Structures - Chapter Summary

1 What is a structure?

A structure is a collection of related data items, which can be of different types.

An every-day analogue to a structure in C would be a card in a card index. This might, in the case of a card in a library index, contain such information as: the title of a book, the author's name, the ISBN, the accession number and the book's shelf reference. Some of these items of information would be in the form of strings, some in the form of integers and some might be in the form of floating-point numbers, but they would all refer to the same book and be placed on the same index card.

Another way of looking at a structure would be as an array in which the elements, being of possibly different types, may require different amounts of storage space. Whereas it is possible, once the type of an array is known, to find the address of any element by moving the required number of uniform storage units along from the starting address, it would only be possible to find the address of an item in a structure if not only the number but also the type of each element were known.

Clearly, this information must be declared when any structure is created, so that sufficient storage space in memory can be allotted, and so that any data contained in the structure may be accessed.

2 Setting up a Structure Template

The first step in creating a structure, therefore, is to declare a structure template. This defines the type of the structure and allocates a template name for this type. The template declares the type and name of each data item, often referred to as a member, to be stored in the structure.

The following template declares a structure type called `licence`:

```
struct licence
{
     char name[30];
     int age;
     char vehicle_class;
};
```

A structure of this type contains three members, called `name`, `age` and `vehicle_class`. The first is an array of type `char`, the second is of type `int` and the third of type `char`.

Note the helpful, but not universally-adopted, convention of capitalising the first letter of the template name to distinguish it from a structure data name.

3 Declaring Structure Variables

Once a structure type has been defined by declaring a template, individual structures of that type may be declared.

Structure data items may be initialised either by separate assignment statements or as part of the initial declaration, as in the following example:

```
struct licence my_licence = { "A.C.Programmer", 27, 'A' };
```

This is a declaration for a structure variable called `my_licence` of type `struct licence`. Each member of the structure must be assigned a value of the correct type. In the case of `my_licence`, the first member of the structure is assigned a character string, the second an integer and the third a character constant.

4 Accessing Structure Members

A structure member may be accessed by using the operator `'.'`, which is known as the member operator, to link the name of the `struct` to the member name. The name of an individual member of a specific structure is therefore written:

`struct_data_name.member_name`

Continuing the use of the structure variable `my_licence`, the three members in this structure could be given new values thus:

```
strcpy(my_licence.name, "Richard Triance");
my_licence.age = 57;                /* educated guess */
my_licence.vehicle_class = 'C';     /* naturally! */
```

The members of a structure may be treated as data items in their own right and used in expressions and as arguments to functions that take data of the corresponding type, as in the following examples:

```
if (my_licence.age < retirement_age)
{
     years_to_retirement(
               retirement_age - my_licence.age);
}
printf("My name is %s.", my_licence.name);
```

5 Arrays of Structures

The following example features a representation of the first six chemical elements in the Periodic Table as an array of structures. Each element of the array is a structure of four members, which contain information about a particular chemical element.

```
struct element            /* Stage 1 - the template */
{
     char name[15];
     char symbol[3];
     int at_no;
     float at_wt;
};


struct element table[6] =
{                         /* Stage 2 - the definition */

     { "Hydrogen", "H",  1,  1.0 }, /* and initialisation  */
     { "Helium",   "He", 2,  4.0 }, /* of an array of 6    */
     { "Lithium",  "Li", 3,  6.9 }, /* structures of type  */
     { "Beryllium","Be", 4,  9.0 }, /* struct element */
     { "Boron",    "B",  5, 10.8 },
     { "Carbon",   "C",  6, 12.0 }
};

int main(void)
{
     char answer[15];
     int times = 0, count = 0, choice = -1;

     for ( ; times < 6; times++, choice = -1)
     {
          putchar('\n');
          puts("Which element are you interested in?\n");
          scanf("%14s", answer);
          for (count=0; count < 6; count++)
          {
               if (strcmp(answer, table[count].name) == 0)
               {
                    choice = count;
               }
```

```
                }
                if (choice == -1)
                     puts("\n\nSorry - unknown element!");
                else
                {
                     printf("\n Element %s has",
                               table[choice].name);
                     printf("\nsymbol :%s",
                               table[choice].symbol);
                     printf("\natomic number :%d",
                               table[choice].at_no);
                     printf("\nand an atomic weight of :%f",
                               table[choice].at_wt);
                }
           }
           return 0;
     }
```

The program consists of a declaration of a structure template, followed by a declaration and assignment of an array of six structures of that type, followed by declarations of scalar variables. A `for` loop is then used to give the user six opportunities to ask for details of the six elements listed.

6 Nested Structures

C allows all meaningful combinations of structures and arrays, including arrays of structures, structures including arrays as structure members, and structures including other structures as members. There is also no limit in C to the degree of nesting allowed for any of these combinations.

When a structure contains members that are themselves structures, templates and corresponding structure variables must be declared for each type of structure involved. A member of a nested structure may be accessed by linking its name with the member operator `'.'` to the composite name of the two structures that contain it. The composite name itself consists of the names of first the outer and then the inner structure variables, linked with another structure member operator. The name of the member in the nested structure would thus have the general form:

```
outer_struct.inner_struct.member
```

The following program, based on an imaginary railway timetable, demonstrates the nesting of structures within structures.

The structure `Nov25_1997` is a variable of type `struct Roster`, and structures of this type have two members, both of which are themselves structures, the first of type `struct Train` and the second of type `struct Engine`.

The `printf` statements at the end of the program use members of the nested structures as arguments:

```
struct train        /* template for 1st inner structure */
{
     char time[6];
     char from[30];
     char to[30];
     char restaurant;
};

struct engine       /* template for 2nd inner structure */
{
     long engine_no;
     char driver[30];
};


struct roster       /* template for outer structure */
{
     struct train flyer; /* member of type struct Train */
     struct engine hst;  /* member of type struct Engine */
};


int main(void)
{
     struct roster Nov25_2012 =    /* Define variable */
     {
          { "08.45", "Cheltenham", "Paddington", 'B' },
          { 254004, "K.C.Jones" }
     };

     char catering[25];

     switch (Nov25_2012.flyer.restaurant) /* Access */
     {
          case 'R':
               strcpy(catering, "a restaurant car");
```

```
                    break;

               case 'B':
                    strcpy(catering, "a buffet car");
                    break;

               default:
                    strcpy(catering, "no catering facilities");
                    break;
          }

          printf("\nNext train is the %s",
                    Nov25_2012.flyer.time);
          printf("\nfrom %s to %s",
                    Nov25_2012.flyer.from, Nov25_2012.flyer.to);
          printf("\nThere will be %s on this train.", catering);

          return 0;
     }
```

When compiled and run, the program should announce:

```
Next train is the 08.45
from Cheltenham to Paddington.
There will be a buffet car on this train.
```

7 Structures and Functions

7.1 Limitations of Early Implementations of C

In the original definition of C, function arguments were intended to pass only scalar values to functions, and functions were likewise able to return only scalar values. This meant that arguments and function types could be any of the integer, floating-point or character types, but not arrays or structures.

The values of individual structure members of scalar type could, of course, be passed as arguments to functions and returned by functions like any other scalar values.

7.2 Pointers to Structures

Pointers, which will be discussed in the Pointers chapter and following, have always provided a powerful mechanism to effect whole-structure operations. A pointer to a data item evaluates to the address of that data, which, for aggregate types, is the

starting address of the array or structure. As an address is a scalar value, pointers to structures can be passed to and from functions, and thus give them access to the data held in the structures.

7.3. Structures as Arguments in ISO Standard C

The ISO standard for C specifies that whole structures, though not arrays, may now be passed as arguments to functions.

When a structure is used as an argument, the function creates a copy of the structure, assigning the same values to each of the individual members as were assigned in the original. This protects the values held in the original, since the function can only effect changes to the values held in the copy.

Functions can now return structures, so `struct` has become a legal type for functions. This is only possible because whole-structure operations have been implemented in the ISO standard, and it is now possible to assign the values held in one structure to another structure of the same type in a single operation:

```
structure2 = structure1;
```

This statement assigns the value of each member in `structure1` to the corresponding member in `structure2`.

Intentionally left blank