A First Book of ANSI C

Fourth Edition

Chapter 12
Structures
Objectives

• Single Structures
• Arrays of Structures
• Passing and Returning Structures
• Unions (Optional)
• Common Programming and Compiler Errors
Introduction

• Each data item listed in Figure 12.1 is an entity by itself, called a **data field**
• Together, all the data fields form a single unit called a **record**
  – In C, a record is referred to as a **structure**
Introduction (continued)

Name: 
Type: 
Location in Dungeon: 
Strength Factor: 
Intelligence Factor: 
Type of Armor: 

Figure 12.1 Typical components of a video game character
Introduction (continued)

• A structure’s **form** consists of the symbolic names, data types, and arrangement of individual data fields in the record

• The structure’s **contents** consist of the actual data stored in the symbolic names
Introduction (continued)

Name: Golgar
Type: Monster
Location in Dungeon: G7
Strength Factor: 78
Intelligence Factor: 15
Type of Armor: Chain Mail

Figure 12.2 The form and contents of a structure
Single Structures

• **Structure definition in C:**

```c
struct {
    int month;
    int day;
    int year;
} birth;
```

– Reserves storage for the individual data items listed in the structure

– The three data items are the **members of the structure**

• **Assigning actual data values to the data items of a structure is called** **populating the structure**
Single Structures (continued)

Program 12.1

```c
#include <stdio.h>
int main()
{
    struct
    {
        int month;
        int day;
        int year;
    } birth;

    birth.month = 12;
birth.day = 28;
birth.year = 1987;
printf("My birth date is %d/%d/%d\n",
birth.month,birth.day,birth.year % 100);
return 0;
}
```
Single Structures (continued)

- Multiple variables can be defined in one statement
  ```c
  struct {int month; int day; int year;} birth, current;
  ```
- Common to list the form of the structure with no following variable names
  - The list of structure members must be preceded by a user-selected `structure type name`
    ```c
    struct Date
    {
      int month;
      int day;
      int year;
    };
    ```
Single Structures (continued)

Program 12.2

```c
#include <stdio.h>
struct Date {
    int month;
    int day;
    int year;
};

int main() {
    struct Date birth;
    birth.month = 12;
    birth.day = 28;
    birth.year = 1987;
    printf("My birth date is %d/%d/%d\n",
           birth.month, birth.day, birth.year & 100);
    return 0;
}
```

By convention the first letter of user-selected structure type names is uppercase.
Single Structures (continued)

• Initialization of structures follows the same rules as for the initialization of arrays:
  - struct Date birth = {12, 28, 1987};

• Structure members can be of any data type

```c
struct PayRecord
{
  char name[20];
  int idNum;
  double regRate;
  double otRate;
};

struct PayRecord employee = {"H. Price", 12387, 15.89, 25.50};
```
Single Structures (continued)

- Advantage of structures is when the same structure type is used in a list many times
- Individual members can be arrays and structures

```c
struct
{
    char name[20];
    struct Date birth;
} person;
```

- **Example**: `person.name[4]`
Arrays of Structures

<table>
<thead>
<tr>
<th>Employee number</th>
<th>Employee name</th>
<th>Employee pay rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>32479</td>
<td>Abrams, B.</td>
<td>6.72</td>
</tr>
<tr>
<td>33623</td>
<td>Bohm, P.</td>
<td>7.54</td>
</tr>
<tr>
<td>34145</td>
<td>Donaldson, S.</td>
<td>5.56</td>
</tr>
<tr>
<td>35987</td>
<td>Ernst, T.</td>
<td>5.43</td>
</tr>
<tr>
<td>36203</td>
<td>Gwodz, K.</td>
<td>8.72</td>
</tr>
<tr>
<td>36417</td>
<td>Hanson, H.</td>
<td>7.64</td>
</tr>
<tr>
<td>37634</td>
<td>Monroe, G.</td>
<td>5.29</td>
</tr>
<tr>
<td>38321</td>
<td>Price, S.</td>
<td>9.67</td>
</tr>
<tr>
<td>39435</td>
<td>Robbins, L.</td>
<td>8.50</td>
</tr>
<tr>
<td>39567</td>
<td>Williams, B.</td>
<td>7.20</td>
</tr>
</tbody>
</table>

Figure 12.3  A list of employee data
Arrays of Structures (continued)

<table>
<thead>
<tr>
<th>1st Structure</th>
<th>Employee Number</th>
<th>Employee Name</th>
<th>Employee Pay Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32479</td>
<td>Abrams, B.</td>
<td>6.72</td>
</tr>
<tr>
<td>2nd Structure</td>
<td>33623</td>
<td>Bohm, P.</td>
<td>7.54</td>
</tr>
<tr>
<td>3rd Structure</td>
<td>34145</td>
<td>Donaldson, S.</td>
<td>5.56</td>
</tr>
<tr>
<td>4th Structure</td>
<td>35987</td>
<td>Ernst, T.</td>
<td>5.43</td>
</tr>
<tr>
<td>5th Structure</td>
<td>36203</td>
<td>Gwodz, K.</td>
<td>8.72</td>
</tr>
<tr>
<td>6th Structure</td>
<td>36417</td>
<td>Hanson, H.</td>
<td>7.64</td>
</tr>
<tr>
<td>7th Structure</td>
<td>37634</td>
<td>Monroe, G.</td>
<td>5.29</td>
</tr>
<tr>
<td>8th Structure</td>
<td>38321</td>
<td>Price, S.</td>
<td>9.67</td>
</tr>
<tr>
<td>9th Structure</td>
<td>39435</td>
<td>Robbins, L.</td>
<td>8.50</td>
</tr>
<tr>
<td>10th Structure</td>
<td>39567</td>
<td>Williams, B.</td>
<td>7.20</td>
</tr>
</tbody>
</table>

**Figure 12.4** A list of records
Arrays of Structures (continued)

Program 12.3

```c
#include <stdio.h>
#define NUMRECS 5
struct PayRecord /* construct a global structure type */
{
    int id;
    char name[20];
    double rate;
};

int main()
{
    int i;
                                            {33623, "Bohm, P.", 7.54},
                                            {34145, "Donaldson, S.", 5.56},
                                            {35987, "Ernst, T.", 5.43},
                                            {36203, "Gwodz, K.", 8.72}};
    for (i = 0; i < NUMRECS; i++)
        printf("%d %-20s %4.2f\n",
               employee[i].id, employee[i].name, employee[i].rate);
    return 0;
}
```

Inner braces are not necessary
Arrays of Structures (continued)

• Without explicit initializers, the numeric elements of both static and external arrays or structures are initialized to 0 (or nulls)
• An inferior alternative to an array of structures is parallel arrays
  – **Parallel arrays** are two or more arrays, where each array has the same number of elements and the elements in each array are directly related by their position in the arrays
  – They are rarely used any more
Passing and Returning Structures

- Individual structure members may be passed to a function in the same manner as any scalar variable
  - `display(emp.idNum)`
  - `calcPay(emp.payRate, emp.hours)`;

- On most compilers, complete copies of all members of a structure can also be passed to a function by including the name of the structure as an argument to the called function
  - `calcNet(emp)`;
Although the structures in `main()` and `calcNet()` use the same globally defined structure type, this is not strictly necessary (although it is preferable)

```c
#include <stdio.h>
struct Employee /* declare a global structure type */
{
    int idNum;
    double payRate;
    double hours;
};

double calcNet(struct Employee); /* function prototype */

int main()
{
    struct Employee emp = {6787, 8.93, 40.5};
    double netPay;
    netPay = calcNet(emp); /* pass copies of the values in emp */
    printf("The net pay of employee %d is \$%6.2f\n", emp.idNum, netPay);
    return 0;
}

double calcNet(struct Employee temp)
/* temp is of data type struct Employee */
{
    return (temp.payRate * temp.hours);
}
```
Passing and Returning Structures (continued)

• A structure can be passed by reference
  - `calcNet(&emp);`
  - double `calcNet(struct Employee *pt)`
  - `(*pt).idNum` or `*pt->idNum`
Passing and Returning Structures (continued)

Figure 12.5 A pointer can be used to access structure members
Passing and Returning Structures (continued)

```
#include <stdio.h>
struct Employee /* declare a global structure type */ {
    int idNum;
    double payRate;
    double hours;
};

double calcNet(struct Employee *); /* function prototype */

int main() {
    struct Employee emp = {6787, 8.93, 40.5};
    double netPay;
    netPay = calcNet(&emp); /* pass an address*/
    printf("The net pay for employee %d is $%6.2f\n", emp.idNum, netPay);
    return 0;
}

double calcNet(struct Employee *pt) /* pt is a pointer to a */ {
    /* structure of Employee type */
    return (pt->payRate * pt->hours);
}
```
Passing and Returning Structures (continued)

• ++ and -- can be applied to structures
  - ++pt->hours
  - (pt++)->hours
  - (++pt)->hours
Passing and Returning Structures (continued)

![Diagram showing how pointer addresses change with increment and decrement operations.]

**Figure 12.6** Changing pointer addresses
Returning Structures

Program 12.6

```c
#include <stdio.h>

struct Employee /* declare a global structure type */
{
  int idNum;
  double payRate;
  double hours;
};

struct Employee getValues(); /* function prototype */
```
returning structures (continued)

```c
11 int main()
12 {
13       struct Employee emp;
14
15       emp = getValues();
16       printf("\nThe employee id number is %d\n", emp.idNum);
17       printf("The employee pay rate is $%5.2f\n", emp.payRate);
18       printf("The employee hours are %5.2f\n", emp.hours);
19
20       return 0;
21 }
22
23 struct Employee getValues()
24 {
25       struct Employee newemp;
26
27       newemp.idNum = 6789;
28       newemp.payRate = 16.25;
29       newemp.hours = 38.0;
30
31       return (newemp);
32 }
```
Unions

- A **union** is a data type that reserves the same area in memory for two or more variables
  ```c
  union
  {
    char key;
    int num;
    double price;
  } val;
  ```
  - Each of these types, but only one at a time, can actually be assigned to the union variable
  - A union reserves sufficient memory locations to accommodate its largest member's data type
Unions (continued)

- Individual union members are accessed using the same notation as structure members
- Typically, a second variable keeps track of the current data type stored in the union

```c
switch(uType)
{
    case 'c': printf("%c", val.key); break;
    case 'i': printf("%d", val.num); break;
    case 'd': printf("%f", val.price); break;
    default : printf("Invalid type : %c", uType);
}
```
Unions (continued)

- A type name can be associated with a union to create templates
  ```c
  union DateTime {
    long days;
    double time;
  };
  union DateTime first, second, *pt;
  ```
- Pointers to unions use the same notation as pointers to structures
- Unions may be members of structures and arrays; structures, arrays, and pointers may be members of unions
Unions (continued)

```c
struct
{
    char uType;
    union
    {
        char *text;
        double rate;
    } uTax;
} flag;
```

- `rate` is referenced as `flag.uTax.rate`
- The first character of the string whose address is stored in the pointer `text` is accessed as `*flag.uTax.text`
Common Programming Errors

• Attempting to use structures and unions, as complete entities, in relational expressions
• Assigning an incorrect address to a pointer that is a member of a structure or union
• Storing one data type in a union and accessing it by the wrong variable name can result in an error that is particularly troublesome to locate
## Common Compiler Errors

<table>
<thead>
<tr>
<th>Error</th>
<th>Typical Unix-based Compiler Error Message</th>
<th>Typical Windows-based Compiler Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the wrong type of braces when declaring a structure. For example:</td>
<td>The following error will be reported on each line containing a brace: (S) Syntax error.</td>
<td>:error: syntax error : missing ';' before '[' : error: syntax error : missing ']' before ' ';</td>
</tr>
<tr>
<td>struct [</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int month;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int day;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int year;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>] birth;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attempting to initialize the elements of a structure inside the declaration. For example:</td>
<td>S) Syntax error: possible missing ';' or ','?</td>
<td>:error: 'month' : only const static integral data members can be initialized inside a class or struct</td>
</tr>
<tr>
<td>struct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int month = 6;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int day;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int year;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>} birth;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assigning a pointer to a structure rather than the address of the structure. For example:</td>
<td>(S) Operation between types <em>struct Date</em>* and <em>struct Date</em> is not allowed.</td>
<td>:error: '=' : cannot convert from 'Date' to 'Date *'</td>
</tr>
<tr>
<td>int main()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>struct Date *ptr;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>struct Date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>birth;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ptr = birth;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

- A structure allows individual variables to be grouped under a common variable name.
- A structure type name can be used to create a generalized structure type describing the form and arrangement of elements in a structure.
- Structures are particularly useful as elements of arrays.
Summary (continued)

• Individual members of a structure are passed to a function in the manner appropriate to the data type of the member being passed.
• Structure members can be any valid C data type, including structures, unions, arrays, and pointers.
• Unions are declared in the same manner as structures.