Unit : 5

Structure / union

Introduction

- 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.

How to Defining a Structure?

- To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows –

```
struct [structure tag]
{
   member definition;
   member definition;
} [one or more structure variables];
```

Accessing Structure Members

- 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.

Union

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

Typedef keyword

- The **Typedef** Keyword in **C** and C++ The **typedef** keyword allows the programmer to create new names for **types** such as int or, more commonly in C++, templated **types**--it literally stands for "**type** definition".

Scope of variables

- A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language.

1. Inside a function or a block which is called **local** variables.
2. Outside of all functions which is called **global** variables.
3. In the definition of function parameters which are called **formal** parameters.

Local Variables

- Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

Global Variables

- Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.
- A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration.

Formal Parameters

- Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.

Type casting

- Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the **cast operator** as follows –

- (type_name) expression

Example

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int sum = 17, count = 5;
   double mean;
   clrscr();
   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );
   getch();
}
```

Header Files

- A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.
- You request to use a header file in your program by including it with the C preprocessing directive **#include**, like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

- Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

Include Syntax

- Both the user and the system header files are included using the preprocessing directive **#include**.

C Storage Class

- A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program –

1. auto
2. register
3. static
4. extern

The auto Storage Class

- The **auto** storage class is the default storage class for all local variables.
- { int mount; auto int month; }
- The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

- The **register** storage class is used to define local variables that should be stored in a register instead of RAM.

  { register int miles; }

- The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

- The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

- he static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.
- In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class.