Introduction to Structures

Structure is a user-defined *compound* data type in C. It allows you to combine different data types to store a particular type of record.

Structure is used to represent a record. Suppose you want to store record of student which consists of name, address, roll number and marks (in five subjects). You can define a structure to hold this information.

Defining a Structure

struct keyword is used to define a structure. The declaration syntax of a structure is:

```
struct structure_name {
  data_type1 member1;
  data_type2 member2;
  data_type3 member3;
  data_typeN memberN;
};
```

The actual declaration for the student record described above would be,

```
struct student_t {
  char name[40];
  char address[100];
  char rollnum[12];
  int marks[5];
};
```

Let us look at another example:

```
struct book_t {
  char title[25];
  char author[40];
  float price;
  int pages;
};
```

Here the struct book_t declares a structure to hold the details of book which consists of four data fields, namely the title of the book, the name of the author, the price and the number of pages. These fields are called structure elements or *members*. Each member can have different data type, like in this case, where title is of char [] type and price is of **float** type and pages is of **int** type. `book_t` is the name of the structure and is called *structure tag*. It is a convention to append _t to structure names. It is not mandatory.

Declaring Variables

Structure variable declaration is similar to the declaration of variables of any other data types. For example we can define a variable `mathsBook` as struct book_t mathsBook;

You can also declare structure variables along with the definition of the structure; but it should be avoided. That would be something like,

```
struct book_t {
  char title[25];
  char author[40];
  float price;
  int pages;
} mathsBook;
```

Accessing Members

Structure members can be accessed and assigned values in number of ways. Structure member has no meaning independently. In order to assign a value to a structure member, the member name must be linked with the structure variable using dot (.) operator aka member access operator. `mathsBook.price`

Continuing with the mathsBook, the way to set a price for it will be,

`mathsBook.price = 200;`

Structure members are like any other variable, except for the notational difference of having a dot in them. So you can display them using `printf()`, get their value using `scanf()` and so on.

```
scanf(‘‘%s’’, mathsBook.title);
scanf(‘‘%f’’, &mathsBook.price);
```

Initialization

Like any other data type, structure variable can also be initialized.

```
struct book_t mathBook = {
    ''Advanced Calculus'', ''Apostol'', 816.00, 284};
```

Structure as parameter

A structure can be passed to any function as a parameter. It is passed by value. Recall that this means that any changes made to the values of the members of that structure are not visible to the called function.

Similarly a structure may be returned from a function as a return value.

Example

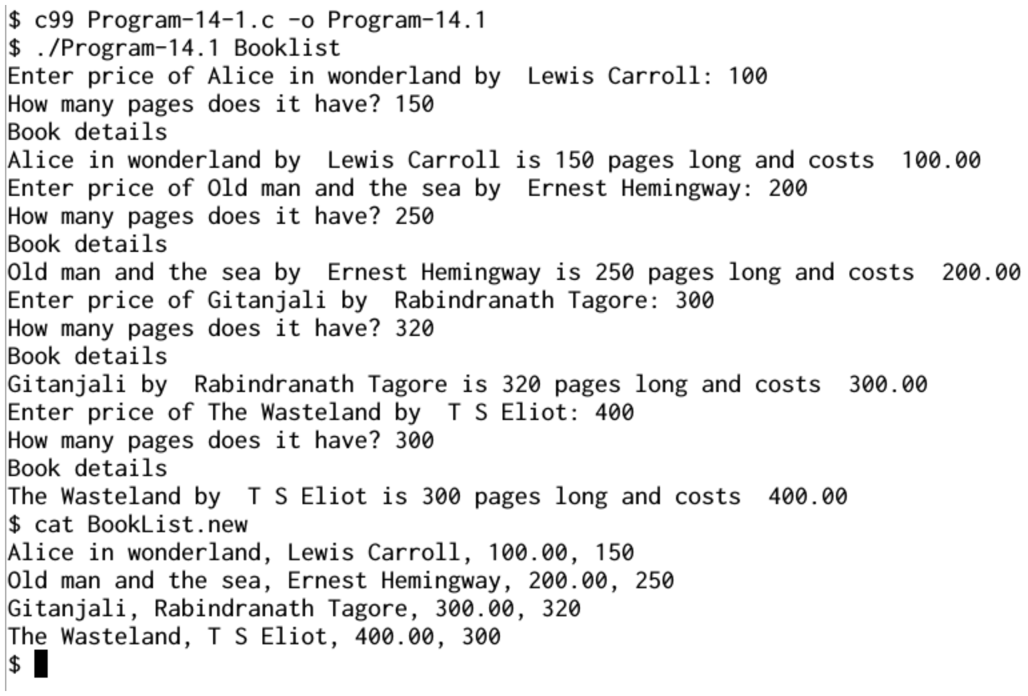
Let us say we have a file which contains details of some books: one per line comma separated list of titles and authors. We want to read it, display the data, accept the price and number of pages and write the file back.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct book_t {
    char title[25];
    char author[40];
    float price;
    int pages;
};

void display(struct book_t);
void writenewlist(struct book_t, FILE*);

int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s Booklist\n", argv[0]);
        exit(0);
```

```c
    }

    FILE* inp = fopen(argv[1], "r");
    FILE* out = fopen("BookList.new", "w");
    struct book_t aBook;

    char line[1024];
    while (fgets(line, 1024, inp) != NULL) {
        char buffer[40];
        char* p = line;
        char* q = buffer;

        // -------------------------- *
        // Copy title to buffer *
        // -------------------------- *
        while (*p != ',') {
            *q = *p;
            q++;
            p++;
        }
        *q = '\0'; //Terminate the title string
        strcpy(aBook.title, buffer);
        p++; // skip the comma

        // ---------------------------- *
        // Copy author to buffer *
        // ---------------------------- *
        q = buffer;
        while (*p != '\n') {
            *q = *p;
            q++;
            p++;
        }
        *q = '\0'; //Terminate the author name
        strcpy(aBook.author, buffer);

        printf("Enter price of %s by %s: ",
           aBook.title, aBook.author);
        scanf("%f", &aBook.price);
        printf("How many pages does it have? ");
        scanf("%d", &aBook.pages);

        display(aBook);
        writenewlist(aBook, out);
    }
    return 0;
}

void display(struct book_t book) {
    printf("Book details\n");
    printf("%s by %s is %d pages long and costs %7.2f\n",
         book.title, book.author, book.pages, book.price);
    return;
}

void writenewlist(struct book_t book, FILE* f) {
    fprintf(f, "%s,%s,%7.2f,%4d\n", book.title,
          book.author, book.price, book.pages);
    return;
}
```

Unions

A union is a special data type available in C that enables you to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Think of these members as synonyms for the same memory location; but with different syntax and semantics depending on their data type.



```
$ c99 Program-14-1.c -o Program-14.1
$ ./Program-14.1 Booklist
Enter price of Alice in wonderland by  Lewis Carroll: 100
How many pages does it have? 150
Book details
Alice in wonderland by  Lewis Carroll is 150 pages long and costs  100.00
Enter price of Old man and the sea by  Ernest Hemingway: 200
How many pages does it have? 250
Book details
Old man and the sea by  Ernest Hemingway is 250 pages long and costs  200.00
Enter price of Gitanjali by  Rabindranath Tagore: 300
How many pages does it have? 320
Book details
Gitanjali by  Rabindranath Tagore is 320 pages long and costs  300.00
Enter price of The Wasteland by  T S Eliot: 400
How many pages does it have? 300
Book details
The Wasteland by  T S Eliot is 300 pages long and costs  400.00
$ cat BookList.new
Alice in wonderland, Lewis Carroll, 100.00, 150
Old man and the sea, Ernest Hemingway, 200.00, 250
Gitanjali, Rabindranath Tagore, 300.00, 320
The Wasteland, T S Eliot, 400.00, 300
$ █
```

Figure 1: Using structures

Defining a Union

To define a union, you must use the union statement. The union statement defines a new data type, with more than one member for your program.

The format of defining union is as follows:

```
union <union_name> {
     member definition;
     member definition;
            ...
     member definition;
} [one or more union variables];
```

Here is the way you would define a union named *data* which has the three members i, f, and str of int, float and char array.

```
union data_t {
  int i;
  float f;
```

```
    char str[20];
  } data;
```

Now, a variable of data_t type can store an integer, OR a floating-point number, OR a string of characters. This means that the same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

To access any member of a union, we use the member access operator (.), similar to structures.

`union_variable.union_member`

Union vs Structure

The differences between structure and union are:

- union allocates the memory equal to the memory required by the largest member of the union but structure allocates the memory equal to the total memory required by all the members.
- In union, one memory block is used by all the member of the union but in case of structure, each member have their own memory space.
- structure enables us to treat a collection of number of data items as a single composite entity, a union enables us to look at the same space in memory in one of many different ways.

Arrays of Structure

Structure is used to store the information of one particular object but if we need to store 100 such objects then Array of Structure is used. Since structures are like any other data type it is easy to understand the declaration, definition and usage.

```
  struct book_t {
    char title[25];
    char author[40];
    float price;
    int pages;
  } library[100];
```

OR

```
  struct book_t {
    char title[25];
    char author[40];
    float price;
    int pages;
  };
  ....
 struct book_t library[100];
```

Both are equivalent methods of declaring an array of 100 books called a library. Accessing an individual book in an array is by the usual array access mechanism: `library[n]` refers to the $(n+1)^{th}$ element of the array. Accessing a member of a particular element is also as expected: `library[n].title`

Pointers to Structures

Like any other data item a structure variable also has an address and can be manipulated using pointers.

For example, continuing with the book, struct book_t* pBook; defines a pointer to a structure. The way to initialize would be to take the address of a book_t, for example: `pBook = &mathBook;`

Referring to a member becomes a little tricky. `*pBook.title` will not work as the precedence of dot is higher; so you have to write `(*pBook).title`. There is an alternative notation available, namely `pBook->title`, and this definitely is better.