MANGALAYATAN UNIVERSITY
Learn Today to Lead Tomorrow
NAAC GRADE A+ Accredited University

BACHELOR OF COMPUTER APPLICATION (BCA)

CSB-1112 (Problem Solving Using C)

Block 2: Arrays and Strings

Unit 8: Structures and File Handling

8.0 Introduction

In C programming, managing complex data and storing it efficiently are essential aspects of developing robust applications. Two fundamental concepts that facilitate these tasks are structures and file handling.

Structures in C allow the grouping of related variables under a single name, providing a convenient way to manage and organize complex data. Each variable in a structure can be of a different data type, making structures a versatile tool for modeling real-world entities. For instance, a structure can represent a student with fields for the student's name, age, and grades. By using structures, programmers can create more readable and maintainable code, making it easier to manage data that logically belongs together.

A union is a user-defined data type, similar to a structure, but with the key difference that all members share the same memory location. The size of a union is determined by the size of its largest member. Unions are particularly useful when you need to work with different data types but want to conserve memory.

File handling in C enables the storage and retrieval of data to and from files, allowing for persistent data storage beyond the program's runtime. This functionality is crucial for applications that need to save user data, configuration settings, or any form of long-term information. The C Standard Library provides a comprehensive set of functions for file operations, such as opening, reading, writing, and closing files. These functions ensure efficient data management and manipulation, enabling programs to interact with files seamlessly.

This unit will be discussing the user-defined data type Structures and Unions of language C.

8.1 Objectives

After completing this unit, you will be able to:

- Declare and initialize the members of the structures;
- Access the members of the structures;
- Pass the structures as function arguments;
- Declare the array of structures;
- Declare and define union; and
- Perform all operations on the variables of type union.

8.2 Declaration of Structures

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly, structure is another user-defined data type available in C that allows to combine data items of different kinds.

To declare a structure you must start with the keyword struct followed by the structure name or structure tag and within the braces the list of the structure's member variables. Note that the structure declaration does not actually create any variables.

The syntax for the structure declaration is:

struct structure-tag

{

datatype variable1;

datatype variable2;

dataype variable 3;

...

};

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

Example:

struct Books

{

```
char title[50];
char author[50];
char subject[100];
int book_id; } book;
```

This defines a structure which can be referred to either as struct books or Books, whichever you prefer. Strictly speaking, you don't need a tag name both before and after the braces if you are not going to use one or the other. But it is a standard practice to put them both in and to give them the same name, but the one after the braces starts with an uppercase letter. The typedef statement doesn't occupy storage: it simply defines a new type. Variables that are declared with the typedef above will be of type struct book.

8.3 Accessing the Members of A Structure

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword struct to define variables of structure type.

The following syntax shows how to use a structure:

structurevariable. member-name;

```
struct coordinate
{
int x;
int y;
};
```

Thus, to have the structure named first refer to a screen location that has coordinates x=50, y=100, write as:

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure second, to write,

printf ("%d,%d", second.x, second.y);

The individual members of the structure behave like ordinary date elements and can be accessed accordingly.

Example:

```
#include<stdio.h>
#include<string.h>
struct Books
{
char title[50];
char author[50];
char subject[100];
```

```
int book_id;
};
int main( )
{
struct Books Book1;              /* Declare Book1 of type Book */
struct Books Book2;              /* Declare Book2 of type Book */
 /* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;
/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);
 /* print Book2 info */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);
return 0;
}
```

The following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
```

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book_id : 6495700

8.4 Initializing Structures

C variable types, structures can be initialized when they're declared. This procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values is separated by commas and enclosed in braces.

Example:

```
struct customer
{
char firm[20];
char contact[25];
}
struct sale
{
struct customer buyer1;
char item [20];
float amt;
}
mysale = {
{ "XYZ Industries", "Tyran Adams"};
"toolskit";
600.00
};
```

These statements perform the following initializations:

- The structure member mysale.buyer1.firm is initialized to the string "XYZ Industries".
- The structure member mysale.buyer1.contact is initialized to the string "Tyran Adams".
- The structure member mysale.item is initialized to the string "toolskit".
- The structure member mysale.amount is initialized to the amount 600.00.

8.5 Structures as Function Arguments

C is a structured programming language and the basic concept in it is the modularity of the programs. This concept is supported by the functions in C language. Let us look into the techniques of passing the structures to the functions. This can be achieved in primarily two ways: Firstly, to pass them as simple parameter values by passing the structure name and secondly, through pointers. We will be concentrating on the first method in this unit and passing using pointers will be taken up in the next unit. Like other data

types, a structure can be passed as an argument to a function. The program listing given below shows how to do this. It uses a function to display data on the screen.

Example:

```
#include<stdio.h>
#include<string.h>
struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id; };
/* function declaration */
void printBook( struct Books book );
int main( )
{
struct Books Book1;                    /* Declare Book1 of type Book */
struct Books Book2;                    /* Declare Book2 of type Book */
/* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;
/* print Book1 info */
printBook( Book1 );
/* Print Book2 info */
printBook( Book2 );
return 0;
}
void printBook( struct Books book );
```

```
{
printf( "Book title : %s\n", book.title);
printf( "Book author : %s\n", book.author);
printf( "Book subject : %s\n", book.subject);
printf( "Book book_id : %d\n", book.book_id);
}
```

The following result:

```
Book title : C Programming
Book author : Nuha Ali Book
subject : C Programming Tutorial Book
book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

8.6 Pointers to Structures

To define pointers to structures in the same way as you define pointer to any other variable:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above-defined pointer variable. To find the address of a structure variable, place the '&' operator before the structure's name as:

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, to operator as:

```
struct_pointer -> title;
```

Let us rewrite the example using structure pointer.

```
#include<stdio.h>
#include<string.h>
struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
};
/* function declaration */
```

```
void printBook( struct Books *book );
int main( )
{
struct Books Book1;        /* Declare Book1 of type Book */
struct Books Book2;        /* Declare Book2 of type Book */
/* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;
/* print Book1 info by passing address of Book1 */
printBook( &Book1 );
/* print Book2 info by passing address of Book2 */
printBook( &Book2 );
return 0;
}
void printBook( struct Books *book )
{
printf( "Book title : %s\n", book->title);
printf( "Book author : %s\n", book->author);
printf( "Book subject : %s\n", book->subject);
printf( "Book book_id : %d\n", book->book_id);
}
```

The following result:

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
```

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book_id : 6495700

8.7 Unions

A union is a special data type available in C that allows storing 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. Unions provide an efficient way of using the same memory location for multiple purposes.

Example:

In case of the support price of shares you require only the latest quotations. And only the ones that have changed need to be stored. So if we declare a structure for all the scripts, it will only lead to crowding of the memory space. Hence it is beneficial if we allocate space to only one of the members. This is achieved with the concepts of the UNIONS.

UNIONS are similar to STRUCTURES in all respects but differ in the concept of storage space. A UNION is declared and used in the same way as the structures.

A union can be declared using the syntax:

```
union union-tag {
datatype variable1;
 datatype variable2;
 ……..
};
```

Example:

```
union temp{
int x;
char y;
float z;
};
```

In this case a float is the member which requires the largest space to store its value hence the space required for float (4 bytes) is allocated to the union.

8.8 Initializing an Union

Initializing a union in programming involves declaring a union variable and assigning initial values to its members. A union is a data structure that allows storing different data types in the same memory location.

Example:

```
union date_tag {
char complete_date [9];
struct part_date_tag {
```

char month[2];

char break_value1;

char day[2];

char break_value2;

char year[2];

}

parrt_date;

}

date = {“01/01/05”};

Example:

```c
#include <stdio.h>

union MyUnion {
    int intValue;
    float floatValue;
    char stringValue[20];
};

int main() {
    union MyUnion data;

    // Initializing the union members
    data.intValue = 10;
    // Accessing the intValue member
    printf("Integer value: %d\n", data.intValue);

    data.floatValue = 3.14;
    // Accessing the floatValue member
    printf("Float value: %f\n", data.floatValue);

    // Initializing the stringValue member
    strcpy(data.stringValue, "Hello, Union!");
    // Accessing the stringValue member
    printf("String value: %s\n", data.stringValue);

    return 0;
}
```

8.9 Accessing the Members of an Union

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type.

The following example to use unions in a program:

```
#include<stdio.h>
#include<string.h>
union Data
{
int i;
float f;
char str[20];
};
int main( )
{
union Data data;
data.i = 10;
data.f = 220.5;
strcpy( data.str, "C Programming");
printf( "data.i : %d\n", data.i);
printf( "data.f : %f\n", data.f);
printf( "data.str : %s\n", data.str);
return 0;
}
```

The following result:

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, the values of i and f members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

Now, use one variable at a time which is the main purpose of having unions:

```
#include<stdio.h>
#include<string.h>
union Data
{
int i;
float f;
char str[20];
};
```

```
int main( )
{
union Data data;
data.i = 10;
printf( "data.i : %d\n", data.i);
data.f = 220.5;
printf( "data.f : %f\n", data.f);
strcpy( data.str, "C Programming");
printf( "data.str : %s\n", data.str);
return 0;
}
```

The following result:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

8.10 File Handling

File handling in C involves performing operations such as creating, opening, reading, writing, and closing files. C provides a set of standard library functions to manage files, enabling efficient data storage and retrieval. These operations are crucial for persistent storage of data beyond the program's execution lifecycle.

Basic File Operations

1. **Opening a File**
2. **Reading from a File**
3. **Writing to a File**
4. **Closing a File**

Opening a File

To perform any operation on a file, it must first be opened using the **fopen** function. This function returns a pointer to a **FILE** object that represents the file.

Syntax:

FILE *fopen(const char *filename, const char *mode);

Parameters:

- filename: Name of the file to open.
- mode: Mode in which to open the file (e.g., "r", "w", "a", "r+", "w+", "a+").

Example:

```
FILE *file = fopen("example.txt", "r");
```

```
 if (file == NULL)
{
 printf("Could not open file.\n");
return 1;
}
```

Reading from a File

To read data from a file, functions like **fgetc**, **fgets**, and **fread** are used.

- **fgetc:** Reads a single character from a file.
- **fgets:** Reads a string from a file.
- **fread:** Reads a block of data from a file.

Example:

```
char ch;
while ((ch = fgetc(file)) != EOF) {
   putchar(ch);  // Print each character to the console
}
```

Writing to a File

To write data to a file, functions like **fputc**, **fputs**, and **fwrite** are used.

- **fputc:** Writes a single character to a file.
- **fputs:** Writes a string to a file.
- **fwrite:** Writes a block of data to a file.

Example:

```
FILE *file = fopen("example.txt", "w");
if (file == NULL) {
   printf("Could not open file for writing.\n");
   return 1;
}
const char *text = "Hello, World!";
fputs(text, file);
```

Closing a File

After performing the necessary operations, the file should be closed using the **fclose** function to free resources.

Syntax:

```
int fclose(FILE *stream);
```

Example:

```
fclose(file);
```

Additional File Handling Functions

- **fprintf and fscanf:** These functions are similar to **printf** and **scanf** but work with files.
- **fseek:** Sets the file position to a specific location.
- **ftell:** Returns the current file position.
- **rewind:** Sets the file position to the beginning of the file.

Example Using fprintf and fscanf:

```
FILE *file = fopen("example.txt", "w+");
if (file == NULL) {
   printf("Could not open file.\n");
   return 1;
}
fprintf(file, "Name: %s, Age: %d\n", "John Doe", 30);
rewind(file);  // Move file pointer back to the beginning
char name[50];
int age;
fscanf(file, "Name: %s, Age: %d", name, &age);
printf("Read from file: Name = %s, Age = %d\n", name, age);
fclose(file);
```

Example Program

Here's a complete example that demonstrates opening a file, writing to it, reading from it, and closing it.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   // Open a file for writing
   FILE *file = fopen("example.txt", "w");
   if (file == NULL)
{
      printf("Could not open file for writing.\n");
      return 1;
   }
   // Write to the file
   const char *text = "Hello, World!";
   fputs(text, file);
   // Close the file
```

```
    fclose(file);
    // Open the file for reading
    file = fopen("example.txt", "r");
    if (file == NULL)
    {
        printf("Could not open file for reading.\n");
        return 1;
    }
    // Read from the file and print to console
    char ch;
    while ((ch = fgetc(file)) != EOF) {
        putchar(ch);
    }
    // Close the file
    fclose(file);
    return 0;
}
```

8.11 Summary

The combination of structures, unions, and file handling in C provides a comprehensive toolkit for managing complex data and interacting with external files efficiently. Structures allow for the organization of related data fields into a single entity, facilitating the creation of more readable and maintainable code. Unions, on the other hand, offer a memory-efficient way to store different data types in the same memory location, enabling versatile data management in memory-constrained environments.

Structures and unions can be seamlessly integrated with file handling operations, such as opening, reading, writing, and closing files. This integration allows C programs to interact with external files for persistent storage and retrieval of data. By using structures to organize data and unions to conserve memory, programmers can efficiently manage complex data structures and interact with files to store and retrieve data as needed.

In summary, structures, unions, and file handling in C complement each other to provide powerful capabilities for data management and storage. Understanding how to leverage these features effectively enables developers to create robust and efficient applications that can handle complex data structures and interact with external files seamlessly.

8.12 Questions

1. **Discuss the potential pitfalls of using unions in C programming. How can these pitfalls be avoided or mitigated?**

Ans.1 One potential pitfall of using unions is the risk of accessing the wrong member, leading to undefined behavior or data corruption. This can be mitigated by ensuring proper initialization and

careful handling of union members. Another concern is the alignment and padding of union members, which can vary depending on the compiler and architecture. To address this, developers can use compiler-specific directives or pragma statements to control padding and alignment.

2. **Explain the concept of a tagged union in C. How can tagged unions be useful in certain programming scenarios?**

Ans.2 A tagged union, also known as a discriminated union, is a union that includes an additional member to indicate which type of data is currently stored in the union. This allows for safe access to union members by providing runtime type information. Tagged unions are useful in scenarios where different types of data need to be stored and manipulated, such as in interpreters, parsers, and protocol implementations. By tagging each union member with a type identifier, the code can safely handle different data types and perform appropriate operations based on the current type.

3. **Describe the steps involved in reading data from a file in C. Discuss error handling strategies when reading files.**

Ans.3 The steps for reading data from a file in C typically include:

- Open the file using fopen.
- Read data from the file using functions like fscanf, fgetc, or fgets.
- Process the read data.
- Close the file using fclose. Error handling strategies may involve checking the return value of file opening functions to ensure the file was opened successfully, checking for end-of-file conditions during reading operations, and handling errors gracefully using conditional statements or error-handling functions.

4. **Explain the difference between text and binary file modes in file handling. When would you choose one over the other?**

Ans.4 Text file mode ("t") is used for reading and writing files in text format, where data is stored as human-readable characters. Binary file mode ("b") is used for reading and writing files in binary format, where data is stored as a sequence of bytes without any formatting. Text mode is suitable for working with text files, such as plain text documents or configuration files, while binary mode is preferred for working with non-text files, such as images, audio files, or executable programs.

5. **Discuss the advantages of using structures in C programming. Provide examples where structures can be used effectively.**

Ans.5 Structures offer several advantages in C programming:

- They allow for the organization of related data fields into a single entity, improving code readability and maintainability.
- They enable the creation of complex data structures, such as linked lists, trees, and graphs, by combining multiple variables into one.
- They facilitate the passing of multiple arguments to functions as a single parameter, reducing the complexity of function prototypes. For example, a function to calculate the area of a rectangle can take a structure representing the rectangle as its parameter.

6. **Explain the concept of nested structures in C. Provide an example to illustrate nested structures.**

Ans.6 Nested structures refer to structures that are members of other structures. For example:

```
struct Address {
```

```
    char city[50];
    char state[50];
    int zipCode;
};
struct Person {
    char name[50];
    int age;
    struct Address address;
};
```

In this example, the Person structure contains a member address, which is itself a structure of type Address. This allows for hierarchical organization of data, with the Person structure encapsulating both personal information and address details.

8.13 References

- Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.
- King, K. N. (2008). C Programming: A Modern Approach (2nd ed.). W. W. Norton & Company.
- Prata, S. (2014). C Primer Plus (6th ed.). Addison-Wesley Professional.
- Deitel, P., & Deitel, H. (2016). C How to Program (8th ed.). Pearson.

***