

Module-5

Structure, Union and Files

- **Structure :-** Structure is a collection of one or more variables of same or different data types grouped to gather under a single name for easy handling.
- **Structure is a user defined** data type that can store related information about an object.
- **Declaration of a Structure :-** A structure is declared using the keyword **struct** followed by **structure name and variables** are declared within a structure

Declaration of a Structure

- A structure is declared using the keyword **struct** followed by **structure name** and **variables** are declared within a structure

- **The structure is usually declared before the `main()` function.**

- Syntax : -

```
struct structure_name
{
    datatype member 1;
    datatype member 2;
    datatype member 3;
    .....
    .....
    datatype member n;
};
```

- Example :-

```
struct employee
```

```
{
```

```
    int emp_no;
```

```
    char name[20];
```

```
    int age;
```

```
    float emp_sal;
```

```
};
```

Initialization of structure

A structure initialization is done after the declaration of the structure

```
struct employee {
```

```
int emp_no;
```

```
char name[20];
```

```
int age;
```

```
float emp_sal;
```

```
} ;
```

```
emp1={65421, “Hari”,29,25000.00};
```

The order of values enclosed in the braces must match the order of members in the structure definition

Accessing structure members

- A structure uses a .(dot) [Member Access Operator] to access any member of the structure.
- `emp1.emp_no = 65421;`

Accessing structure members

The order of values enclosed in the braces must match the order of members in the structure definition

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

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

```
struct employee
```

```
{
```

```
    int emp_no;
```

```
    char empname[20];
```

```
    int age;
```

```
    float emp_sal;
```

```
};
```



```
void main()
{
    struct employee emp1;
    emp1.emp_no = 65421;
    emp1.empname="Hari";
    emp1.age=29;
    emp1.emp_sal=25000.00;
    printf("Employee Number=%d\n",emp1.emp_no);
    printf("Employee Name=%s\n",emp1.empname);
    printf("Employee Age=%d\n",emp1.age);
    printf("Employee Salary=%f\n",emp1.emp_sal);
}
```

```
#include<stdio.h>
#include<string.h>
struct employee
{
    int emp_no;
    char empname[20];
    int age;
    float emp_sal;
};
```

```
void main()
{
    struct employee emp1;
    printf("Enter Employee Number:");
    scanf("%d",&emp1.emp_no);
    printf("Enter Employee Name:");
    scanf("%s",emp1.empname);
    printf("Enter Employee age:");
    scanf("%d",&emp1.age);
    printf("Enter Employee salary:");
    scanf("%f",&emp1.emp_sal);
    printf("Employee Number=%d\n",emp1.emp_no);
    printf("Employee Name=%s\n",emp1.empname);
    printf("Employee Age=%d\n",emp1.age);
    printf("Employee Salary=%f\n",emp1.emp_sal);
}
```

Passing Structures Through Pointers

Structure pointer is defined as the [pointer](#) which points to the address of the memory block that stores a [structure](#) known as the structure pointer.

```
#include <stdio.h>
#include <string.h>
struct Student {
```

```
    int roll_no;
    char name[30];
    char branch[40];
    int batch;
```

```
};
```

```
int main()
```

```
{
```

```
    struct Student s1;
    struct Student* ptr = &s1;
    s1.roll_no = 27;
    strcpy(s1.name, "Kamlesh Joshi");
    strcpy(s1.branch, "Computer Science And Engineering");
    s1.batch = 2019;
    printf("Roll Number: %d\n", (*ptr).roll_no);
    printf("Name: %s\n", (*ptr).name);
    printf("Branch: %s\n", (*ptr).branch);
    printf("Batch: %d", (*ptr).batch);
    return 0;
```

```
}
```

Roll Number: 27

Name: Kamlesh Joshi

Branch: Computer Science And Engineering

Batch: 2019

Array of Structures

- An array within a structure is a member of the structure and can be accessed just as we access other elements of the structure.
- If we want to store the data of 100 employees we would require 100 structure variables from emp1 to emp100 which is definitely impractical the better approach would be to use the array of structures.

```
/* C program to illustrate array of structures */  
#include<stdio.h>  
#include<string.h>  
struct employee  
{  
    int emp_no;  
    char empname[20];  
    int age;  
    float emp_sal;  
};
```

```
void main()
```

```
{   struct employee emp[20];
```

```
    int n,i;
```

```
    printf("Enter the number of employee entries\n");
```

```
    scanf("%d",&n);
```

```
    for(i=0;i<n;i++)
```

```
    {   printf("Enter the details of employee %d\n",i+1);
```

```
        printf("Enter Employee Number:");
```

```
        scanf("%d",&emp[i].emp_no);
```

```
        printf("Enter Employee Name:");
```

```
        scanf("%s",emp[i].empname);
```

```
        printf("Enter Employee age:");
```

```
        scanf("%d",&emp[i].age);
```

```
        printf("Enter Employee salary:");
```

```
        scanf("%f",&emp[i].emp_sal);
```

```
    }
```



```
printf("\nEMP_NO\t EMP_NAME\t EMP_AGE\t\t  
EMP_SALARY \n");  
for(i=0;i<n;i++)  
{  
printf("%d\t%s\t%d\t%f\n",emp[i].emp_no,emp[i].empn  
ame,emp[i].age,emp[i].emp_sal);  
}  
}
```

- Nested Structures :- A nested structure is a structure that contains another structure as its member.

Syntax :-

```
struct structure_name1
```

```
{
```

```
    datatype member 1;
```

```
    datatype member 2;
```

```
};
```

```
struct structure_name2
```

```
{
```

```
    datatype member 1;
```

```
    datatype member 2;
```

```
    struct structure_name1 var1;
```

```
};
```

```
/* C program to demonstrate nested structures */
```

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

```
struct stud_dob
```

```
{
```

```
    int day;
```

```
    int month;
```

```
    int year;
```

```
};
```

```
struct student
```

```
{
```

```
    int rollno;
```

```
    char sname[20];
```

```
    struct stud_dob date;
```

```
};
```

```
void main()
{
    struct student s;
    printf("Enter Student Rollno :");
    scanf("%d",&s.rollno);
    printf("Enter Student Name :");
    scanf("%s",s.sname);
    printf("Enter date of birth as day month year:");
    scanf("%d%d%d",&s.date.day,&s.date.month,&s.date.year);

    printf("Student Details are\n");
    printf("Student Rollno = %d\n",s.rollno);
    printf("Student Name=%s",s.sname);
    printf("Student DOB= %d-%d-%d\n", s.date.day, s.date.month,
        s.date.year);
}
```

Unions

- A union is **a special data type available in C that allows to store different data types in the same memory location.**
- Union can be defined with many members, but only one member can contain a value at any given time.
- Syntax:

```
union union_name  
{  
    datatype field_name;  
    datatype field_name; // more variables  
}union_variable;
```

Accessing Union Members

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

```
union Job {
```

```
float salary;
```

```
int workerNo;
```

```
} j;
```

```
void main()
```

```
{ j.salary = 12.3;
```

```
j.workerNo = 100; // when j.workerNo is assigned a value, j.salary will no  
longer hold 12.3 size of both the data type is 4byte
```

```
printf("Salary = %f\n", j.salary);
```

```
printf("Number of workers = %d", j.workerNo);
```

```
}
```

Output

Salary = 0.0

Number of workers = 100

Difference between structure and union

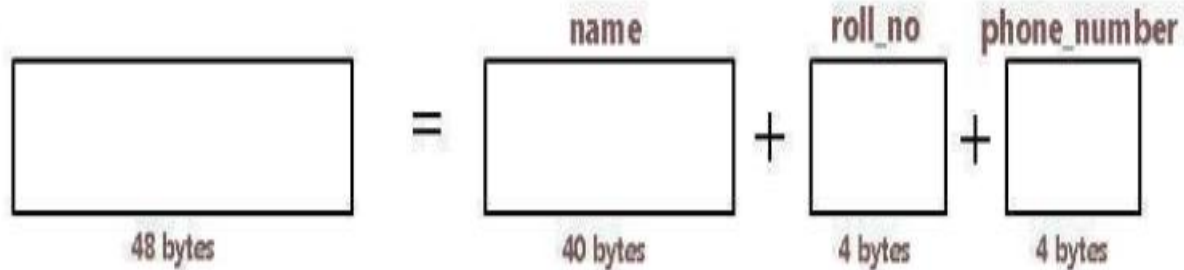
```
#include <stdio.h>
struct Job {
float salary;
int workerNo;
} j;
int main()
{
j.salary = 12.3;
j.workerNo = 100;
printf("Salary = %f\n", j.salary);
printf("Number of workers = %d", j.workerNo);
return 0;
}
```

Salary = 12.300000
Number of workers = 100

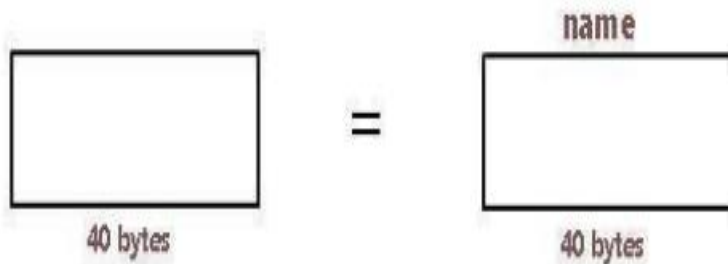
Salary = 0.000000
Number of workers = 100

```
#include <stdio.h>
union Job {
float salary;
int workerNo;
} j;
int main()
{
j.salary = 12.3;
j.workerNo = 100;
printf("Salary = %f\n", j.salary);
printf("Number of workers = %d", j.workerNo);
return 0;
}
```

Difference between structure and union

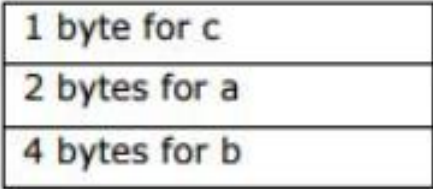
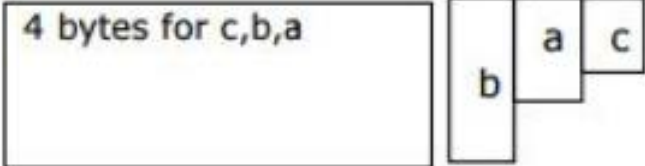


Memory allocated to structure



Memory allocated to union

Unions

Structure	Union
Structure is a collection of one or more variables of same or different data types grouped to gather under a single name for easy handling.	A union is a special data type available in C that allows to store different data types in the same memory location.
All members are active at a time.	Only one member is active a time.
All members can be initialized.	Only the first member can be initialized.
Requires more memory.	Requires less memory.
Example: <pre>struct SS { int a; float b; char c; };</pre> 	Example: <pre>union UU { int a; float b; char c; };</pre> 
Total bytes = 1 + 2 + 4 = 7 bytes. 622 x 426	4 bytes are there between a,b and c because largest memory occupies by float which is 4 bytes.

Arrays of Union Variables

We can create array of unions similar to creating array of any primitive data type.

general form of declaration for array of union.

```
union <union_name> <array_name>[size];
```

Consider the following example,

```
union values {
```

```
int int_val;
```

```
float float_val;
```

```
};
```

```
union values arr[2];
```

Here, *arr* is an array of union which can hold two union elements.

Arrays of Union Variables

Array of union initialization:

union values arr[2] = {{1}, {2}}; The above statement initializes first union member of both the array elements.

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

```
union values {
```

```
int integer_val; // union member    };
```

```
int main() {
```

```
union values arr1[2] = {{10}, {20}};
```

```
printf("arr1[0].integer_val: %d\n", arr1[0].integer_val);
```

```
printf("arr1[1].integer_val: %d\n", arr1[1].integer_val);
```

```
printf("Sizeof(union values): %d\n", sizeof(union values));
```

```
return 0;
```

```
}
```

```
arr1[0].integer_val: 10  
arr1[1].integer_val: 20  
Sizeof(union values): 4
```

Files

- A file is a container in computer storage devices used for storing data.

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.

Types of Files: there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

Text files are the normal **.txt** files. You can easily create text files using any simple text editors such as Notepad.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

2. Binary files

Binary files are mostly the **.bin** files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

File Operations

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

1. Creating a new file

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

```
FILE * fptr;
```

File Operations

2. Opening an existing file

Opening a file is performed using the `fopen()` function defined in the `stdio.h` header file.

```
fptr = fopen("fileopen", "mode");
```

```
fopen("E:\\cprogram\\newprogram.txt", "w");
```

writing as per the mode **'w'**.

existing file for reading in binary mode **'rb'**

File Operations

3. Closing a file

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using the `fclose()` function.

```
fclose(fptr);
```

4. Reading and writing to a text file

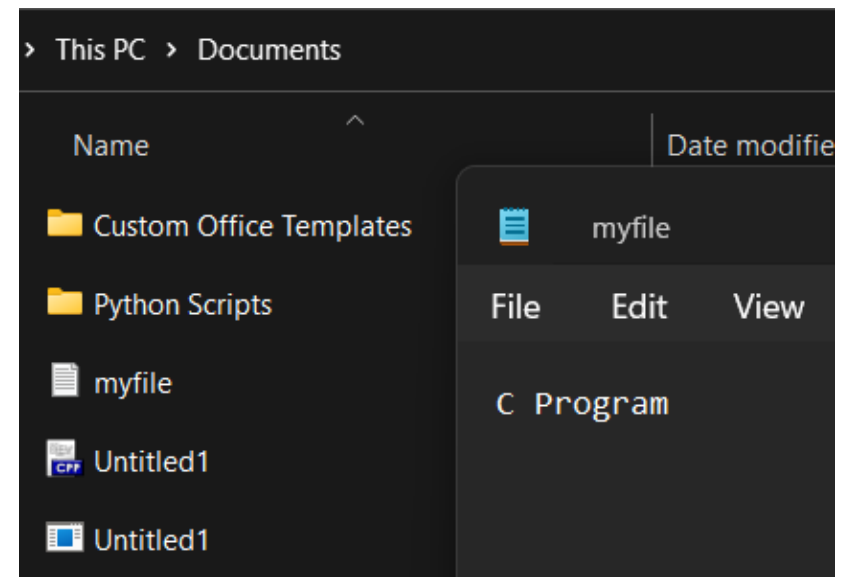
For reading and writing to a text file, we use the functions `fprintf()` and `fscanf()`.

File Operations - open and close mode

```
#include <stdio.h>

int main() {
    // Declare a file pointer
    FILE *file;
    // Open the file for reading
    file = fopen("myfile.txt", "r");
    // Check if the file was opened successfully
    if (file == NULL) {
        printf("Failed to open the file.\n");
        return 1; // Return an error code
    } // Read and print the content of the file
    char character;
    while ((character = fgetc(file)) != EOF) {
        putchar(character);
    } // Close the file
    fclose(file);
    return 0; // Return success
}
```

NULL is a special value in C that represents a null pointer, which essentially means that the pointer is not pointing to any valid memory location.

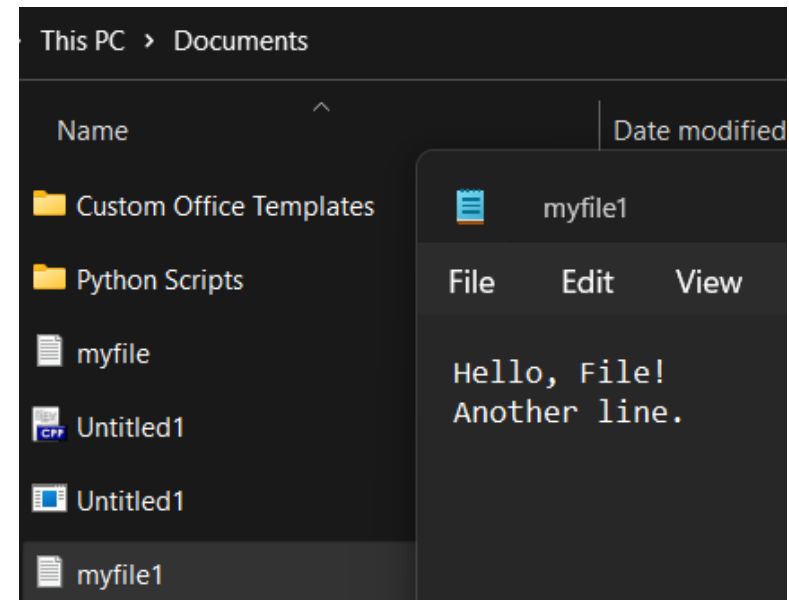


File Operations – Write mode

```
#include <stdio.h>

int main() { // Open for writing
    FILE *file = fopen("myfile1.txt", "w");

    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }
    fprintf(file, "Hello, File!\n");
    fputs("Another line.\n", file);
    fclose(file);
    return 0;
}
```



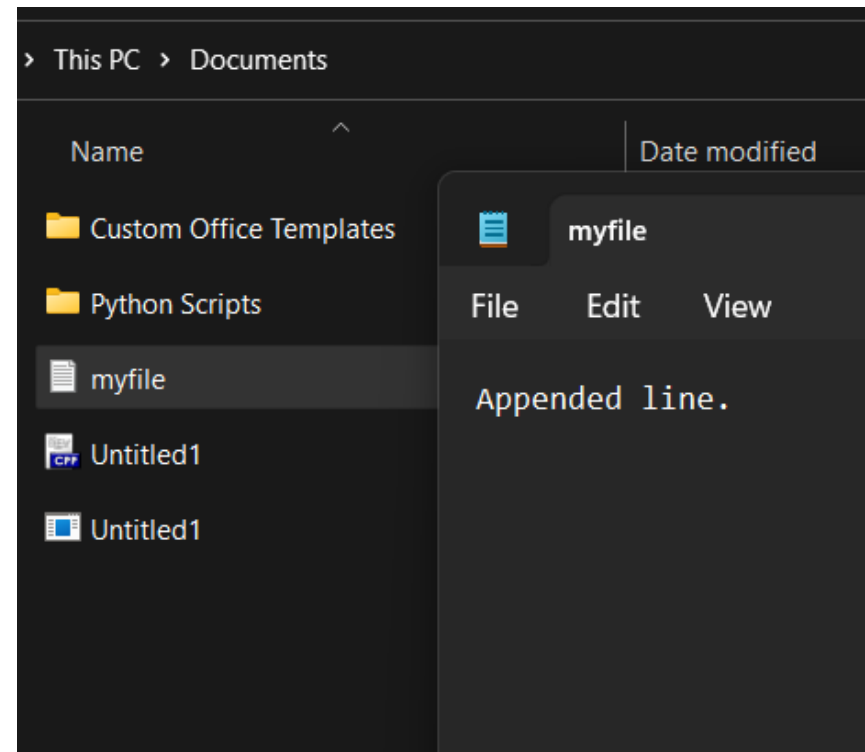
File Operations- Append

```
#include <stdio.h>

int main() {
    FILE *file = fopen("myfile.txt", "a"); // Open for appending

    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    fprintf(file, "Appended line.\n");
    fclose(file);
    return 0;
}
```



File Operations

Following are the most important file management functions available in 'C,'

function	purpose
fopen ()	Creating a file or opening an existing file
fclose ()	Closing a file
fprintf ()	Writing a block of data to a file
fscanf ()	Reading a block data from a file
getc ()	Reads a single character from a file
putc ()	Writes a single character to a file
getw ()	Reads an integer from a file
putw ()	Writing an integer to a file
fseek ()	Sets the position of a file pointer to a specified location
ftell ()	Returns the current position of a file pointer
rewind ()	Sets the file pointer at the beginning of a file

Detecting the End of File

- "End of file" (EOF) is a term used in computer programming and operating systems **to indicate the point in a file where there is no more data to be read.**
- When a program reads data from a file, it **typically keeps reading until it encounters the end of file marker**, at which point it knows that there is no more data to be processed.

Detecting the End of File

- The function **eof()** is used to check the end of file after EOF mark.
- It tests the end of file indicator.
- It returns non-zero value if successful otherwise, zero.

Step 1: Open file in write mode.

Step 2: Until character reaches end of the file, write each character in file pointer.

Step 3: Close file.

Step 4: Again open file in read mode.

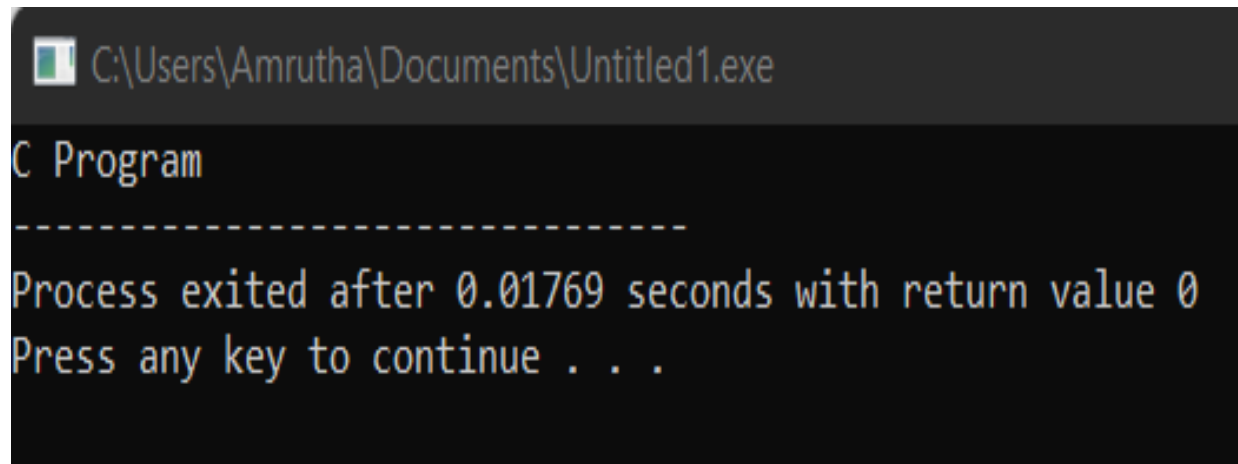
Step 5: Reading the character from file until file pointer equals to EOF.

Step 5: Print character on console.

Step 6: Close file.

Detecting the End of File

```
#include <stdio.h>
int main() {
    FILE *file = fopen("myfile.txt", "r"); // Open the file for reading
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }
    int ch;
    while ((ch = fgetc(file)) != EOF) {
        putchar(ch); // Print each character from the file
    }
    fclose(file);
    return 0;
}
```



C:\Users\Amrutha\Documents\Untitled1.exe

C Program

Process exited after 0.01769 seconds with return value 0

Press any key to continue . . .