Parul® University

Programming for Problem Solving **(PPS)**

Chapter-5

User-Defined Functions, Structure and Unions

Functions

In Functions, you will learn about

- Introduction to function
- Concepts of User define function
  - ➢ Function prototype (declaration)
  - ➢ Function definition
  - ➢ Function call and return
- Parameters
  - ➢ Actual and Formal
- Parameter Passing
- Calling a function
- Recursive function
- Macros
- Pre-Processing

Introduction to Function

- A function is a block of code which is used to perform a specific task.
- It can be written in one program and used by another program without having to rewrite that piece of code. Hence, it promotes usability!!!
- Functions can be put in a library. If another program would like to use them, it will just need to include the appropriate header file at the beginning of the program and link to the correct library while compiling.

Introduction to Function

Functions can be divided into two categories :

- Predefined functions (standard functions)
  - Built-in functions provided by C that are used by programmers without having to write any code for them. i.e: printf( ), scanf( ), etc
- User-Defined functions
  - Functions that are written by the programmers themselves to carry out various individual tasks.

Predefined (Standard) Functions

- Standard functions are functions pre-defined by C and put into standard C libraries.
  - Example: printf(), scanf(), pow(), ceil(), rand(), etc.
- We need to include the appropriate header files.
  - Example: #include <stdio.h>, #include <math.h>
- What contained in the header files are the prototypes of the standard functions. The function definitions (the body of the functions) has been compiled and put into a standard C library which will be linked by the compiler during compilation.

User defined functions

- A programmer can create his/her own function(s).
- It is easier to plan and write our program if we divide it into several functions instead of writing a long piece of code inside the main function.
- A function is **reusable** and therefore prevents us (programmers) from having to unnecessarily rewrite what we have written before.
- In order to write and use our own function, we need to do the following:
  - create a function prototype (declare the function)
  - define the function somewhere in the program (implementation)
  - call the function whenever it needs to be used

Function Prototype (Declare the function)

- A function prototype will tell the compiler that there exist a function with this name defined somewhere in the program and therefore it can be used even though the function has not yet been defined at that point.
- Function prototypes need to be written at the beginning of the program.
- If the function receives some arguments, the variable names for the arguments are not needed. State only the data types

Function Prototype (Declare the function)

- Function prototypes can also be put in a header file. Header files are files that have a .h extension.
- The header file can then be included at the beginning of our program.
- To include a user defined header file, type:

```
#include “header_file.h”
```

- Notice that instead of using < > as in the case of standard header files, we need to use “ ”. This will tell the compiler to search for the header file in the same directory as the program file instead of searching it in the directory where the standard library header files are stored.

Function Prototype (Declare the function) Example

- A function prototype Consists of the following elements:
  - function’s name
  - Parameters
  - Return type.
- It must appear before the function is called.

Example:

int add(int, int); // function prototype

Function Definitions

- Is also called as *function implementation*
- A function definition is where the actual code for the function is written. This code will determine what the function will do when it is called.
- A function definition consists of the following elements:
  - ➢ A function name
  - ➢ An optional list of formal parameters enclosed in parentheses (function arguments)
  - ➢ A function return data type (return value)
  - ➢ A compound statement (function body)

Function Definition

- A function definition has this format:

```
return_data_type FunctionName(data_type variable1,  data_type
variable2, data_type variable3, …..)
        {
                local variable declarations;
                statements;
        }
```

- The return data type is the type of data that will be returned to the calling function. There can be only one return value.
- Any function not returning anything must be of type 'void'.
- If the function does not receive any arguments from the calling function, 'void' is used in the place of the arguments.

Function Definition Example 1

This is where the actual code of the function is written.
The definition must match the prototype.

Example:

```
int add(int a, int b) {
   return a + b;
}
```

Function Definition Example 2

- A simple function is :

```
void print_message (void)
{
 printf("Hello, are you having fun?\n");
}
```

- Note the function name is ***print_message***. No arguments are accepted by the function, this is indicated by the keyword ***void*** enclosed in the parentheses. The return_value is ***void***, thus data is not returned by the function.
- So, when this function is called in the program, it will simply perform its intended task which is to print ***Hello, are you having fun?***

Function Definition Example 3

- Consider the following example:

```
int calculate (int num1, int num2)
{
        int sum;
        sum = num1 + num2;
        return sum;
}
```

- The above code segments is a function named ***calculate***. This function accepts two arguments i.e. ***num 1*** and ***num2*** of the type ***int***. The return_value is ***int***, thus this function will return an integer value.
- So, when this function is called in the program, it will perform its task which is to **calculate the sum of any two numbers** and **return the result of the summation.**
- Note that if the function is returning a value, it needs to use the keyword 'return'.

Function Call and Return

Any function (including main) could utilize any other function definition that exist in a program – hence it is said to call the function (function call).

To call a function (i.e. to ask another function to do something for the calling function), it requires the FunctionName followed by a list of actual parameters (or arguments), if any, enclosed in parenthesis.

Function Call and Return: Example

If the function requires some arguments to be passed along, then the arguments need to be listed in the bracket ( ) according to the specified order.

For example:

```
void Calc(int, double, char, int);
void main(void) {
    int a, b;
    double c;
    char d;
    …
    Calc(a, c, d, b);
}
```

Function Call

Function Call and Return: Example

If the function returns a value, then the returned value need to be assigned to a variable so that it can be stored. For example:

```
int GetUserInput (void); /* function prototype*/
void main(void) {
    int input;
    input = GetUserInput( );
}
```

However, it is perfectly okay (syntax wise) to just call the function without assigning it to any variable if we want to ignore the returned value.

We can also call a function inside another function. For example:

```
printf("User input is: %d", GetUserInput( ));
```

Basic Skeleton

```
#include <stdio.h>
//function prototype
//global variable declaration
void main(void)
{
    local variable declaration;
    statements;
    fn1( );
    fn2( );
}
```

```
void fn1(void)
{
    local variable declaration;
    statements;
}

void fn2(void)
{
    local variable declaration;
    statements;
}
```

Examples

A function may

Receive no input parameter and return nothing

Receive no input parameter but return a value

Receive input parameter(s) and return nothing Receive input parameters(s) and return a value

Parul® University

Example 1: Receive nothing and Return nothing

```
#include <stdio.h>
void greeting(void); /* function
prototype */

void main(void)
{
        greeting( );
        greeting( );
}

void greeting(void)
{
        printf("Have fun!! \n");
}
```

- In this example, function greeting does not receive any arguments from the calling function (main), and does not return any value to the calling function, hence type 'void' is used for both the input arguments and return data type.
- The output of this program is:

```
Have fun!!
Have fun!!
```

Example 2: Receive nothing and Return a value

```
#include <stdio.h>
int getInput(void);


void main(void)
{
        int num1, num2, sum;
        num1 = getInput( );
        num2 = getInput( );
        sum = num1 + num2;
        printf("Sum is %d\n",sum);
}
```

```
int getInput(void)
{
     int number;
     printf("Enter a number:");
     scanf("%d",&number);

     return number;
}
```

Sample Output:

```
Enter a number: 3
Enter a number: 5
Sum is 8
```

Example 3: Receive parameters and Return nothing

```
#include <stdio.h>
int getInput(void);
void displayOutput(int);
void main(void)
{
        int num1, num2, sum;

        num1 = getInput();
        num2 = getInput();

        sum = num1 + num2;

        displayOutput(sum);
}
```

```
int getInput(void)
{
     int number;
     printf("Enter a number:");
     scanf("%d",&number);
     return number;
}

void displayOutput(int sum)
{
     printf("Sum is %d \n",sum);
}
```

Sample Output:

```
Enter a number: 3
Enter a number: 5
Sum is 8
```

Example 4: Receive parameters and Return a value

```
#include <stdio.h>
int calSum(int,int); /*function protototype*/

void main(void)
{
        int sum, num1, num2;

        printf("Enter two numbers to calculate its sum:\n");

        scanf("%d%d",&num1,&num2);

        sum = calSum(num1,num2);  /* function call */
        printf("\n %d + %d = %d", num1, num2, sum);
}
```

```
int calSum(int val1, int val2)
/*function definition*/
{
        int sum;
        sum = val1 + val2;
        return sum;
}
```

Sample Output:

```
Enter two numbers to calculate
its sum:
3
5


 3 + 5 = 8
```

Example 4: Explanation

- In this example, the calSum function receives input parameters of type int from the calling function (main).
- The calSum function returns a value of type int to the calling function.
- Therefore, the **function definition** for calSum:

```
int calSum(int val1, int val2)
```

- Note that the **function prototype** only indicates the type of variables used, not the names.

```
int calSum(int,int);
```

- Note that the function call is done by (main) calling the function name (calSum), and supplying the variables (num1,num2) needed by the calSum function to carry out its task.

Function – Complete Flow

```
#include<stdio.h>
int calSum(int, int);
void main( )
{
 int num1, num2, sum;
 printf("Enter 2 numbers to calculate its sum:\n");
 scanf("%d %d", &num1, &num2);
 sum = calSum (num1, num2);
 printf ("\n %d + %d = %d", num1, num2, sum);
}
int calSum(int val1, int val2)
{
 int sum;
 sum = val1 + val2;
 return sum;
}
```

Sample Output:

```
Enter two numbers to calculate its sum:
3
5


 3 + 5 = 8
```

Function with Parameter/Argument
Formal and Actual Parameter

- When a function calls another function to perform a task, the calling function may also send data to the called function. After completing its task, the called function may pass the data it has generated back to the calling function.
- Two terms used:
  - Formal parameter
    - Variables declared in the formal list of the function header (written in function prototype & function definition)
  - Actual parameter
    - Constants, variables, or expression in a function call that correspond to its formal parameter

Function with Parameter/Argument

The three important points concerning functions with parameters are: (number, order and type)

- The number of actual parameters in a function call must be the same as the number of formal parameters in the function definition.
- A one-to-one correspondence must occur among the actual and formal parameters. The first actual parameter must correspond to the first formal parameter and the second to the second formal parameter, an so on.
- The type of each actual parameter must be the same as that of the corresponding formal parameter.

Formal and Actual Parameter: Example

```
#include <stdio.h>
int calSum(int,int);/*function prototype*/

void main(void)
{
 int sum, num1, num2;
 printf("Enter two numbers to calculate its
sum:\n");
 scanf("%d%d",&num1,&num2);
 sum = calSum(num1,num2);  /* function call */
// num1 and num2 are actual parameters
 printf("\n %d + %d = %d", num1, num2, sum);
}
```

```
int calSum(int val1, int val2)
/*function definition*/
// val1 and val2 are formal parameters
{
 int sum;
 sum = val1 + val2;
 return sum;
}
```

Sample Output:

```
Enter two numbers to calculate
its sum:
3
5


 3 + 5 = 8
```

Parameter Passing

There are 2 ways to call a function:
Parameter/argument Passing to a function

Call by value

- In this method, only the **copy of variable's value** (copy of actual parameter's value) is passed to the function. Any modification to the passed value inside the function **will not** affect the actual value.
- In all the examples that we have seen so far, this is the method that has been used.

Call by reference

- In this method, the **reference** (memory address) of the variable is passed to the function. Any modification passed done to the variable inside the function **will** affect the actual value.
- To do this, we need to have knowledge about pointers and arrays, which will be discussed in chapter pointer.

Call by value: Example

```
#include <stdio.h>
void change(int data);
int main()
{
   int data = 3;
   change(data);
   printf("Value of the data is: %d\n", data);
   return 0;
}
void change(int data)
{
   data = 5;
}
```

Output:

```
Value of the data is: 3
```

Call by reference/pointer: Example

```
#include <stdio.h>
void swap(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
int main()
{
   int x = 500, y = 100;
   swap(&x, &y); // passing address to function (call by reference)
   printf("Value of x is: %d\n", x);
   printf("Value of y is: %d\n", y);
   return 0;
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Calling a function without Argument: Example

```
#include <stdio.h>

void greet() {
printf("Hello from the greet function!\n");
}

int main() {
greet(); // Calling the greet function
return 0;}
```

Output:
```
Hello from the greet function!
```

Calling a function with Arguments and Return Value: Example

```
#include <stdio.h>
int add(int a, int b) {
return a + b;
}

int main() {
int num1 = 5;
int num2 = 10;
int sum_result;
sum_result = add(num1, num2); // Calling the add function with arguments and return value
printf("The sum is: %d\n", sum_result);
return 0;
}
```

Output:

```
The sum is: 15
```

Recursive Function

- **Recursion** is a programming technique where a function calls itself repeatedly until a specific base condition is met.
- A function that performs such self-calling behavior is known as a **recursive function**, and each instance of the function calling itself is called a **recursive call**.
- It typically consists of two main parts:

Base Case – the condition under which the recursion stops.

Recursive Case – where the function calls itself with modified arguments, gradually approaching the base case.

Recursive Function Syntax

```
returntype function(parameters) {

   // base case
   if (base condition) {
      return base value;
   }
   // recursive case
   return recursive expression involving function(modified
parameters);
}
```

Recursive Function: Example

```
#include <stdio.h>
void rec(int n) {
    // Base Case
    if (n == 6) return;
    printf("Recursion Level %d\n", n);
    rec(n + 1);
}

int main() {
    rec(1);
    return 0;
}
```

Output:

```
Recursion Level 1
Recursion Level 2
Recursion Level 3
Recursion Level 4
Recursion Level 5
```

Parul® University

DIGITAL LEARNING CONTENT

Recursive Function: Factorial Example

```
#include <stdio.h>
// Recursive function to calculate factorial
long long int factorial(int n)
{
// Base case: factorial of 0 is 1
if (n == 0) {
return 1;
} else {
// Recursive case: n * factorial of (n-1)
return n * factorial(n - 1);
}
}
```

```
int main() {
int num;
// Prompt user to enter a non-negative integer
printf("Enter a non-negative integer: ");
scanf("%d", &num);
// Validate input
if (num < 0) {
printf("Factorial is not defined for negative numbers.\n");
} else
{
// Calculate and print the factorial
printf("Factorial of %d is %lld\n", num, factorial(num));
}
return 0;
}
```

```
Enter a non-negative integer: 5
Factorial of 5 is 120
```

Macros

- In C programming, a macro is a segment of code that is assigned a name and processed by the preprocessor before the actual compilation.
- Macros are defined using the #define directive.
- When the preprocessor encounters a macro name in the code, it replaces it with the corresponding code segment, a process known as macro expansion.

Macros: Object-Like

There are primarily two types of macros:

Object-like Macros:

- These macros are used to define constants or simple values.
- They don't take any arguments.

Example:

```
#include <stdio.h>
#define PI 3.14159 // Defining a constant PI
int main() {
printf("Value of PI: %f\n", PI);
return 0;
}
```

In this example PI will be replaced by 3.14159 during preprocessing

Output:

```
Value of PI: 3.141590
```

Macros: Function-Like

Function-like Macros:

- These macros resemble functions and can take arguments.
- They are often used for code reusability or to create inline functions for performance optimization.

Macros: Function-Like

Example

```
#include <stdio.h>
#define SQUARE(x) ((x) * (x)) // Defining a macro to calculate square
int main() {
int num = 5;
int result = SQUARE(num); // Macro expansion: ((num) * (num))
printf("Square of %d is %d\n", num, result);
int a = 3, b = 7;
#define MAX(a, b) ((a) > (b) ? (a) : (b)) // Macro to find maximum of
two numbers
int largest = MAX(a, b);
printf("The largest of %d and %d is %d\n", a, b, largest);
return 0;
}
```

In this example, SQUARE(num) expands to ((num) * (num)), and MAX(a, b) expands to ((a) > (b) ? (a) : (b)) during preprocessing.

The parentheses in function-like macros are crucial to prevent unexpected behavior due to operator precedence issues during expansion.

Pre Processing

- Preprocessing in C refers to the phase that occurs before the actual compilation of a C program.
- The C preprocessor is a macro processor that modifies the source code based on preprocessor directives.
- These directives always begin with a hash symbol (#).
- The preprocessor does not understand C syntax; it performs text-based substitutions and manipulations.

Pre Processing Directives

Common Preprocessor Directives:

•#include: Used to include the content of a specified file (usually a header file) into the current source file. This is crucial for using functions and declarations defined in other files.

```
#include <stdio.h> // Includes standard input/output library functions
#include "myheader.h" // Includes a user-defined header file
```

•#define: Used to define macros, which are symbolic names or code snippets that the preprocessor replaces with their defined values before compilation.

```
#define PI 3.14159 // Object-like macro for a constant value
#define SQUARE(x) ((x) * (x)) // Function-like macro for squaring a number
```

Pre Processing: Example

```
#include <stdio.h> // Include standard input/output library
#define PI 3.14159 // Define PI as a constant macro
#define CIRCLE_AREA(r) (PI * (r) * (r)) // Define a function-like macro for circle area

int main() {
float radius;
float area;
printf("Enter the radius of the circle: ");
scanf("%f", &radius);

// Calculate area using the defined macro
area = CIRCLE_AREA(radius);
printf("The area of the circle is: %.2f\n", area);
return 0;
}
```

Output:

```
Enter the radius of the circle: 5
The area of the circle is: 78.54
```

Local and Global Variables

- Local variables only exist within a function. After leaving the function, they are ‘destroyed’. When the function is called again, they will be created (reassigned).
- Global variables can be accessed by any function within the program. While loading the program, memory locations are reserved for these variables and any function can access these variables for read and write (overwrite).
- If there exist a local variable and a global variable with the same name, the compiler will refer to the local variable.

Local Variables: Example

```
#include <stdio.h>

void displaySum() //function prototype and definition
{
   int a, b, sum;   // Local variables

   printf("Enter two numbers: \n");
   scanf("%d%d", &a, &b);

   sum = a + b;

   printf("Sum = %d\n", sum);
}
```

```
int main() {
    displaySum();   // Function call
 return 0;
}
```

```
Output:
Enter num1 and num2:
5
3
Sum is 8
```

Parul® University

DIGITAL LEARNING CONTENT

Global Variables: Example

```
#include <stdio.h>
void initialise(void);
void getInputs(void);
void calSum(void);
int sum, num1, num2; // Global Variables
void main(void)
{
        /* initialise sum to 0 */
        initialise( );
        /* read num1 and num2 */
        getInputs( );
        calSum( );
        printf("Sum is %d\n",sum);
}
```

```
void initialise(void)
{
    sum = 0;
}

void getInputs(void)
{
    printf("Enter num1 and num2:\n");
    scanf("%d%d",&num1,&num2);
}

void calSum(void)
{
    sum = num1 + num2;
}
```

Output:

```
Enter num1 and num2:
5
3
Sum is 8
```

Global Variables Explained

- In the previous example, no variables are passed between functions.
- Each function could have access to the global variables, hence having the right to read and write the value to it.
- Even though the use of global variables can simplify the code writing process (promising usage), it could also be dangerous at the same time.
- Since any function can have the right to overwrite the value in global variables, a function reading a value from a global variable can not be guaranteed about its validity.

Parul® University

DIGITAL LEARNING CONTENT

Global Variables: The Dangerous Side

```
#include <stdio.h>
void initialise(void);
void getInputs(void);
void calSum(void);
int sum, num1, num2; // Global Variables
void main(void)
{
        /* initialise sum to 0 */
        initialise( );
        /* read num1 and num2 */
        getInputs( );
        calSum( );
        printf("Sum is %d\n",sum);
}
```

```
void initialise(void)
{
    sum = 0;
}
void getInputs(void)
{
    printf("Enter num1 and num2:\n");
    scanf("%d%d",&num1,&num2);
}
void calSum(void)
{
    sum = num1 + num2;
    initialise( );

}
```

Output:
```
Enter num1 and num2:
5
3
Sum is 0
```

Imagine what would be the output of this program if someone 'accidently' write the following function call inside calSum?

Structures and Unions

- Structures
  - ➢ Introduction to Structures
  - ➢ Structure Definition
  - ➢ Declaring and Initializing Structure variables
  - ➢ Accessing Structure members
  - ➢ Copying and comparison of Structures
  - ➢ Array of Structures
  - ➢ Arrays within Structures
  - ➢ Structures within structures
  - ➢ Structures and functions
- Unions

Introduction to Structure

What is a Structure?

- A **structure** in C is a **user-defined data type** that allows you to **group different types of data** together under one name.
- It is used when you want to store information about a single entity that has multiple attributes of **different data types**.

Introduction to Structures:

- Heterogeneous data to be stored together e.g student’s result data
- Group logically related items to make the complex design more meaningful
- Structure and unions are two different data types which hold other data types in turn

Example Situation

Suppose you want to store information about a student:

- Roll number (integer)
- Name (string)
- Marks (float)

You could use a **structure** to group these together instead of using separate variables.

Structure Syntax (Definition)

```
struct structure_name {
    data_type member1;
    data_type member2;
    ...
    data_type memberN;
};
```

Declaring and initializing Structure Variables

You can declare variables in two ways:

1. After the structure definition:

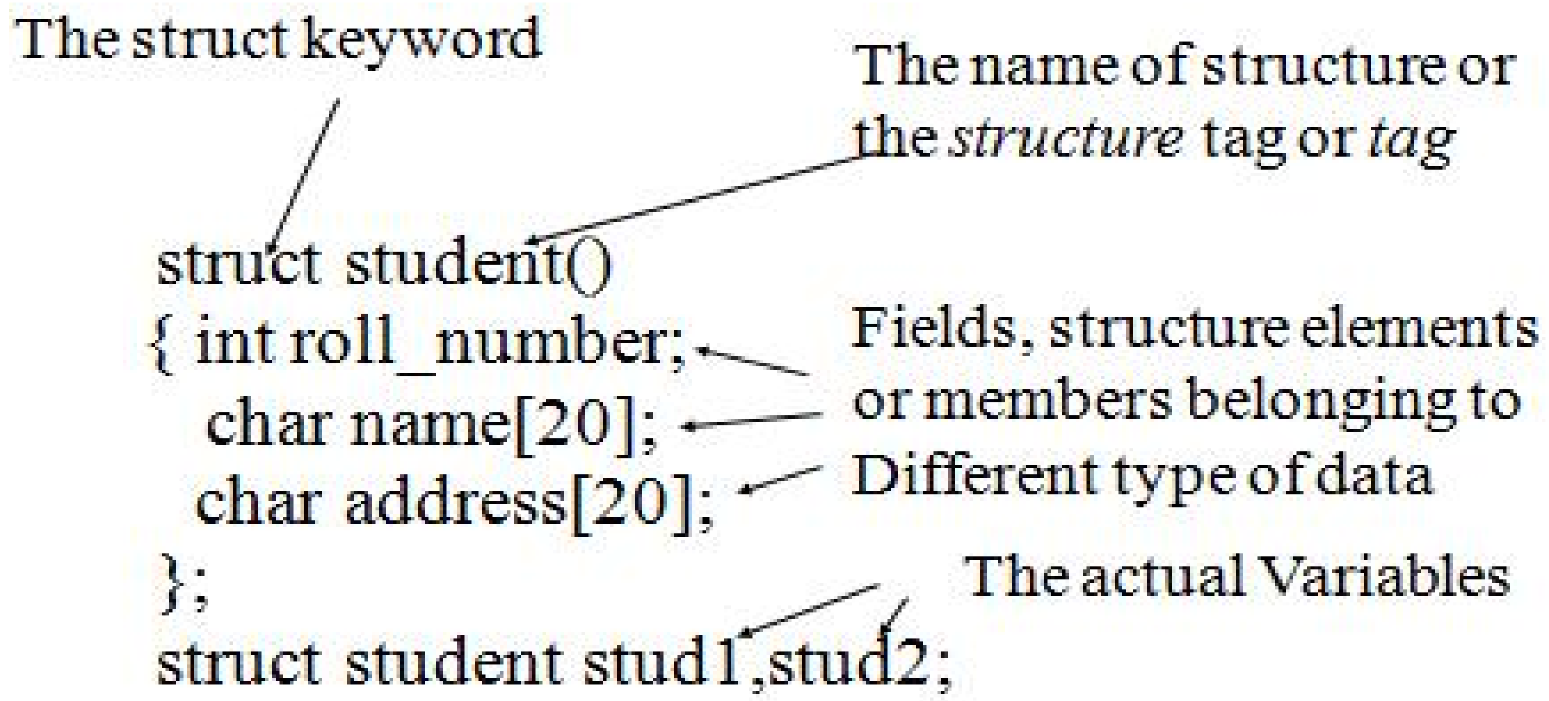
```
struct student s1, s2;
```

2. Together with the definition:

```
struct student {
    int roll_no;
    char name[20];
    float marks;
} s1, s2;
```

Structure Syntax (Definition and Declaration): Example

Structure Definition


The struct keyword
The name of structure or the *structure* tag or *tag*
Fields, structure elements or members belonging to Different type of data
The actual Variables

```
struct  student()
{ int roll_number;
   char name[20];
  char address[20];
};
struct  student stud1,stud2;
```

Accessing Structure members

Using the **dot (.) operator**:

```
s1.roll_no = 101;
strcpy(s1.name, "Alice");
s1.marks = 89.5;

printf("Name: %s\n", s1.name);
```

Example Program

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

//structure definition
struct student {
    int roll_no;
    char name[20];
    float marks;
};
```

```
int main() {
    struct student s1;

    printf("Enter roll number, name and marks:\n");
    scanf("%d %s %f", &s1.roll_no, s1.name, &s1.marks);

    printf("\nStudent Details:\n");
    printf("Roll No: %d\n", s1.roll_no);
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);

    return 0;
}
```

Output:

```
Enter roll number,
name and marks:
1 Anil 74

Student Details:
Roll No: 1
Name: Anil
Marks: 74.00
```

Copying and Comparison of Structures

- Two variables of the same structure type can be copied the same way as ordinary variables.
- If person1 and person 2 belong to same structure, then the following statement is valid.

```
Person1=person2;
Person2=person1;
```

Example: If person1 and person2 are structure variables of the same type, person1 = person2 will copy all the member values from person2 to person1.

Copying and Comparison of Structures: Example

```
#include <stdio.h>
#include <string.h>
struct Person {
    char name[50];
    int age;
};

int main(){
struct Person person1;
person1.age = 25;
strcpy(person1.name, "Alice");
printf("Person1 Age and Name: %s %d\n", person1.name,
person1.age);
struct Person person2;
person2 = person1; // Copies the values from person1 to
person2
printf("Person2 Age and Name: %s %d\n", person2.name,
person2.age);
}
```

What's Copied: This is a "shallow copy," meaning all the member values are copied directly from the source structure to the destination.

OutPut

```
Person1 Age and Name: Alice 25
Person2 Age and Name: Alice 25
```

Copying and Comparison of Structures: Example

```
#include <stdio.h>
#include <string.h>
struct Student {
    int id;
    char name[50];
    int score;
};
int main(){
struct Student student1 = {1, "Bob", 90};
struct Student student2 = {2, "Charlie", 85};
struct Student student3 = {1, "Bob", 90};
// To compare student1 and student3:
if (student1.id == student3.id &&
    strcmp(student1.name, student3.name) == 0 && //
For strings
    student1.score == student3.score) {
    printf("student1 and student3 are equal\n");
}
}
```

- Member-by-Member Comparison:
- Direct comparison of structure variables with == or != is not allowed in C.
- You must compare each corresponding member individually.

Example: To check if student1 and student2 are equal, you'd check:

student1.id == student2.id
student1.name == student2.name
student1.score == student2.score

Output: student1 and student3 are equal

Arrays of Structures

```
#include <stdio.h>

struct book {
   char name[50]; // Use a
character array for the book title
   float price;
   int pages;
};

int main() {
   struct book b[100];
   int i;
```

```
for(i=0; i<3; ++i) // Loop from 0 to 2 for 3 inputs
   {
      printf("\nEnter name, price and pages for book %d\n", i + 1);
      scanf(" %[^\n]s", b[i].name); // Use scanset to read a line with spaces
      scanf("%f", &b[i].price);
      scanf("%d", &b[i].pages);
      while (getchar() != '\n'); // Clear the input buffer
   }

printf("\nBook Details:\n");
   for(i=0; i<3; ++i) // Loop from 0 to 2 for 3 outputs
   {
      printf("Book %d: Name: %s, Price: %.2f, Pages: %d\n", i + 1,
b[i].name, b[i].price, b[i].pages);
   }
   return 0;
}
```

Arrays of Structures

Output:

```
Enter name, price and pages for book 1
Anil
20.50
10

Enter name, price and pages for book 2
Shail
41.00
20

Enter name, price and pages for book 3
Hinal
10.25
5

Book Details:
Book 1: Name: Anil, Price: 20.50, Pages: 10
Book 2: Name: Shail, Price: 41.00, Pages: 20
Book 3: Name: Hinal, Price: 10.25, Pages: 5
```

Arrays within Structures

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

// Define a structure named Student
struct Student {
   int roll_number;
   char name[50];
   int marks[3]; // An array of integers to store marks for 3 subjects
   float average;
};

int main() {
   // Declare a single variable of type struct Student
   struct Student s1;
   int total_marks = 0;

   // --- Assigning values to the structure members ---

   // Assign a roll number
   s1.roll_number = 101;

   // Use strcpy to assign a string to the character array 'name'
   strcpy(s1.name, "Alice");
// Assign values to the array of marks
   s1.marks[0] = 95;
   s1.marks[1] = 88;
   s1.marks[2] = 92;

   // --- Calculating and assigning the average ---
   for (int i = 0; i < 3; i++) {
      total_marks += s1.marks[i];
   }
   s1.average = (float)total_marks / 3;

   // --- Printing the data ---
   printf("--- Student Information ---\n");
   printf("Roll Number: %d\n", s1.roll_number);
   printf("Name: %s\n", s1.name);
   printf("Marks in 3 subjects: %d, %d, %d\n", s1.marks[0],
s1.marks[1], s1.marks[2]);
   printf("Average Marks: %.2f\n", s1.average);

   return 0;
}
```

Parul® University

DIGITAL LEARNING CONTENT

Arrays within Structures

Output:

```
--- Student Information ---
Roll Number: 101
Name: Alice
Marks in 3 subjects: 95, 88, 92
Average Marks: 91.67
```

Structure within Structures

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

// Define a structure named Student
struct Student {
    int roll_number;
    char name[50];
    int marks[3]; // An array of integers to store marks for 3 subjects
    float average;
};

int main() {
    // Declare a single variable of type struct Student
    struct Student s1;
    int total_marks = 0;

    // --- Assigning values to the structure members ---

    // Assign a roll number
    s1.roll_number = 101;

    // Use strcpy to assign a string to the character array 'name'
    strcpy(s1.name, "Alice");
// Assign values to the array of marks
    s1.marks[0] = 95;
    s1.marks[1] = 88;
    s1.marks[2] = 92;

    // --- Calculating and assigning the average ---
    for (int i = 0; i < 3; i++) {
        total_marks += s1.marks[i];
    }
    s1.average = (float)total_marks / 3;

    // --- Printing the data ---
    printf("--- Student Information ---\n");
    printf("Roll Number: %d\n", s1.roll_number);
    printf("Name: %s\n", s1.name);
    printf("Marks in 3 subjects: %d, %d, %d\n", s1.marks[0],
s1.marks[1], s1.marks[2]);
    printf("Average Marks: %.2f\n", s1.average);

    return 0;
}
```

Parul® University

Structure and functions

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

// 1. Define the `Student` structure
// This groups a student's ID, name, and
marks into a single data type.
struct Student {
    int id;
    char name[50];
    float marks;
};

// 2. A function that takes a struct by
value
// It receives a copy of the struct and
displays its contents.
void displayStudent(struct Student s) {
    printf("--- Student Details ---\n");
    printf("ID: %d\n", s.id);
    printf("Name: %s\n", s.name);
    printf("Marks: %.2f\n", s.marks);
}

// 3. A function that takes a struct by
reference (pointer)
// This is more memory-efficient for large
structs.
// It uses the arrow operator `->` to
access members.
void updateMarks(struct Student *s, float
new_marks) {
    s->marks = new_marks;
}

// 4. A function that returns a struct
// It creates and returns a new `Student`
struct.
struct Student createStudent(int id, const
char *name, float marks) {
    struct Student new_student;
    new_student.id = id;
    strcpy(new_student.name, name);
    new_student.marks = marks;
    return new_student;
}

int main() {
    // Call the function that returns a struct
    struct Student student1 = createStudent(101,
"Alice", 95.5);

    // Call the function that takes a struct by value
    displayStudent(student1);

    // Call the function that takes a struct by
reference to modify it
    printf("\nUpdating Alice's marks...\n");
    updateMarks(&student1, 98.0);

    // Display the updated struct
    displayStudent(student1);

    return 0;
}
```

Output:

```
--- Student Details ---
ID: 101
Name: Alice
Marks: 95.50

Updating Alice's
marks...
--- Student Details ---
ID: 101
Name: Alice
Marks: 98.00
```

Parul® University

DIGITAL LEARNING CONTENT

Structure and functions

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

// 1. Define the `Student`
structure
// This groups a student's
ID, name, and marks into a
single data type.
Structure Name: Student
struct Student {
    int id;
    char name[50];
    float marks;
};
```

```
// 2. A function that takes a struct by value
// It receives a copy of the struct and displays
its contents.
Function Name:  displayStudent
Parameter Passed: struct Student s (s is the
member of Structure Student)
Return type: void
void displayStudent(struct Student s) {
    printf("--- Student Details ---\n");
    printf("ID: %d\n", s.id);
    printf("Name: %s\n", s.name);
    printf("Marks: %.2f\n", s.marks);
}
```

Structure and functions

// 3. A function that takes a struct by reference (pointer)
// This is more memory-efficient for large structs.
// It uses the arrow operator `->` to access members.

Function Name: updateMarks

Parameter Passed: struct Student *s, float new_marks (*s is pointer structure student’s member s)

Return type: void

```
void updateMarks(struct Student *s, float
new_marks) {
    s->marks = new_marks;
}
```

// 4. A function that returns a struct
// It creates and returns a new `Student` struct.

Function Name: createStudent

Parameter Passed: int id, const char *name, float marks

Return type: struct Student

```
struct Student createStudent(int id, const char
*name, float marks) {
    struct Student new_student;
    new_student.id = id;
    strcpy(new_student.name, name);
    new_student.marks = marks;
    return new_student;
}
```

Parul® University

DIGITAL LEARNING CONTENT

Union

- A union in C is a user-defined data type that allows different data types to be stored in the same memory location.
- Unlike structures, where each member has its own distinct memory space, all members of a union share the same memory.
- The size of a union is determined by the size of its largest member.
- At any given time, only one member of a union can hold a meaningful value; assigning a value to one member overwrites the value of any previously assigned member.

Syntax of Union:

```
union UnionName {
dataType member1;
dataType member2;
// ... more members
};
```

Union: Example

```
#include <stdio.h>
#include <string.h> // Required for strcpy
// Define a union named Data
union Data {
int i;
float f;
char str[20]; // Character array to store a string
};

int main() {
// Declare a union variable union Data
myUnion;
union Data myUnion;
// Assign a value to the integer member
myUnion.i = 10;
printf("Integer value: %d\n", myUnion.i);

// Assign a value to the float member (this
overwrites the integer value)
myUnion.f = 220.5;
printf("Float value: %f\n", myUnion.f);
// Assign a value to the string member this overwrites the
float value)
strcpy(myUnion.str, "C Programming");
printf("String value: %s\n", myUnion.str);
// Attempting to access other members after assigning to 'str‘
// will yield undefined behavior as their memory has been
overwritten.
printf("Integer value after string assignment (undefined
behavior): %d\n",myUnion.i);;
printf("Float value after string assignment (undefined
behavior): %f\n", myUnion.f);

printf("Size of union Data: %lu bytes\n",
sizeof(union Data));

return 0;
}
```

```
Integer value: 10
Float value: 220.500000
String value: C Programming
Integer value after string assignment (undefined behavior):
1917853763
Float value after string assignment (undefined behavior):
4122360580327794860452759994368.000000
Size of union Data: 20 bytes
```

Difference Between Structure and Union

```
#include <stdio.h>
//defining a union
union job {
   char name [32];
   float salary;
   int worker_no;
}u;

struct job1 {
   char name [32];
   float salary;
   int worker_no;
}s;
```

```
int main(){
printf("size of union = %d",sizeof(u));
printf("\nsize of structure = %d", sizeof(s));
return 0;
}
```

Output:

```
size of union = 32
size of structure = 40
```

Difference Between Structure and Union

Structure	Union
1.The keyword **struct** is used to define a structure	1. The keyword union is used to define a union.
2. When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members. The smaller members may end with unused slack bytes.	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
3. Each member within a structure is assigned unique storage area of location.	3. Memory allocated is shared by individual members of union.
4. The address of each member will be in ascending order This indicates that memory for each member will start at different offset values.	4. The address is same for all the members of a union. This indicates that every member begins at the same offset value.
5 Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. Individual member can be accessed at a time	6. Only one member can be accessed at a time.
7. Several members of a structure can initialize at once.	7. Only the first member of a union can be initialized.

DIGITAL LEARNING CONTENT