads several different names and addresses into the computer, rearranges the names into alphabetical order, and then writes out the alphabetized list. (See Examples 9.20 and 10.26.) Make use of structure variables within the program.

**11.66** For each of the following programming problems described in earlier chapters, write a complete C program that makes use of structure variables.

(*a*) The student exam score averaging problem described in Prob. 9.40.
(*b*) The more comprehensive version of the student exam score averaging problem described in Prob. 9.42.
(*c*) The problem that matches the names of countries with their corresponding capitals, described in Prob. 9.46.
(*d*) The text encoding-decoding problem as described in Prob. 9.49, but extended to accommodate multiple lines of text.

**11.67** Write a C program that will accept the following information for each team in a baseball or a football league.

1. Team name, including the city (e.g., Pittsburgh Steelers)
2. Number of wins
3. Number of losses

For a baseball team, add the following information:

4. Number of hits
5. Number of runs
6. Number of errors
7. Number of extra-inning games

Similarly, add the following information for a football team:

4. Number of ties
5. Number of touchdowns
6. Number of field goals
7. Number of turnovers
8. Total yards gained (season total)
9. Total yards given up to opponents

Enter this information for all of the teams in the league. Then reorder and print the list of teams according to their win-lose records, using the reordering techniques described in Examples 9.13 and 10.16 (see also Examples 9.21 and 10.26). Store the information in an array of structures, where each array element (i.e., each structure) contains the information for a single team. Make use of a union to represent the variable information (either baseball or football) that is included as a part of the structure. This union should itself contain two structures, one for baseball-related statistics and the other for football-related statistics.

Test the program using a current set of league statistics. (Ideally, the program should be tested using both baseball and football statistics.)

**11.68** Modify the program given in Example 11.32 so that it makes use of each of the following linked structures.

(*a*) A linear linked list with two sets of pointers: one set pointing in the forward direction, the other pointing backwards.

(*b*) A circular linked list. Be sure to include a pointer to identify the beginning of the circular list.

**11.69** Modify the program given in Example 11.32 so that each node contains the following information:

(*a*) Name

(*b*) Street address

(*c*) City/State/ZIP code

(*d*) Account number

(*e*) Account status (a single character indicating current, overdue or delinquent status)

**11.70** Write a complete C program that will allow you to enter and maintain a computerized version of your family tree. Begin by specifying the number of generations (i.e., the number of levels within the tree). Then enter the names and nationalities in a hierarchical fashion, beginning with your own name and nationality. Include capabilities for modifying the tree and for adding new names (new nodes) to the tree. Also, include a provision for displaying the entire tree automatically after each update.

Test the program, including at least three generations if possible (you, your parents and your grandparents). Obviously, the tree becomes more interesting as the number of generations increases.

**11.71** An RPN calculator utilizes a scheme whereby each new numerical value is followed by the operation that is to be performed between the new value and its predecessor. (RPN stands for "reverse Polish notation.") Thus, adding two numbers, say 3.3 and 4.8, would require the following keystrokes:

```
3.3  <enter>
4.8  +
```

The sum, `8.1`, would then be displayed in the calculator's single visible register.

RPN calculators make use of a *stack*, typically containing four registers (four components), as illustrated in Fig. 11.7. Each new number is entered into the *X* register, causing all previously entered values to be pushed up in the stack. If the top register (i.e., the *T* register) was previously occupied, then the old number will be lost (it will be overwritten by the value that is pushed up from the *Z* register).

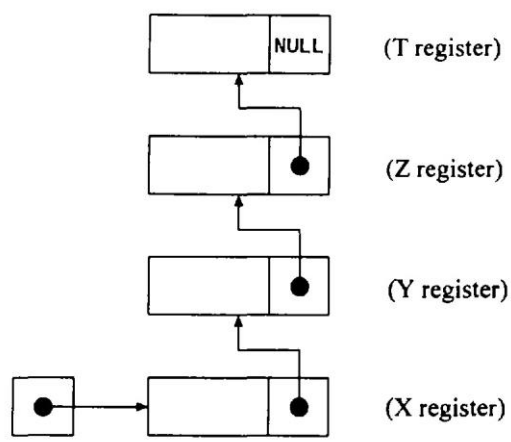



**Fig. 11.7**

Arithmetic operations are always carried out between the numbers in the *X* and *Y* registers. The result of such an operation will always be displayed in the *X* register, causing everything in the upper registers to drop down one level (thus "popping" the stack). This procedure is illustrated in Fig. 11.8(*a*) to (*c*) for the addition of the values 3.3 and 4.8, as described above.

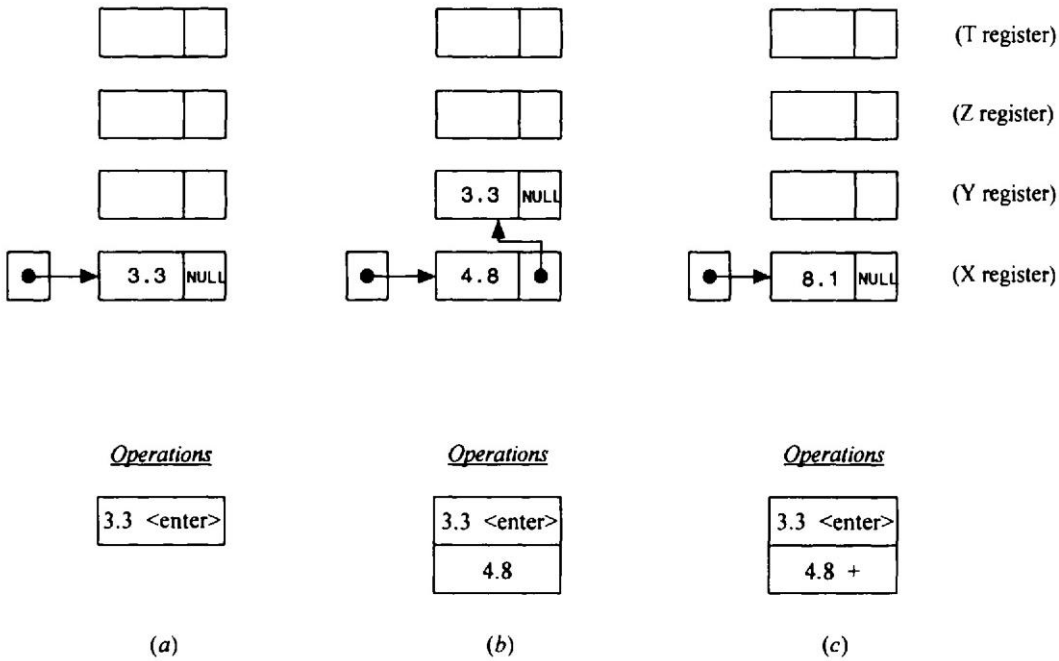



**Fig. 11.8**

Write an interactive C program that will simulate an RPN calculator. Display the contents of the stack after each operation, as in Fig. 11.8(*a*) to (*c*). Include a provision for carrying out each of the following operations.

| *Operation* | *Keystrokes* |
|---|---|
| enter new data | (value) <enter> |
| addition | (value) + |
| subtraction | (value) - |
| multiplication | (value) * |
| division | (value) / |

Test the program using any numerical data of your choice.