Pointers and Structures - Chapter Summary

1 Structures and Functions

The ISO Standard for C now allows structures to be passed as arguments to functions and returned as values by functions, thus making structure types valid types for functions. This contrasts with earlier implementations of C that did not allow aggregate data types to be used as arguments or return values for functions. In fact, the standard still does not permit the use of arrays in either of these roles.

Under the restrictions previously in force, the only way to pass structures to functions was to use pointers to structures as arguments, thus achieving call by reference. Although this is no longer the sole method available, it remains a useful and powerful programming technique, significantly different from the mechanism used for passing structures as arguments to functions in ISO Standard C.

ISO Standard C uses a call-by-value mechanism for passing structures to functions. The function being called creates a copy of the structure it is passed as an argument, and then assigns to it the values contained in the original structure. This means that the data in that original structure remains unchanged by any operations carried out by the function.

When a structure is returned by a function, the structure within the function is not itself returned to the calling environment. Again, it is the values contained in the structure within the function that are assigned to the equivalent structure in the calling environment.

These operations are possible because ISO Standard C permits whole-structure assignments for structures of the same type. Thus, statements such as

```
structure1 = structure2;
```

where `structure1` and `structure2` are structures of the same type, assign the value of each member of `structure2` to the equivalent member of `structure1`.

When a function is passed a pointer to a structure, however, there is no need for a copy of the structure to be created within the function, or for values to be exchanged between equivalent structures in the function and the calling environment. A pointer gives a function access to the original structure in the calling environment, and allows it to use indirection to change the actual values that the structure contains.

2 Access to Structures Using Pointers

Once a structure template has been created to define a structure type, individual structures of that type may be declared and assigned values. Individual members of a structure may be accessed by linking the name of the member to the name of the structure data item using the `.` structure-member operator.

The following example creates a structure template called `Dictionary`, then declares and assigns values to a structure of this type called `program`, which contains information for a short dictionary entry on the subject of computer programs:

```
#include <string.h>

struct dictionary
{
     char word[20];
     char part[20];
     char defn[80];
};

struct dictionary program;

int main(void)
{
     strcpy(program.word, "DISK DRIVE");
     strcpy(program.part, "noun");
     strcpy(program.defn,
             "Sequence of instructions for a computer");

     return 0;
}
```

A pointer to a structure is declared to be a pointer to a structure of a particular type, and is assigned the address of a specific structure variable of that type. In the following example, `entry` is declared to be a pointer to a structure of type `Dictionary`, and it is assigned the address of a structure of that type called `ddrive`:

```
     struct dictionary * entry = &ddrive;
```

Note that whereas the name of an array is synonymous with its address, the name of a structure is not, so the address operator & is used in the above example to assign the address of a structure to a pointer.

3 The Structure-Pointer Operator

The structure called `ddrive` in the last example will contain information for a short dictionary entry on the subject of disk drives, and this information can be assigned using the pointer `entry` declared above:

```
strcpy(ddrive.word, "DISK DRIVE");
strcpy((*entry).part, "noun");
strcpy(entry->defn, "Device to read/write data");
```

Three different methods are used to assign information to the members of the structure `ddrive` in the above example. The first uses the method already demonstrated and connects the name of the structure to the name of the member by means of the structure-member operator `.`.

The second method also uses the structure-member operator, but this time substitutes an expression consisting of the name of a pointer to the structure and the contents of operator `*`, instead of the name of the structure variable. This allows access to the contents of the structure by indirection.

The third method is similar to the second, but uses the structure pointer operator `->` to link the name of the pointer to the structure member.

The structure-pointer operator is written using two characters: the dash `'-'`, followed by the greater than symbol `'>'`. Conceptually, it combines the operations of two operators, the structure-member operator `'.'` and the contents of operator `*`, and when linking a pointer to a structure with a structure member, may be thought of as referring to the contents of the specified member of the structure that is pointed to.

4 Passing Structure Pointers to Functions

The structure-pointer operator allows references to structure members by means of pointers to be expressed neatly and concisely. It also allows generalised functions to be written, which can act on structures, without any need for the names of the structures they act on to be known in advance.

The following function, for instance, prints out a brief explanation of a computer term when passed a pointer to a structure of type `Dictionary`, as defined in the examples `above:`

```
void print_out(const struct dictionary * item)
{
     printf("\n%s, (%s).", item->word, item->part);
     printf("\n%s", item->defn);
}
```

When passed a pointer to the structure `ddrive`, the function `print_out` will output:

```
DISK DRIVE, (noun).
Device to read/write data.
```

5 Returning Structure Pointers

A function may be declared to return a pointer to a structure of a given type.

The following function called `find` takes three arguments. The first is a string, which is the name of a topic that the user wants to look up in a dictionary of computer terms. The second argument is a pointer to the starting address of an array of structures of type `Dictionary`. The structures in this array contain the actual entries for the different computer terms in the dictionary. The third argument is the number of elements in the array, which is the total number of entries in the dictionary.

The function uses a `for` loop, pointer arithmetic and the expression `(point+count)` to move a pointer through the array of structures, pointing to each of the structures in turn. The structure pointer operator `->` is used with the pointer expression `(point+count)` to gain access to the `word` member of each structure, and the `strcmp` function compares the content of this member, which is the word defined in that structure, with the word that the user has requested information on. The `for` loop ends either when the end of the array is reached or when the sought-for topic is located. The function then returns a pointer to the structure containing a definition of the word that the user is looking up:

```
struct dictionary *      /* Return type */
find(char * string, struct dictionary * point, int total)
{
     int count;
     struct dictionary * end = (point+total); /* end of array */
     struct dictionary * answer = NULL;

     for (count = 0;
         ((point + count) < end) && (answer == NULL);
```

```
        count++)
    {
         if (strcmp((point+count)->word, string) == 0)
              answer = (point + count);
    }

    return answer;  /* pointer to structure containing definition, or NULL */
}
```

6 Structures Containing Pointers: Linked Lists and Binary Trees

A structure may contain members that are pointers, and this provides a powerful mechanism for expressing logical connections between data items, especially when structures contain pointers to other structures.

Linked lists may be created, in which each of a series of structures contains, in addition to other information, either a member that is a pointer to the next structure in the list, to give a singly-linked list, or two members that are pointers, one pointing to the previous structure and one to the next in the sequence. This second kind of series is known as a doubly-linked list.

Binary trees are also possible, which allow complex selection and decision-making processes to be speeded up by reducing the number of decisions to the minimum.

The binary tree in the example below contains seven `Dictionary` structures, each containing information on a different topic. The template for `Dictionary` structures would have been modified to include two pointers, each a pointer to another structure of type `Dictionary`:

```
struct dictionary
{
     char word[20];
     char part[20];
     char defn[80];
     struct dictionary * before;
     struct dictionary * after;
};
```

If the seven structures in this example were put in alphabetical order into an array of seven elements, or into a linked list, it could take up to a maximum of seven binary decisions to find a given topic, as each element would be examined in turn. In the tree arrangement, a maximum of only five decisions would be needed.

First, the `word` member in the structure `microprocessor` would be tested with the `strcmp` function to see whether the structure contained information on the required topic. If it did not, the next question would be:

Is the required topic before the word `microprocessor` in the dictionary?

This could be established by reference to the value returned by `strcmp`, which returns a negative value if its first argument comes before its second argument in the alphabet, and a positive value if it comes after it. If the searched-for topic were before `microprocessor`, the `floppy` structure would be examined next, and the same two tests applied. If the topic came after `microprocessor`, the `software` structure would be examined next.

The process would be repeated until the topic is eventually found.

If the search was for information on `utility`, the path through the tree would be as follows:

1) Is `microprocessor` the topic? (No)
2) Is `utility` before `microprocessor` in the alphabet? (No)
3) Is `software` the topic? (No)
4) Is `utility` before `software` in the alphabet? (No)
5) Is `utility` the topic? (Yes!)

7 Further Examples

There are other orders in which a binary tree may be searched, and many other forms of linked data structure are possible. A full discussion of these topics is beyond the scope of this summary.

The following, however, is a program for a small computer dictionary based on the binary tree described in Section 6 above. Note that the pointers `before` and `after` are not assigned values until all the structures in the dictionary have been created. A potentially dangerous error occurs if a pointer is assigned the address of an object that has not yet been created, and has therefore not been allocated an actual address in memory.

Note also that the user's input string is converted to upper-case characters by a call to the function `capitalise` before it is compared with the `word` member in any of the structures. This ensures that the input string and the searched-for string are both in upper-case characters, whatever combination of lower-case and upper-case characters was entered by the user. A cast is used in `capitalise` to convert the value returned by `toupper` from type `int` to type `char`.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

struct dictionary
{
     char word[20];
     char part[20];
     char defn[80];
     struct dictionary * before;
     struct dictionary * after;
};

struct dictionary ddrive =
{
     "DISK DRIVE",
     "noun phrase",
     "Device to read/write data",
     NULL,
     NULL
};

struct dictionary floppy =
{
     "FLOPPY DISK",
     "noun phrase",
     "Portable disk for storing computer programs/files.",
     NULL,
     NULL
};

struct dictionary hard =
{
     "HARD DISK",
     "noun phrase",
     "Fixed, large capacity storage disk for computer data.",
     NULL,
     NULL
```

```
};

struct dictionary micro =
{
     "MICROPROCESSOR",
     "noun",
     "Instruction processing chip in a computer.",
     NULL,
     NULL
};

struct dictionary program =
{
     "PROGRAM",
     "noun",
     "Sequence of instructions to a computer.",
     NULL,
     NULL
};

struct dictionary software =
{
     "SOFTWARE",
     "noun",
     "Generic term for computer programs.",
     NULL,
     NULL
};

struct dictionary utility =
{
     "UTILITY",
     "noun",
     "Useful program for handling files, disks or devices.",
     NULL,
     NULL
};
```

```
struct dictionary * search(char *, struct dictionary *);
void capitalise(char *);
void print_out(struct dictionary *);

int main(void)
{
     struct dictionary * pointer;

     micro.before    = &floppy;
     micro.after     = &software;
     floppy.before   = &ddrive;
     floppy.after    = &hard;
     software.before = &program;
     software.after  = &utility;

     pointer = &micro;

     do
     {
          char response[30], wanted[30];

          puts("Enter name of topic you are interested in");
          putchar('\n');
          puts("or type 'quit' to exit from program.");

          scanf("%29s", response);
          strcpy(wanted, response);
          capitalise(wanted);
          if (strcmp(wanted, "QUIT") != 0)
          {
               struct dictionary * result
                              = search(wanted, pointer);

               if (result == NULL)
                    printf("\nSorry, no information "
                           "available on %s.", wanted);
               else
                    print_out(result);
```

```
          }
     }
while (strcmp(wanted, "QUIT") != 0)
     ; /* all done! */

return 0;
}

struct dictionary *
       search(char * string, struct Dictionary * point)
{
     int alphatest = strcmp(string, point->word);

     if (alphatest == 0)
          return point;
     else if (alphatest < 0)
          return search(string, point->before);
     else
          return search(string, point->after);
}

void print_out(const struct dictionary * item)
{
     puts("\n\n****************************************");
     printf("\n\n%s, (%s).", item-> word, item->part);
     printf("\n%s", item->defn);
     puts("\n\n****************************************");
}

void capitalise(char * subject)
{
     int count;
     for(count = 0; *(subject + count) != '\0'; count++)
     {
          *(subject + count) =
               (char)toupper(*(subject + count));
     }
}
```