**Unit V**

**C Structures (structs)**

Structures (also called structs) are a way to group several related variables into one place. Each variable in the structure is known as a **member** of the structure.

Unlike an [array](https://www.w3schools.com/c/c_arrays.php), a structure can contain many different data types (int, float, char, etc.).

**Create a Structure**

You can create a structure by using the struct keyword and declare each of its members inside curly braces:

struct MyStructure {   // Structure declaration  
  int myNum;           // Member (int variable)  
  char myLetter;       // Member (char variable)  
}; // End the structure with a semicolon

To access the structure, you must create a variable of it.

Use the struct keyword inside the main() method, followed by the name of the structure and then the name of the structure variable:

Create a struct variable with the name "s1":

struct myStructure {  
  int myNum;  
  char myLetter;  
};  
  
int main() {  
**struct myStructure s1;**  
  return 0;  
}

## Access Structure Members

To access members of a structure, use the dot syntax (.):

### Example

// Create a structure called myStructure  
struct myStructure {  
  int myNum;  
  char myLetter;  
};  
  
int main() {  
  // Create a structure variable of myStructure called **s1**  
  struct myStructure s1;  
  
  // Assign values to members of s1  
  s1.myNum = 13;  
  s1.myLetter = 'B';  
  
  // Print values  
  printf("My number: %d\n", s1.myNum);  
  printf("My letter: %c\n", s1.myLetter);  
  
  return 0;  
}

# C Array of Structures

## Array of Structures in C

An array of structres in [C](https://www.javatpoint.com/c-programming-language-tutorial) can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of [structures in C](https://www.javatpoint.com/structure-in-c) are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures

## Why use an array of structures?

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

#include<stdio.h>

1. **struct** student
2. {
3. **char** name[20];
4. **int** id;
5. **float** marks;
6. };
7. **void** main()
8. {
9. **struct** student s1,s2,s3;
10. **int** dummy;
11. printf("Enter the name, id, and marks of student 1 ");
12. scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
13. scanf("%c",&dummy);
14. printf("Enter the name, id, and marks of student 2 ");
15. scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
16. scanf("%c",&dummy);
17. printf("Enter the name, id, and marks of student 3 ");
18. scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
19. scanf("%c",&dummy);
20. printf("Printing the details....\n");
21. printf("%s %d %f\n",s1.name,s1.id,s1.marks);
22. printf("%s %d %f\n",s2.name,s2.id,s2.marks);
23. printf("%s %d %f\n",s3.name,s3.id,s3.marks);
24. }

**Output**

Enter the name, id, and marks of student 1 James 90 90

Enter the name, id, and marks of student 2 Adoms 90 90

Enter the name, id, and marks of student 3 Nick 90 90

Printing the details....

James 90 90.000000

Adoms 90 90.000000

Nick 90 90.000000

In the above program, we have stored data of 3 students in the structure. However, the complexity of the program will be increased if there are 20 students. In that case, we will have to declare 20 different structure variables and store them one by one. This will always be tough since we will have to declare a variable every time we add a student. Remembering the name of all the variables is also a very tricky task. However, c enables us to declare an array of structures by using which, we can avoid declaring the different structure variables; instead we can make a collection containing all the structures that store the information of different entities.

## Features of structures

The features are listed below:

### 1) Using the “=” operator

Structure elements are stored in the memory in contiguous memory locations which allow us the use of ***“=”***(the equals to assignment operator) to copy the entire elements of one structure to another. This is not possible in case of arrays where we need to copy all the elements one by one in order to copy the entire elements. Other than equating the two structure variables the user can also copy the elements piece-meal. But that would be more complex. Here is an example of how we can simply equate two structures to copy the values from one to another.

### 2) Nested Structures

C programming allows us to create ***nested structures***,i.e., one structure within another structure. This property enables us to create complex data types for storing various elements. The nesting process can continue as long as the user wants.

### 3) Passing Structure Variables to a Function

This feature of structure allows us to pass any of the structure variables to the function like any other normal variable. This can be done in two ways: **Passing individual structure elements**to the function or ***passing the entire structure*** to the function. Let us have a look at both of them:

# Advantages of Structure in C Programming

In C programming, structure is a collection of different data items which are referenced by single name. It is also known as user-defined data-type in C.

Using structure in C language has several benefits. In this article we are going to list key advantages of structure while programming in C.

1. **Heterogeneous collection of data items:** structure allows us to create user defined data-type which can store items with different data types. For example, if we want to store students' records with their roll number, name, marks and address then it comprises items with different types i.e. roll number can be of integer number type, name is of string type, marks is floating number type and address is of string type.  
   In C, it can be achieved by using structure like this:
2. **struct** student
3. {
4. **int** roll\_number;
5. **char** name[30];
6. **float** marks;
7. **char** address[50];
8. };

Now structure variables can be created using struct student s1, s2; and these variables can store all the details of two students.

1. **Reduced complexity:** think of storing student records of five different students with roll number, name, marks and address like in above example without structure! What do we need here? Five integer type variables for storing roll number of five students, five string variables for storing name of five students, five floating type variables for storing marks, and again five string variables for storing address of five students. Terrible! Don’t do it that way! Just create structure and create an array of structure. That’s it!  
   Here is simple setup for doing that:
2. **struct** student
3. {
4. **int** roll\_number;
5. **char** name[30];
6. **float** marks;
7. **char** address[50];
8. };
9. **struct** student s[5];
10. **Increased productivity:** structure in C eliminates lots of burden while dealing with records which contain heterogeneous data items, thereby increasing productivity.
11. **Maintainability of code:** using structure, we represent complex records by using a single name, which makes code maintainability like a breeze.
12. **Enhanced code readability:** code readability is crucial for larger projects. Using structure code looks user friendly which in turn helps to maintain the project.
13. **Suitable for some mathematical operations:** by using basic data-types we can not represent complex numbers, distances (feet-inch system), times (hour-minute-second system). Mathematical operations like adding complex numbers, or subtracting them; adding two distances in a feet-inch system; finding the difference between two time periods in an hour-minute-second system can be done effectively using the concept of structure.  
    Here is simple setup for dealing with complex numbers:

**struct** complex

1. {
2. **float** real;
3. **float** imaginary;
4. };

**struct** complex c1, c2, sum;

# Basics of File Operations in C

So far the operations using C program are done on a prompt / terminal which is not stored anywhere. But in the software industry, most of the programs are written to store the information fetched from the program. One such way is to store the fetched information in a file. Different operations that can be performed on a file are: 

1. Creation of a new file (**fopen with attributes as “a” or “a+” or “w” or “w+”)**
2. Opening an existing file (**fopen**)
3. Reading from file (**fscanf or fgets**)
4. Writing to a file (**fprintf or fputs**)
5. Moving to a specific location in a file (**fseek, rewind**)
6. Closing a file (**fclose**)

**Opening or creating file**   
For opening a file, fopen function is used with the required access modes. Some of the commonly used file access modes are mentioned below

**Closing a file –**:   
After every successful file operations, you must always close a file. For closing a file, you have to use fclose function. The snippet for closing a file is given as : 

FILE \*filePointer ;

filePointer= fopen(“fileName.txt”, “w”);

---------- Some file Operations -------

fclose(filePointer)

# include <stdio.h>

# include <string.h>

 int main( )

{

    // Declare the file pointer

    FILE \*filePointer ;

    // Get the data to be written in file

    char dataToBeWritten[50]

        = "GeeksforGeeks-A Computer Science Portal for Geeks";

    // Open the existing file GfgTest.c using fopen()

    // in write mode using "w" attribute

    filePointer = fopen("GfgTest.c", "w") ;

    // Check if this filePointer is null

    // which maybe if the file does not exist

    if ( filePointer == NULL )

    {

        printf( "GfgTest.c file failed to open." ) ;

    }

    else

    {

        printf("The file is now opened.\n") ;

        // Write the dataToBeWritten into the file

        if ( strlen ( dataToBeWritten ) > 0 )

        {

            // writing in the file using fputs()

            fputs(dataToBeWritten, filePointer) ;

            fputs("\n", filePointer) ;

        }

        // Closing the file using fclose()

        fclose(filePointer) ;

        printf("Data successfully written in file GfgTest.c\n");

        printf("The file is now closed.") ;

    }

    return 0;

}

**C Program to count the Number of Characters in a File**

The characters can be counted easily by reading the characters in the file using [getc()](https://www.geeksforgeeks.org/eof-and-feof-in-c/) method. For each character read from the file, increment the counter by one.

// C Program to count

// the Number of Characters in a Text File

#include <stdio.h>

#define MAX\_FILE\_NAME 100

int main()

{

    FILE\* fp;

    // Character counter (result)

    int count = 0;

    char filename[MAX\_FILE\_NAME];

    // To store a character read from file

    char c;

    // Get file name from user.

    // The file should be either in current folder

    // or complete path should be provided

    printf("Enter file name: ");

    scanf("%s", filename);

    // Open the file

    fp = fopen(filename, "r");

    // Check if file exists

    if (fp == NULL) {

        printf("Could not open file %s",

               filename);

        return 0;

    }

    // Extract characters from file

    // and store in character c

    for (c = getc(fp); c != EOF; c = getc(fp))

        // Increment count for this character

        count = count + 1;

    // Close the file

    fclose(fp);

    // Print the count of characters

    printf("The file %s has %d characters\n ",

           filename, count);

    return 0;

}

# C Program to count characters, tabs, spaces and lines in a file

C Program to count characters, tabs, spaces and lines in a file using the concept of file handling.

#include<stdio.h>

int main(){

FILE \*fp;

char ch;

int nol=0,not=0,nob=0,noc=0;

fp=fopen("codin.txt","r");

if(fp==NULL)

printf("Not found");

while(1){

ch=fgetc(fp);

if(ch==EOF)

break;

noc++;

if(ch==' ')

nob++;

if(ch=='\n')

nol++;

if(ch=='\t')

not++;

}

fclose(fp);

printf("Number of characters=%d\n",noc);

printf("Number of blanks=%d\n",nob);

printf("Number of tabs=%d\n",not);

printf("Number of lines=%d\n",nol);

return 0;

}

**File opening modes in C:**

* **“r” –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
* **“rb” –** Open for reading in binary mode. If the file does not exist, fopen( ) returns NULL.
* **“w” –** Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.
* **“wb” –** Open for writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
* **“a” –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.
* **“ab” –** Open for append in binary mode. Data is added to the end of the file. If the file does not exist, it will be created.
* **“r+” –** Searches file. If is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.
* **“rb+” –** Open for both reading and writing in binary mode. If the file does not exist, fopen( ) returns NULL.
* **“w+” –** Searches file. If the file exists, its contents are overwritten. If the file doesn’t exist a new file is created. Returns NULL, if unable to open file.
* **“wb+” –** Open for both reading and writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
* **“a+” –** Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer which points to the last character in it. If the file doesn’t exist, a new file is created. Returns NULL, if unable to open file.
* **“ab+” –** Open for both reading and appending in binary mode. If the file does not exist, it will be created.
* **Reading from a file –**   
  The file read operations can be performed using functions fscanf or fgets. Both the functions performed the same operations as that of scanf and gets but with an additional parameter, the file pointer. So, it depends on you if you want to read the file line by line or character by character.  
  And the code snippet for reading a file is as:

FILE \* filePointer;

filePointer = fopen(“fileName.txt”, “r”);

fscanf(filePointer, "%s %s %s %d", str1, str2, str3, &year);

* **Writing a file –**:  
  The file write operations can be performed by the functions fprintf and fputs with similarities to read operations. The snippet for writing to a file is as :

FILE \*filePointer ;

filePointer = fopen(“fileName.txt”, “w”);

fprintf(filePointer, "%s %s %s %d", "We", "are", "in", 2012);

# C program to copy contents of one file to another file

int main()

{

    FILE \*fptr1, \*fptr2;

    char filename[100], c;

    printf("Enter the filename to open for reading \n");

    scanf("%s", filename);

    // Open one file for reading

    fptr1 = fopen(filename, "r");

    if (fptr1 == NULL)

    {

        printf("Cannot open file %s \n", filename);

        exit(0);

    }

    printf("Enter the filename to open for writing \n");

    scanf("%s", filename);

    // Open another file for writing

    fptr2 = fopen(filename, "w");

    if (fptr2 == NULL)

    {

        printf("Cannot open file %s \n", filename);

        exit(0);

    }

    // Read contents from file

    c = fgetc(fptr1);

    while (c != EOF)

    {

        fputc(c, fptr2);

        c = fgetc(fptr1);

    }

    printf("\nContents copied to %s", filename);

    fclose(fptr1);

    fclose(fptr2);

    return 0;

}

# Enumeration (or enum) in C

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

enum State {Working = 1, Failed = 0};

The keyword ‘enum’ is used to declare new enumeration types in C and C++. Following is an example of enum declaration. 

// The name of enumeration is "flag" and the constant

// are the values of the flag. By default, the values

// of the constants are as follows:

// constant1 = 0, constant2 = 1, constant3 = 2 and

// so on.

enum flag{constant1, constant2, constant3, ....... };

Variables of type enum can also be defined. They can be defined in two ways:

# typedef in C

The **typedef** is a keyword used in C programming to provide some meaningful names to the already existing variable in the [C program](https://www.javatpoint.com/c-programs). It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.

### Syntax of typedef

1. **typedef** <existing\_name> <alias\_name>

In the above syntax, '**existing\_name'** is the name of an already existing variable while '**alias name'** is another name given to the existing variable.

For example, suppose we want to create a variable of type **unsigned int**, then it becomes a tedious task if we want to declare multiple variables of this type. To overcome the problem, we use **a typedef** keyword.

1. **typedef** unsigned **int** unit;

In the above statements, we have declared the **unit** variable of type unsigned int by using **a typedef** keyword

Now, we can create the variables of type **unsigned int** by writing the following statement:

1. unit a, b;

instead of writing:

1. unsigned **int** a, b;

Till now, we have observed that the **typedef** keyword provides a nice shortcut by providing an alternative name for an already existing variable. This keyword is useful when we are dealing with the long data type especially, structure declarations.

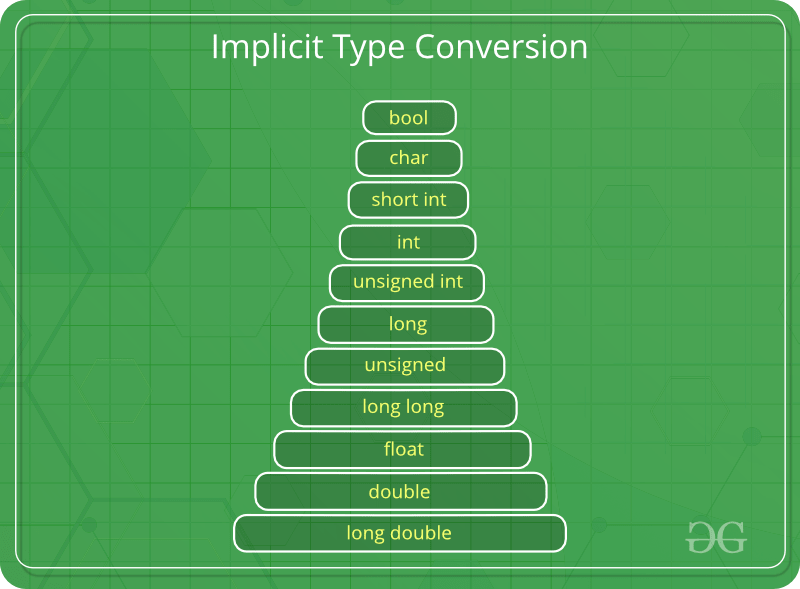
**Let's understand through a simple example.**

1. #include <stdio.h>
2. **int** main()
3. {
4. **typedef** unsigned **int** unit;
5. unit i,j;
6. i=10;
7. j=20;
8. printf("Value of i is :%d",i);
9. printf("\nValue of j is :%d",j);
10. **return** 0;
11. }

# C - Type Casting

Type conversion in C is the process of converting one data type to another. The type conversion is only performed to those data types where conversion is possible. Type conversion is performed by a compiler. In type conversion, the destination data type can’t be smaller than the source data type. Type conversion is done at compile time and it is also called widening conversion because the destination data type can’t be smaller than the source data type. ***There are two types of Conversion:***

## 1. Implicit Type Conversion



An example of implicit conversion

#include <stdio.h>

int main()

{

    int x = 10; // integer x

    char y = 'a'; // character c

    // y implicitly converted to int. ASCII

    // value of 'a' is 97

    x = x + y;

    // x is implicitly converted to float

    float z = x + 1.0;

    printf("x = %d, z = %f", x, z);

    return 0;

}

**2. Explicit Type Conversion**

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type. The syntax in C Programming:

(type) expression

Type indicated the data type to which the final result is converted.

|  |
| --- |
| // C program to demonstrate explicit type casting  #include<stdio.h>      int main()  {      double x = 1.2;          // Explicit conversion from double to int      int sum = (int)x + 1;        printf("sum = %d", sum);          return 0;  } |

**Output**

sum = 2

### Advantages of Type Conversion

* This is done to take advantage of certain features of type hierarchies or type representations.
* It helps us to compute expressions containing variables of different data types.
* The accuracy of our result increases with the help of type conversion.
* Before performing operations, we can refer to the conversion rank hierarchy to get better results.

# Bit Fields in C

In C, we can specify the size (in bits) of the structure and union members. The idea of bit-field is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range. Bit fields are used when the storage of our program is limited. Need of bit fields in C programming language:

* Reduces memory consumption.
* To make our program more efficient and flexible.
* Easy to Implement.

**Applications of Bit-fields:**

* If storage is limited, we can go for bit-field.
* When devices transmit status or information encoded into multiple bits for this type of situation bit-field is most efficient.
* Encryption routines need to access the bits within a byte in that situation bit-field is quite useful.

For example, consider the following declaration of date without the use of bit fields.

* C

|  |
| --- |
| #include <stdio.h>    // A simple representation of the date  struct date {      unsigned int d;      unsigned int m;      unsigned int y;  };    int main()  {      printf("Size of date is %lu bytes\n", sizeof(struct date));      struct date dt = { 31, 12, 2014 };      printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);  } |

### Declaration of bit-fields in C

Bit-fields are variables that are defined using a predefined width or size. Format and the declaration of the bit-fields in C are shown below:

**Syntax:**

struct

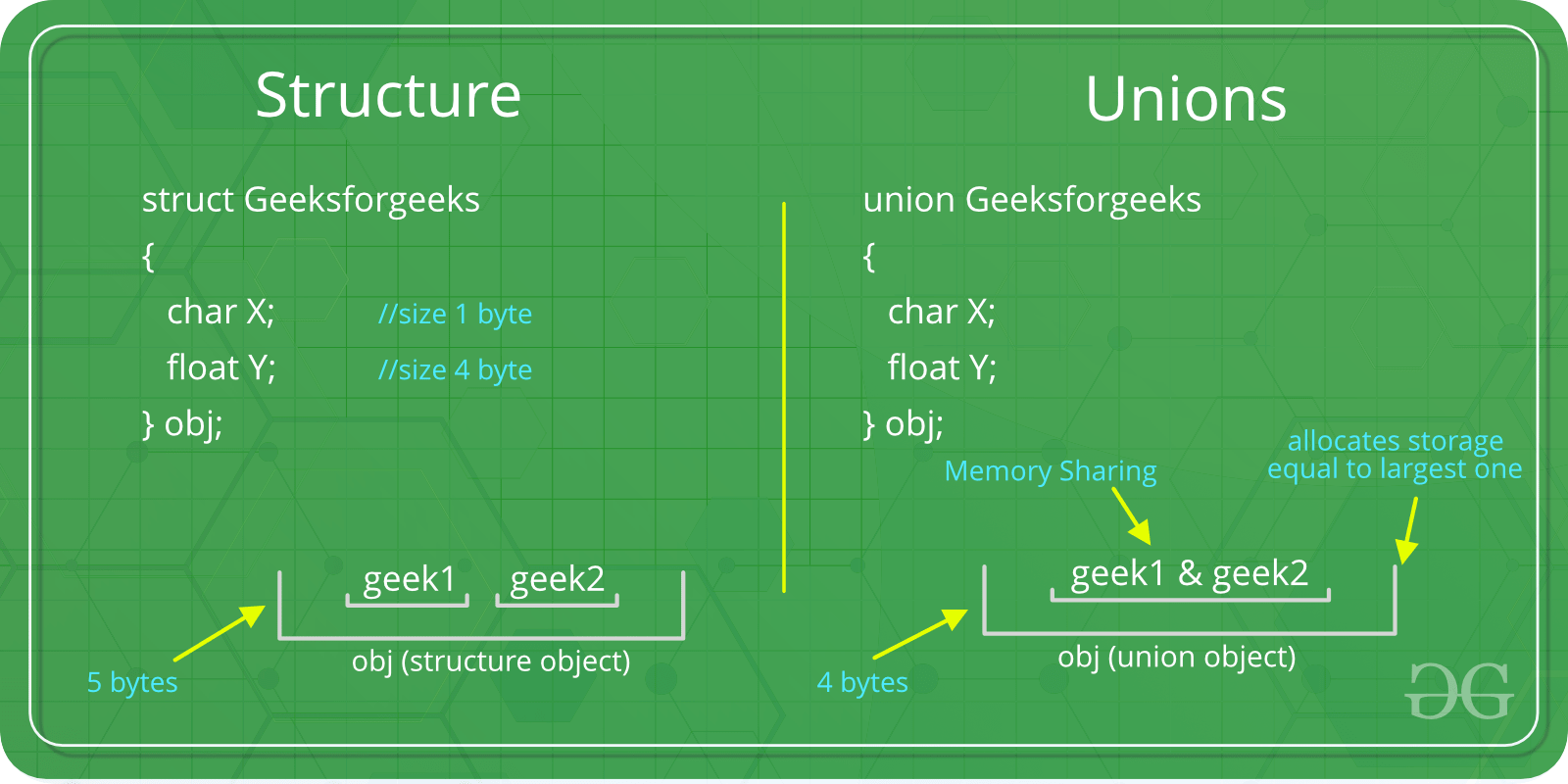
{

data\_type member\_name: width\_of\_bit-field;

};

# Union in C

Like [Structures](https://www.geeksforgeeks.org/structures-c/), union is a user defined data type. In union, all members share the same memory location.



For example in the following C program, both x and y share the same location. If we change x, we can see the changes being reflected in y.

|  |
| --- |
| #include <stdio.h>    // Declaration of union is same as structures  union test {      int x, y;  };    int main()  {      // A union variable t      union test t;        t.x = 2; // t.y also gets value 2      printf("After making x = 2:\n x = %d, y = %d\n\n",             t.x, t.y);        t.y = 10; // t.x is also updated to 10      printf("After making y = 10:\n x = %d, y = %d\n\n",             t.x, t.y);      return 0;  } |