Chapter 3

Structures

3.1 Motivation

Many problems and applications in computing require representation and manipulation of data that are made up of a group of elements. These elements may be of the same data type (i.e., homogeneous) or of different data types (i.e., heterogeneous).

Consider for example how we can represent date. Although we usually abstract date as a single entity, it is actually a group of simple elements which include month, day and year. These elements can be declared as variables of type `int` as shown in the following code snippet:

```
int month;
int day;
int year;
```

The name (of a person) is another object that we abstract as a single entity. We can represent it as a group of elements as shown in the following:

```
char firstname[20];
char middlename[20];
char lastname[20];
```

For trivial programming problems that deal with just one name or just one date, the above representation would be sufficient. However, what if there is a

need to maintain several names and dates? Institutions such as banks, hospitals, hotels collect and store information about their clients (names and birthdates) in a database. Such databases will normally store hundreds or thousands of client information. The simple name and date representations in the code snippets above will not be appropriate.

We will learn in this chapter how to represent, store and manipulate entities that are composed of an aggregate of data values. In C programming language, these are referred to as *structures*.

3.2 What is a Structure?

Simply stated, a *structure* is a group of elements. Unlike arrays, however, the elements of a structure can be heterogeneous, i.e., of different data types. In the C programming language, an element of a structure is referred to as *member*. A member of a structure can be of a simple data type (`char`, `int`, `float`, `double`), a pointer data type, an array or even another structure data type.

The word structure may sound technical but we can easily understand the concept if we associate it with the word *record* – a name used in everyday language. Here are some examples of structures:

1. A mobile phone book directory is made up of several phone book records. Each phone book record (structure) has two members, namely number and name. Number is a numeric value while name is made up of several characters (character array).

2. Consider next an email or a social networking account. We can think of an account as a structure with the following as members: login name and password.

3. Date, for example 12/25/2009 (Christmas of 2009), as noted in the previous section is actually a structure made up of three members namely: month, day, and year. The month member can be represented as a string (for example, "December") or as an integer (for example, 12).

A structure is governed by a parent–child relationship which can be visualized as shown in Figure 3.1. The parent is the structure, and the children are the structure's members. The corresponding diagrams for the sample structures mentioned above are shown in Figure 3.2.

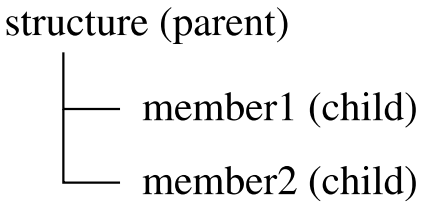

Figure 3.1: Graphical representation of a structure and its members

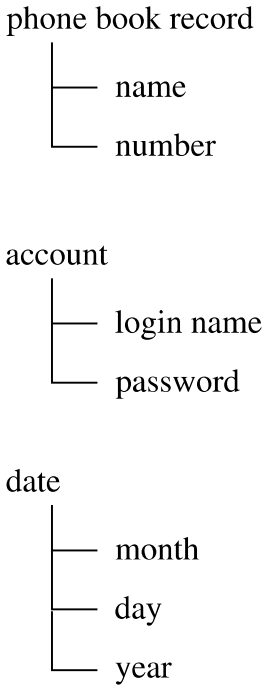

Figure 3.2: Graphical representations of structures in the previous example

3.3 `struct` Type and Structure Variable

A structure variable is a variable whose data type is a *structure type*. C uses `struct` as keyword to denote a structure type.[1]

The syntax for declaring a `struct` type and structure variable is

```
struct [tag-name] {
   <data type> <member-name>;
   <data type> <member-name>;
               :
               :
   <data type> <member-name>;
} [structure-var-name];
```

There are four different variations by which the syntax can be applied.

Variation #1 - Unnamed struct without an instance. The simplest variation contains only the `struct` keyword and a list of member declarations inside a pair of curly brackets. An example declaration of a `struct` type made up of four members representing the four basic data types is shown below:

```
struct {
   char ch;
   int i;
   float f;
   double d;
};
```

Try to compile a program that has a similar declaration as in above. Notice that `gcc` reports a ‘‘unnamed struct/union that defines no instances’’ warning message. The “unnamed struct” part of the warning means that the structure type does not have a tag–name. On the other hand, the “defines no instances” portion means that a structure variable of such structure type was not declared.

Variation 1 is by itself not useful. In a latter section, we will see how to use it together with `typedef`.

[1] Note that `struct` is a *user–defined data type*, i.e., it is the programmer who specifies the actual details about the type.

Variation #2 - Named struct without an instance. When a tag–name is added to variation 1, we will have the 2nd variation. For example:

```
struct sampleTag {
   char ch;
   int i;
   float f;
   double d;
};
```

This declaration specifies that the name of the `struct` type is `sampleTag`. Try to compile a program that has a declaration similar to the example above. Notice that the warning "unnamed struct ..." will no longer appear because the `struct` has been named.

Variation 2 is prevalent in actual programming practice.

Variation #3 - Unnamed struct with an instance. Adding a struct–var–name to variation 1 results into the 3rd variation. For example:

```
struct {
   char ch;
   int i;
   float f;
   double d;
} x;
```

This variation declares (i) an unnamed `struct` type, and (ii) a structure variable, i.e., an instance of the `struct` type named as `x`.

What happens if we want to declare multiple instances? This can be done simply by specifying the name of additional instances separated by a comma. For example, three instances named `x`, `y` and `z` are declared in the following code.

```
struct {
   char ch;
   int i;
   float f;
   double d;
} x, y, z;
```

Variation #4 - Named struct with an instance. The fourth variation has a tag–name as well as a struct–var–name. For example:

```
struct sampleTag {
   char ch;
   int i;
   float f;
   double d;
} x;
```

This variations declares both (i) `struct` type with a tag–name of `sampleTag` and (ii) an instance of the `struct` type, i.e., a structure variable, named as `x`. Similar to variation 3, several instances maybe declared by giving the names separated by comma.

Once a named `struct` type is made, as in the case of Variations 2 and 4, instances of such a type can be declared in subsequent codes. The tag–name serves as a substitute for the list of member declarations. The following shows how this is done.

```
/* struct type declaration from Variation 2 */
struct sampleTag {
    char ch;
    int i;
    float f;
    double d;
};

/* declare one instance */
struct sampleTag x0;

/* declare two more instances on separate lines*/
struct sampleTag x1;
struct sampleTag x2;

/* declare three more instances on the same line */
struct sampleTag x3, x4, x5;
```

In C, the declaration of `struct sampleTag` is also referred to as *structure template*. It is a common practice among C programmers to do a two–step declaration similar to the preceding example above. The steps are:

Step 1. Following variation 2 - declare the structure template.

Step 2. Declare the instance, i.e., structure variable, with the structure template as its data type.

Let us apply what we learned by declaring the structures mentioned in Section 3.2. The corresponding C declarations are shown in Listing 3.1. We included the `<string.h>` header file for string related operations which will be needed to manipulate character array structure members such as `name`, and `password`.

Notice that we declared the structure templates outside of the `main()` function. The reason for this will be explained when we go to the discussion of functions and structures in the latter sections.

Listing 3.1: Example `struct` type and instance declarations

```
1  #include <stdio.h>
2  #include <string.h>  /* for string related functions */
3
4  /* first: declare the structure templates */
5  struct phoneTag {
6     int number;
7     char name[30];
8  };
9
10 struct accountTag {
11    char login_name[7];
12    char password[20];
13 };
14
15 struct dateTag {
16    int month;
17    int day;
18    int year;
19 };
20
21 int main()
22 {
23       /* next: declare the instances */
24       struct phoneTag landline;
25
```

```
26      struct accountTag email_account;
27      struct accountTag facebook_account;
28
29      struct dateTag today;
30      struct dateTag birthdate;
31
32      /*--- codes to manipulate structures follow --- */
33      return 0;
34  }
```

► Self–Directed Learning Activity ◄

1. What do you think is the minimum possible number of members in a C `struct`? Verify your answer by writing and compiling a sample C declaration.
2. Encode the examples mentioned in pages 60 to 62, i.e., the declarations in Variations 1 to 4. Note that you have to declare them inside a function such as `main()`. Compile and test the codes. Take special note of the compiler warning for Variation 1 declaration.
3. Encode and compile the program in Listing 3.1.
4. Think of two or more structures. Thereafter, add your own declarations using the program you encoded in the previous item. Compile your program to see if your declarations are correct.

3.4 Operations on Structures

There are only three possible operations that can be performed on a structure variable, namely:

1. access its members
2. assign a structure to another structure variable of the same type
3. get the memory address of the structure variable via the "address–of" `&` operator

We will explain the details of these operations in the following sections.

3.5 Accessing a Member of a Structure

Any member of a structure variable can be accessed by using the *structure member operator* which is denoted by a dot symbol. The syntax is:

```
<struct-var-name>.<member-name>
```

The following codes show how to initialize, and then print the values of the members of structure variable `birthdate` (refer back to Listing 3.1):

```
/* initialize birthdate members */
birthdate.month = 12;
birthdate.day = 25;
birthdate.year = 2009;

/* print birthdate members */
printf("Birthdate is %d/%d/%d\n",
        birthdate.month, birthday.day, birthdate.year);
```

Insert these codes after the comment ("codes to manipulate structures follow") in Listing 3.1 and verify that it indeed works.

We can input the values of individual members as well using `scanf()` as shown in the following example:

```
/* input today members */
printf("Input today's month day and year: ");
scanf("%d %d %d", &today.month, &today.day, &today.year);

/* print today members */
printf("today is %d/%d/%d\n",
        today.month, today.day, today.year);
```

The pre–defined function `strcpy()` can be used to initialize members which are of type character array. Input can be done using `scanf()` function with a `"%s"` as format symbol corresponding to character arrays. For example:

```
/* initialize using strcpy() */
strcpy(email_account.login_name,  "pusa");
strcpy(email_account.password,  "MuNinG123");

/* input using scanf("%s", ...) */
printf("Input login name: ");
scanf("%s", facebook_account.login_name);
printf("Input password: ");
scanf("%s", facebook_account.password);
```

3.6 Nested Structures

It is possible to declare a member which is also of type `struct`. We refer to this construct as nested structures. We declare, for example, `employee` as a nested structure below.

```
struct nameTag {
   char first[20];
   char middle[20];
   char last[20];
};

struct employeeTag {
    int IDnumber;
    struct nameTag name;
    struct dateTag birthdate;
};

struct employeeTag employee;
```

The corresponding parent-child diagram for the `employee` nested structure is shown in Figure 3.3. The diagram depicts that the structure (parent) `employee` has three members (children), namely `IDnumber`, `name` and `birthdate`. The last two members are also structures (parents) on their own. In particular, the members (children) of `name` are `first`, `middle` and `last`.

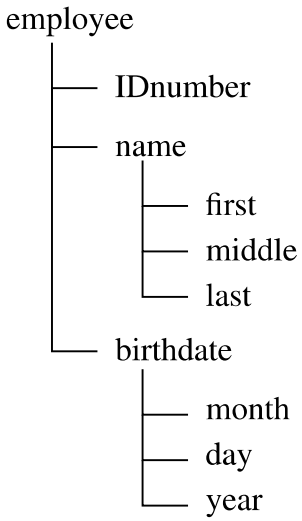

Figure 3.3: Graphical representation of the `employee` nested structure

Members of nested structures can then be accessed using the *structure member operator*. We show how to initialize the `employee` variable in the following code snippet.

```
employee.IDnumber = 123456;
strcpy(employee.name.first, "Jose");
strcpy(employee.name.middle, "Cruz");
strcpy(employee.name.last, "Santos");
employee.birthdate.month = 10;
employee.birthdate.day = 26;
employee.birtdate.year = 1980;
```

Alternatively, we can use `scanf()` to input the `employee` members. We omitted the `printf()` prompts for brevity in the following code.

```
scanf("%d", &employee.IDnumber);
scanf("%s", employee.name.first);
scanf("%s", employee.name.middle);
scanf("%s", employee.name.last);
scanf("%d", &employee.birthdate.month);
scanf("%d", &employee.birthdate.day);
scanf("%d", &employee.birtdate.year);
```

► **Self–Directed Learning Activity** ◄

1. Encode the program incorporating the code snippets above. Compile and test the codes.

2. Assume the following declarations:

```
struct A {
   int i;
   float f;
};

struct B {
   int x;
   double d;
};

struct C {
    int y;
    struct A a;
    struct B b;
};

struct C c;
```

   (a) Initialize the contents of variable `c` using direct assignments.

   (b) Initialize the contents of variable `c` using `scanf()`.

3.7 Structure to Structure Assignment

Let us say that we want to assign the value of `today` to `birthdate`. We can accomplish this by assigning each member of `today` to the corresponding member of `birthdate`, i.e.,

```
birthdate.month = today.month;
birthdate.day = today.day;
birthdate.year = today.year;
```

If a structure is made up of m number of members, then it would require m number of assignment statements to do this.

A better way to achieve the same effect is to assign a structure to another structure using the usual assignment operator. For example

```
birthdate = today;
```

As a rule, a structure to structure assignment is possible only if the variables involved have the same data type. If they are not, a syntax error will be reported by the compiler. In gcc, the corresponding error message is "error: incompatible types in assignment". The following is an example of an invalid assignment

```
landline = today;
```

because `struct phoneTag` is not the same as the data type `struct dateTag`.

► Self–Directed Learning Activity ◄

1. Verify that the assignment `birthdate = today;` is syntactically correct using the program in Listing 3.1.
2. Verify that the assignment `landline = today;` results into a compiler error.
3. Assume the following declarations:

```
struct A {
   int i;
   float f;
};

struct B {
   int i;
   float f;
};

struct C {
   float f;
   int i;
};

struct D {
```

```
    struct A a;
    struct C c;
};

struct A a1, a2;
struct B b1, b2;
struct C c1, c2;
struct D d1, d2;
```

Write the word VALID if the assignment statement below is syntactically correct, otherwise write INVALID.

(a) `a2.i = a1.i;`

(b) `b1.f = a2.f;`

(c) `b2.f = a1.i;`

(d) `c1.f = a1.f;`

(e) `a2 = a1;`

(f) `b1 = b2;`

(g) `a1 = b1;`

(h) `b2 = a1;`

(i) `c1 = a2;`

(j) `b1 = c2;`

(k) `d1.a = a1;`

(l) `d2.c.f = a2;`

(m) `c2 = d2.a;`

3.8 Passing a Structure as Function Parameter

A structure can be passed as a function parameter just like variables of simple data type. This is in consonance with the fact that structure to structure assignment is one of the operations allowed with structures.

Listing 3.2 shows how to define a function that has a structure as a parameter, and how the function can be called. It should be noted that the structure template must be declared prior to the declaration or definition of functions referring to the structure tag–name.

The `main()` function calls `PrintDate(today)`. The actual parameter `today` is assigned as the value of the formal parameter `myDate`.

Listing 3.2: Structure as a Function Parameter

```
1  #include <stdio.h>
2
3  /* take note of this global declaration */
4  struct dateTag {
5      int month;
6      int day;
7      int year;
8  };
9
10 void PrintDate(struct dateTag myDate)
11 {
12     printf("%d/%d/%d", myDate.month, myDate.day, myDate.year);
13 }
14
15 int main()
16 {
17      struct dateTag today;
18
19      today.month = 12;
20      today.day = 25;
21      today.year = 2009;
22
23      printf("Today's date is ");
24      PrintDate(today);
25
26      return 0;
27 }
```

▶ Self–Directed Learning Activity ◀

1. Encode Listing 3.2. Compile and test the program.
2. Add a declaration for `birthdate` variable with a data type of `struct dateTag`. Initialize the contents of `birthdate` using `scanf()`. Thereafter, call `PrintDate()` with `birthdate` as actual parameter.
3. Write a new function `int Equal(struct dateTag date1, struct dateTag date2)` which will return 1 if the two dates are equal. Otherwise, it should return 0. Assume that the two parameters have already been initialized before calling this function.
4. Write a new function `int Latest(struct dateTag date1, struct dateTag date2)` which will return 0 if the first parameter is the latest date between the two parameters, otherwise it should return a 1.

3.9 Function Returning a Structure

Functions that return a structure can also be defined. The value returned by such a function can be assigned to a recipient structure variable.

Listing 3.3 shows an example of how to define a function returning a structure. The return value of `GetDate()` is assigned in the `main()` function to another variable named `today`. Note that function `GetDate()` has a data type of `struct dateTag`.

Listing 3.3: Function Returning a Structure

```
1  #include <stdio.h>
2
3  /* take note of this global declaration */
4  struct dateTag {
5      int month;
6      int day;
7      int year;
8  };
9
10 struct dateTag GetDate(void)
11 {
12     struct dateTag myDate;
13
14     printf("Input month, day and year: ");
```

```
15      scanf("%d %d %d", &myDate.month, &myDate.day, &myDate.year);
16
17      return myDate;
18  }
19
20  void PrintDate(struct dateTag myDate)
21  {
22      printf("%d/%d/%d", myDate.month, myDate.day, myDate.year);
23  }
24
25  int main()
26  {
27       struct dateTag today;
28
29       today = GetDate();
30       printf("Today's date is ");
31       PrintDate(today);
32
33       return 0;
34  }
```

3.10 Pointers and Structures

3.10.1 Address of a Structure Variable

The address of a structure variable can be determined using the address–of operator `&`. For example, the addresses of the variables declared in Listing 3.1 will be displayed in the following `printf()` statements.

```
printf("&landline = %p\n.", &landline);
printf("&email_account = %p\n.", &email_account);
printf("&facebook_account = %p\n.", &facebook_account);
printf("&today = %p\n.", &today);
printf("&birthdate = %p\n.", &birthdate);
```

3.10.2 Pointer to a Structure

The address of a structure variable can be assigned to a structure pointer variable. Thereafter, the pointer variable can be used to access the members of the structure indirectly.

Listing 3.4 shows how to declare a structure pointer variable named `ptr`. It is basically the same as declaring a pointer to a simple data type. Variable `ptr` is then initialized as `ptr = &today`. Thereafter, the members of `today` are accessed indirectly by dereferencing `ptr`. Note that `*ptr` is the indirect expression corresponding to `today`. Try to see the similarity and the difference of an earlier program in Listing 3.2 with the program below.

Listing 3.4: Pointer to Structure

```
1  #include <stdio.h>
2
3  struct dateTag {
4      int month;
5      int day;
6      int year;
7  };
8
9  void PrintDate(struct dateTag myDate)
10 {
11     printf("%d/%d/%d", myDate.month, myDate.day, myDate.year);
12 }
13
14 int main()
15 {
16      struct dateTag today;
17      struct dateTag *ptr;   /* pointer to structure dateTag */
18
19      ptr = &today;        /* initialize pointer variable */
20      (*ptr).month = 12;  /* access today members indirectly */
21      (*ptr).day = 25;
22      (*ptr).year = 2009;
23
24      printf("Date today is ");
25      PrintDate(today);
26      printf("\n");
27
28      return 0;
29 }
```

Table 3.1: Direct and Indirect Access to Structure and its Members

Direct Access	Indirect Access
`today`	`*ptr`
`today.month`	`(*ptr).month`
`today.day`	`(*ptr).day`
`today.year`	`(*ptr).year`

Table 3.1 summarizes the direct access and the indirect access to the structure and its members. The `.` has a higher priority than `*`. This the reason why `*ptr` had to be enclosed within a pair of parentheses.

► Self–Directed Learning Activity ◄

1. Encode Listing 3.4. Compile and test the program.
2. In the function call to `PrintDate()` replace the parameter today with `*ptr`. Will it work?
3. What will happen if the initialization `ptr = &today;` is removed from the program?
4. Using the original codes of Listing 3.4, try to see what will happen if the parentheses enclosing `*ptr` are removed.
5. Refer to the declaration of `struct employeeTag employee` in Section 3.6. Create a C program that will do the following in sequence:
   (a) Declare a structure pointer variable of type `struct employeeTag *`.
   (b) Initialize the pointer variable with the address of `employee`.
   (c) Initialize the members of `employee` indirectly by dereferencing the pointer variable.

3.10.3 Structure Pointer Operator

The *structure pointer operator* denoted by `->`, i.e. a dash immediately followed by a greater than symbol, is used to access a member indirectly via a pointer. Note that it can only be used by pointers to structure variables. The syntax for using the *structure pointer operator* is

```
<structure-pointer-var-name> -> <member-name>
```

The indirect initialization of `today` can then be rewritten as:

```
ptr->month = 12;   /* original: (*ptr).month = 12;  */
ptr->day = 25;     /* original: (*ptr).day = 25;    */
ptr->year = 2009;  /* original: (*ptr).year = 2009; */
```

where `(*ptr).member-name` is equivalent to `ptr->member-name`. Experienced C programmers prefer the `->` notation because it is shorter to write, and "easier" to read.

Table 3.2: Indirect Access via `*` and `->` Operators

Direct Access	Indirect Access via `*`	Indirect Access via `->`
`today`	`*ptr`	not applicable
`today.month`	`(*ptr).month`	`ptr->month`
`today.day`	`(*ptr).day`	`ptr->day`
`today.year`	`(*ptr).year`	`ptr->year`

Table 4.1 summarizes the direct access and the two alternative ways for indirect access to the structure's members.

► Self–Directed Learning Activity ◄

1. Modify Listing 3.4 by replacing the initialization of variable `today` using the *structure pointer operator* `->`. Compile and test the program.

2. Using the codes in the previous item, find out what will happen if a space is inserted between `-` and `>` symbol.

3. Refer to the declaration of `struct employeeTag employee` in Section 3.6. Create a C program that will do the following in sequence:

(a) Declare a structure pointer variable of type `struct employeeTag *`.

(b) Initialize the pointer variable with the address of `employee`.

(c) Initialize the members of `employee` indirectly by using the structure pointer operator.

3.10.4 Pointer to a Structure as Parameter

We learned in Section 3.8 that structures can be passed as parameters to functions. If the size of the structure is large, it may not be a good idea to pass the entire structure because of space and time considerations. It will be faster and economical (in terms of memory space) to pass instead a pointer to a structure as function parameter.

The more important reason for passing a pointer to structure is the need to change the values of the members of the structure outside of the function where the structure variable was declared.

Listing 3.5 shows an example program which passes a pointer to structure as parameter. In functions `InputDate()` and `OutputDate()`, the formal parameter is `ptr` which has a data type of `struct dateTag *`. These functions are called in `main()` with the address of `today`, i.e., `&today` as parameter.

Try to recall Listing 3.3 and compare it with Listing 3.5. In particular, notice the difference between `GetDate()` with `InputDate()`. Compare also `PrintDate()` with `OutputDate()`.

Listing 3.5: Passing a Structure Pointer as Parameter

```
1  #include <stdio.h>
2
3  /* take note of this global declaration */
4  struct dateTag {
5      int month;
6      int day;
7      int year;
8  };
9
10 void InputDate(struct dateTag *ptr)
11 {
12     printf("Input month, day and year: ");
13     scanf("%d %d %d", &ptr->month, &ptr->day, &ptr->year);
14 }
```

```
15
16 void OutputDate(struct dateTag *ptr)
17 {
18     printf("%d/%d/%d", ptr->month, ptr->day, ptr->year);
19 }
20
21 int main()
22 {
23      struct dateTag today;
24
25      InputDate(&today);
26      printf("Today's date is ");
27      PrintDate(&today);
28
29      return 0;
30 }
```

► Self–Directed Learning Activity ◄

Refer to the declaration of `struct employeeTag employee` in Section 3.6. Create a C program such that it contains the definitions for the following functions:

1. `void InputEmployee(struct employeeTag *ptr)`

   This function will be used to input the values of the members of the structure.

2. `void OutputEmployee(struct employeeTag *ptr)`

   This function will be used to output the values of each member of the structure.

3. `int main()`

   This function will declare a structure variable named `employee` of type `struct employeeTag`. It should call the functions in-charge for input and output of `employee` member values.

3.11 Array of Structures

Static memory allocation of multiple instances of structures other than listing the names of instances separated by a comma is possible. This is done by declaring an array of structure.

Consider for example how to declare 5 instances of `struct dateTag`. One of way of doing this is to have 5 separate variables declared as follows:

```
struct dateTag date1, date2, date3, date4, date5;
```

If the number of instances is increased, say to 1000, this kind of declaration will no longer be practical. A better approach is to declare a 1D array of structure. The syntax for an array of structure declaration is the same with how we declare 1D array of simple data types. For example, to declare 5 instances as an array, we write:

```
struct dateTag dateArray[5];
```

In this declaration, each element of `dateArray` is a structure. We will need to use both array indexing and structure member operator to access a particular member of a structure. The expression `dateArray[i]` is the `i`'th element of the array. To access the member `month` of this element, we write `dateArray[i].month`.

The following code snippet show an example of how to initialize the first array element structure members.

```
dateArray[0].month = 12;
dateArray[0].day = 25;
dateArray[0].year = 2009;
```

Notice that array indexing is performed first before the structure member operation.

If we want to input the members of the `i`'th element we would have to write:

```
scanf("%d %d %d", &dateArray[i].month, &dateArray[i].day,
       &dateArray[i].year);
```

The expression `&dateArray[i].month` is quite involved since there are three operators present. The operations are applied in the following order: first array indexing `[]`, followed by structure member operator `.`, and finally address–of operator `&`.

Listing 3.6 shows a complete program that illustrates an application of the concepts that we have learned so far. The name of the array[2] is passed as parameter to functions `InputDateArray()` and `OutputDateArray()`. Parameter `n` represents the number array elements.

Listing 3.6: Array of Structure

```
1  #include <stdio.h>
2
3  /* take note of this global declaration */
4  struct dateTag {
5      int month;
6      int day;
7      int year;
8  };
9
10 void InputDateArray(struct dateTag dateArray[], int n)
11 {
12     int i;
13
14     for (i = 0; i < n; i++)
15     {
16         printf("Input month, day and year of element %d: ", i);
17         scanf("%d %d %d", &dateArray[i].month,
18                &dateArray[i].day, &dateArray[i].year);
19     }
20 }
21
22 void OutputDateArray(struct dateTag dateArray[], int n)
23 {
24     int i;
25     for (i = 0; i < n; i++)
26          printf("Element %d is %d/%d/%d\n", i,
27                  dateArray[i].month, dateArray[i].day,
28                  dateArray[i].year);
29 }
30
31 int main()
32 {
33      struct dateTag dateArray[5];
34
35      InputDateArray(dateArray, 5);
36      OutputDateArray(dateArray, 5);
```

[2]Recall that the name of the array is synonymous with the base address of the array.

```
37
38      return 0;
39  }
```

Table 3.3: Accessing an Element of an Array of Structures

[ ] Notation	* Notation	-> Notation
A[i]	*(A + i)	not applicable
A[i].member-name	(*(A + i)).member-name	(A + i)->member-name

Table 3.3 summarizes the three alternative ways of accessing a structure and its members. The structure in this case is an element with index `i` in an array of structures which we named as `A` in the table.

► Self–Directed Learning Activity ◄

1. Encode and test the program in Listing 3.6.
2. We learned in Chapter 2 (Arrays) that an array element can be accessed via pointer dereference. Replace `dateArray[i]` above with `*(dateArray + i)`. Compile and run the program to see that you'll get the same results.
3. Create a new C program that applies that same idea using `struct employeeTag`.

3.12 `typedef` and `struct` type

`typedef` can be used to declare an alias for a `struct` data type. It can be used with Variation 1 (unnamed structure with no instance) of the structure declaration syntax. For example, the code below

```
typedef struct {
   char ch;
   int i;
   float f;
   double d;
} sampleType;
```

declares `sampleType` as an alias for the unnamed `struct` type.[3]

`typedef` can also be used together with Variation 2, i.e., named structure with no instance. For example, the following is a two part declaration.

```
struct sampleTag {
   char ch;
   int i;
   float f;
   double d;
};

typedef struct sampleTag sampleType;
```

The first part declares the structure template, thereafter the second part declares an alias for the structure template. These two separate declarations can be combined into one declaration as:

```
typedef struct sampleTag {
   char ch;
   int i;
   float f;
   double d;
} sampleType;
```

[3]The author actually finds this kind of declaration humorous. Why? Because we are declaring an alias for an "unnamed" structure. The structure does not have a (tag) name, yet we are giving it an alias! :-)

Structure variables may then be declared using the alias as data type. For example:

```
struct sampleTag s;
sampleType t;
```

declares a structure variable `s` using the structure template, while structure variable `t` was declared using the alias. Variables `s` and `t` have the same data type. Thus, the assignment `t = s;` will not result into a data type mismatch error.

Alias declared with `typedef` can also be used to declare structure pointer variables. For example:

```
sampleType *ptr;
```

We show how to use `typedef` in an actual program by rewriting the codes from Listing 3.3. The modifed codes (i.e., with `typedef`) are shown in Listing 3.7. The original program used `struct dateTag`. The modified program replaced the declarations of variables and parameters parameters and the definition of function `GetDate()` with the alias `dateType`.

Listing 3.7: `typedef` and `struct`

```
1  #include <stdio.h>
2
3  struct dateTag {
4      int month;
5      int day;
6      int year;
7  };
8
9  typedef struct dateTag dateType;
10
11 dateType GetDate(void)
12 {
13     dateType myDate;
14
15     printf("Input month, day and year: ");
16     scanf("%d %d %d", &myDate.month, &myDate.day, &myDate.year);
17
18     return myDate;
19 }
20
```

```
21 void PrintDate(dateType myDate)
22 {
23     printf("%d/%d/%d", myDate.month, myDate.day, myDate.year);
24 }
25
26 int main()
27 {
28      dateType today;
29
30      today = GetDate();
31      printf("Today's date is ");
32      PrintDate(today);
33
34      return 0;
35 }
```

The curious reader, at this junction, is probably asking "So which one is better, use a structure template or use a `typedef` alias?" The author choose not to answer this question with a simple yes or no.

There are groups of programmers who will choose to use a structure template over a `typedef` alias because the template clearly states the `struct` nature of an instance, parameter or function.

Another group of programmers prefer using `typedef` alias primarily because it allows them to write codes that are shorter (which are also easier to type because in general it requires fewer keystrokes).

The recommendation of the author to the reader is to make sure that you become conversant with both styles. In actual practice, you may choose one over the other; just observe consistency of use within the same set of codes.

► Self–Directed Learning Activity ◄

1. Verify that the assignment operation `t = s;` will not produce a data type mismatch error. Recall in the previous discussion that the variables were declared as:

   ```
   struct sampleTag s;
   sampleType t;
   ```

2. Rewrite all the sample programs, and your program solutions to the Self–Directed Learning Activities with a `typedef`.

3.13 Chapter Summary

The key ideas presented in this chapter are summarized below:

- A structure is a group of possibly heterogeneous elements.
- There are four variations for declaring a `struct` data type and structure variable.
- There are only three operations allowed on structure, namely:
  1. access an element of a structure
  2. assign a structure to another structure (includes parameter passing of a structure, and functions that returns a structure)
  3. get the address of a structure variable
- Two new operators were introduced, namely the *structure member operator* and *structure pointer operator*.
- Structures can be allocated using static memory allocation and dynamic memory allocation.
- `typedef` can be used to declare an alias to a structure.

Problems for Chapter 3

Assume the following declarations:

```
struct aTag {
    char ch;
    int i;
};

struct bTag {
    float f;
    double d;
};

struct cTag {
    struct aTag a;
    struct bTag b;
    int A[5];
```

```
};

typedef struct aTag aType;
typedef struct bTag bType;
typedef struct cTag cType;

aType a, A[5]
bType b, B[5];
cType c, C[5];

aType *pa;
bType *pb;
cType *pc;
```

Problem 3.1. What is the data type of the following expressions? DO NOT use the alias as answer. Write the word INCORRECT if the expression is incorrect.

1. a	21. &b.f	41. (*pc).b	61. &pb->f	81. C[2].b
2. b	22. &b.d	42. (*pc).A[3]	62. &pb->d	82. C[2].A[1]
3. c	23. &c.a	43. *pc.A[2]	63. &pc->a	83. C[0].a.ch
4. a.ch	24. &c.b	44. (*pc).a.ch	64. &pc->b	84. C[0].a.i
5. a.i	25. &c.a.ch	45. (*pc).a.i	65. &pc->A[4]	85. C[0].b.f
6. b.f	26. &c.a.i	46. (*pc).b.f	66. &pc->a.ch	86. C[0].b.d
7. b.d	27. &c.b.f	47. (*pc).b.d	67. &pc->a.i	87. C[0].A[5]
8. c.a	28. &c.b.d	48. pa->ch	68. &pc->b.f	88. &C[1].a.ch
9. c.b	29. &c.A[0]	49. pa->i	69. &pc->b.d	89. &C[1].b.f
10. c.a.ch	30. &c.A[3]	50. pb->f	70. A[0]	90. &C[1].b.d
11. c.a.i	31. pa	51. pb->d	71. B[4]	91. &C[1].A[2]
12. c.b.f	32. pb	52. pc->a	72. C[2]	92. *(C + 2)
13. c.b.d	33. pc	53. pc->b	73. &A[1]	93. (*(C + 2)).a
14. c.A[0]	34. *pa	54. pc->A[3]	74. &B[3]	94. (C+2)->b
15. c.A[5]	35. *pb	55. pc->a.ch	75. &C[0]	95. (C+3)->a.i
16. &a	36. (*pa).ch	56. pc->a.i	76. A[0].ch	96. &(C+3)->b.f
17. &b	37. (*pa).i	57. pc->b.f	77. A[0].i	97. A[0].i++
18. &c	38. (*pb).f	58. pc->b.d	78. B[1].f	98. B[1]--
19. &a.ch	39. (*pb).d	59. &pa->ch	79. B[1].d	99. C[2].a++
20. &a.i	40. (*pc).a	60. &pa->i	80. C[2].a	100.(C+4)->a.ch--

Problem 3.2. Implement a function `void InputFunc1(cType *ptr)` that will input via `scanf()` the members of the structure pointed to by `ptr`. Assume that the memory space already exist.

Problem 3.3. Implement a function `cType InputFunc2(void)` that will declare a local variable as `cType temp`. The function will then input the values of the members of `temp` using `scanf()`. Finally, the function will return `temp`.

Problem 3.4. Implement a function `void InputFunc3(cType C[], int n)` that will input via `scanf()` the elements of the array of structure `C`. Parameter `n` represents the number of elements in the array.

Problem 3.5. Implement a function `void OutputFunc1(cType *ptr)` that will output the members of the structure pointed to by `ptr`. Assume that the memory space already exist and that the values of the structure are valid. Use dereference operator and structure member operator only. DO NOT use the structure pointer operator `->`.

Problem 3.6. Implement a function `void OutputFunc2(cType *ptr)` similar to the previous problem. The difference here is that it is required to use the structure pointer operator `->`.

Note: In the following problems the second parameter `int n` *represents the number of elements in an array.*

Problem 3.7. Implement a function `void OutputFunc3(cType C[], int n)` that will output the elements of array C using array indexing.

Problem 3.8. Implement a function `void OutputFunc4(cType C[], int n)` that will output the elements of array C using structure pointer operator instead of array indexing.

Problem 3.9. Implement a function `int Total1(aType A[], int n)` which will return the sum of the member `i` of the structures in array A.

Problem 3.10. Implement a function `int Total2(cType c)` which will return the sum of structure `c`'s member array `A`.

Problem 3.11. Implement a function `int Total3(cType C[], int n)` which will return the sum of *ALL* the elements of member array `A` for all structures. Note: the previous problem computes the sum from just one structure. This problem computes the sum of the member `A` for all elements of `C`.

Problem 3.12. Implement a function `int Minimum(cType c)` which will return

the index of the smallest value in `c`'s member array `A`.

Problem 3.13. Implement a function `float fMinimum(bType B[], int n)` which will return the index of the element in array B whose member `f` is the smallest.

Problem 3.14. Implement a function `double dMaximum(bType B[], int n)` which will return the index of the element in array B whose member `d` is the largest.

Problem 3.15. Implement a function `void SortFunc1(aType A[], int n)` that will sort the elements of array `A` in *increasing* order based on member `i`. Use the straight selection sort algorithm discussed previously.

Problem 3.16. Implement a function `void SortFunc2(bType B[], int n)` that will sort the elements of array `B` in *decreasing* order based on member `d`. Use the straight selection sort algorithm discussed previously.

Problem 3.17. Implement a function `void CopyFunc(cType sourceArr[], cType destArr[], int n)` that will copy elements of the `sourceArr` array to the `destArr` array (same index). Parameter `n` is the number of elements of the arrays.

References

Consult the following resources for more information.

1. comp.lang.c Frequently Asked Questions. `http://c-faq.com/`

2. Kernighan, B. and Ritchie, D. (1998). *The C Programming Language, 2nd Edition.* Prentice Hall.