

## **Module 5**

## Functions

A **function** is a block of code that performs a specific task.

Suppose, a program related to graphics needs to create a circle and color it depending upon the radius and color from the user.

You can create two functions to solve this problem:

- create a circle function
- color function

Dividing complex problem into small components makes program easy to understand and use.

## Types of functions in C programming

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming

There are two types of functions in C programming:

- Standard library functions
- User defined functions

## Standard library functions

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.

These functions are defined in the header file. When you include the header file, these functions are available for use.

For example:

The `printf()` is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "stdio.h" header file.

There are other numerous library functions defined under "stdio.h", such as `scanf()`, `fprintf()`, `getchar()` etc. Once you include "stdio.h" in your program, all these functions are available for use.

## User-defined functions

C allow programmers to define functions. Such functions created by the user are called user-defined functions.

Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want.

## Working of user-defined function

```
#include <stdio.h>
void functionName()
{
    ... ..
    ... ..
}
```

```
int main()
{
    ... ..
    ... ..

    functionName();

    ... ..
    ... ..
}
```

The execution of a C program begins from the main() function. When the compiler encounters functionName(); inside the main function, control of the program jumps to

### **void functionName()**

And, the compiler starts executing the codes inside the user-defined function.

The control of the program jumps to statement next to functionName(); once all the codes inside the function definition are executed.

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

```
void functionName()
```

```
{
```

```
    ... ..  
    ... ..
```

```
}
```

```
int main()
```

```
{
```

```
    ... ..  
    ... ..
```

```
    functionName();
```

```
}
```

```
    ... ..  
    ... ..
```

```
}
```

Remember, function name is an identifier and should be unique. This is just an overview on user-defined function. Visit these pages to learn more on:

- User-defined Function in C programming
- Types of user-defined Functions

### Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

### User-defined functions

Now learn to create user-defined functions in C programming in this article.

A function is a block of code that performs a specific task. C allows you to define functions according to your need. These functions are known as user-defined functions.

For example:

Suppose, you need to create a circle and color it depending upon the radius and color. You can create two functions to solve this problem:

- createCircle() function
- color() function

### **Example: User-defined function**

Here is a example to add two integers. To perform this task, a user-defined function addNumbers() is defined.

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

```
int addNumbers(int a, int b);    // function prototype
```

```
int main()
```

```
{
```

```
    int n1,n2,sum;
```

```
    printf("Enters two numbers: ");
```

```
    scanf("%d %d",&n1,&n2);
```

```
    sum = addNumbers(n1, n2);    // function call
```

```
    printf("sum = %d",sum);
```

```
    return 0;
```

```
}
```

```
int addNumbers(int a,int b)    // function definition
```

```
{
```

```
    int result;
```

```
    result = a+b;
```

```
    return result;    // return statement
```

```
}
```

## Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

## Syntax of function prototype

**returnType functionName(type1 arg1, type2 arg2,...);**

In the above example, `int addNumbers(int a, int b);` is the function prototype which provides following information to the compiler:

1. name of the function is `addNumbers()`
2. return type of the function is `int`
3. two arguments of type `int` are passed to the function

The function prototype is not needed if the user-defined function is defined before the `main()` function.

## Calling a function

Control of the program is transferred to the user-defined function by calling it.

## Syntax of function call

**functionName(arg1, arg2, ...);**

In the above example, function call is made using `addNumbers(n1,n2);` statement inside the `main()`.

## Function definition

Function definition contains the block of code to perform a specific task i.e. in this case, adding two numbers and returning it.

## Syntax of function definition

```
returnType functionName(type1 arg1, type2 arg2, ...)  
{  
    //body of the function  
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.


## Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables *n1* and *n2* are passed during function call.

The parameters *a* and *b* accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

### How to pass arguments to a function?

```
#include <stdio.h>  
  
int addNumbers(int a, int b);  
  
int main()  
{  
    ... ..  
    sum = addNumbers(n1, n2);  
    ... ..  
}  
  
int addNumbers(int a, int b)  
{  
    ... ..  
    ... ..  
}
```

Two arrows originate from the variables *n1* and *n2* in the function call `addNumbers(n1, n2)` within the `main` function. One arrow points from *n1* to the parameter *a* in the function definition `addNumbers(int a, int b)`. The other arrow points from *n2* to the parameter *b* in the same function definition.

The type of arguments passed to a function and the formal parameters must match, otherwise the compiler throws error.

If  $n1$  is of char type,  $a$  also should be of char type. If  $n2$  is of float type, variable  $b$  also should be of float type.

A function can also be called without passing an argument.

## Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after return statement.

In the above example, the value of variable *result* is returned to the variable *sum* in the main() function.

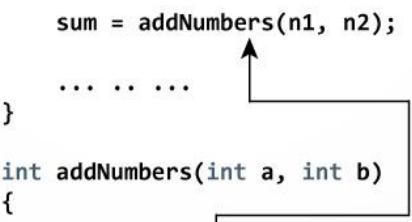
### Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

int addNumbers(int a, int b)
{
    ... ..
    return result;
}
```



sum = result



## Syntax of return statement

**return (expression);**

For example,  
return a;  
return (a+b);

The type of value returned from the function and the return type specified in function prototype and function definition must match.

## Types of User-defined Functions in C Programming

For better understanding of arguments and return value from the function, user-defined functions can be categorized as:

- Function with no arguments and no return value
- Function with no arguments and a return value
- Function with arguments and no return value
- Function with arguments and a return value.

The 4 programs below check whether an integer entered by the user is a prime number or not. And, all these programs generate the same output.

### Example #1: No arguments passed and no return Value

The checkPrimeNumber() function takes input from the user, checks whether it is a prime number or not and displays it on the screen.

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

```
void checkPrimeNumber();
```

```
int main()
```

```
{  
    checkPrimeNumber(); // no argument is passed to prime()  
    return 0;  
}
```

// return type of the function is void because no value is returned from the function

```
void checkPrimeNumber()
{
    int n, i, flag=0;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
        {
            flag = 1;
        }
    }
    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);
}
```

The empty parentheses in `checkPrimeNumber();` statement inside the `main()` function indicates that no argument is passed to the function.

The return type of the function is `void`. Hence, no value is returned from the function.

### **Example #2: No arguments passed but a return value**

The empty parentheses in `n = getInteger();` statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to *n*.

Here, the `getInteger()` function takes input from the user and returns it. The code to check whether a number is prime or not is inside the `main()` function.

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

```
int getInteger();
```

```
int main()
```

```
{
```

```
    int n, i, flag = 0;
```

```
    // no argument is passed to the function
```

```
    // the value returned from the function is assigned to n
```

```
    n = getInteger();
```

```
    for(i=2; i<=n/2; ++i)
```

```
    {
```

```
        if(n%i==0){
```

```
            flag = 1;
```

```
            break;
```

```
        }
```

```
    }
```

```
    if (flag == 1)
```

```
        printf("%d is not a prime number.", n);
```

```
    else
```

```
        printf("%d is a prime number.", n);
```

```
    return 0;
```

```
}
```

```
// getInteger() function returns integer entered by the user
```

```
int getInteger()
```

```
{
```

```
    int n;
```

```
    printf("Enter a positive integer: ");
```

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

```
    return n;
```

```
}
```

**Example #3: Argument passed but no return value**

The integer value entered by the user is passed to checkPrimeAndDisplay() function.

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

```
void checkPrimeAndDisplay(int n);
```

```
int main()
```

```
{
```

```
    int n;
```

```
    printf("Enter a positive integer: ");
```

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

```
    // n is passed to the function
```

```
    checkPrimeAndDisplay(n);
```

```
    return 0;
```

```
}
```

```
// void indicates that no value is returned from the function
```

```
void checkPrimeAndDisplay(int n)
```

```
{
```

```
    int i, flag = 0;
```

```
    for(i=2; i <= n/2; ++i)
```

```
    {
```

```
        if(n%i == 0){
```

```
            flag = 1;
```

```
            break;
```

```
        }
```

```
    }
```

```
    if(flag == 1)
```

```
        printf("%d is not a prime number.",n);
```

```
    else
```

```
        printf("%d is a prime number.", n);
```

```
}
```

Here, the checkPrimeAndDisplay() function checks whether the argument passed is a prime number or not and displays the appropriate message.

#### **Example #4: Argument passed and a return value**

The input from the user is passed to checkPrimeNumber() function.

```
#include <stdio.h>
int checkPrimeNumber(int n);

int main()
{
    int n, flag;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the checkPrimeNumber() function
    // the value returned from the function is assigned to flag
    variable
    flag = checkPrimeNumber(n);

    if(flag==1)
        printf("%d is not a prime number",n);
    else
        printf("%d is a prime number",n);

    return 0;
}

// integer is returned from the function
int checkPrimeNumber(int n)
{
    /* Integer value is returned from function
    checkPrimeNumber() */
    int i;
```

```

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
            return 1;
    }

    return 0;
}

```

The `checkPrimeNumber()` function checks whether the passed argument is prime or not. If the passed argument is a prime number, the function returns 0. If the passed argument is a non-prime number, the function returns 1. The return value is assigned to *flag* variable.

Then, the appropriate message is displayed from the `main()` function.

A function should perform a specific task. The `checkPrimeNumber()` function doesn't take input from the user nor it displays the appropriate message. It only checks whether a number is prime or not, which makes code modular, easy to understand and debug.

## Recursion

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

## Working of recursion

```

void recurse()
{
    ... ..
    recurse();
    ... ..
}

```

```

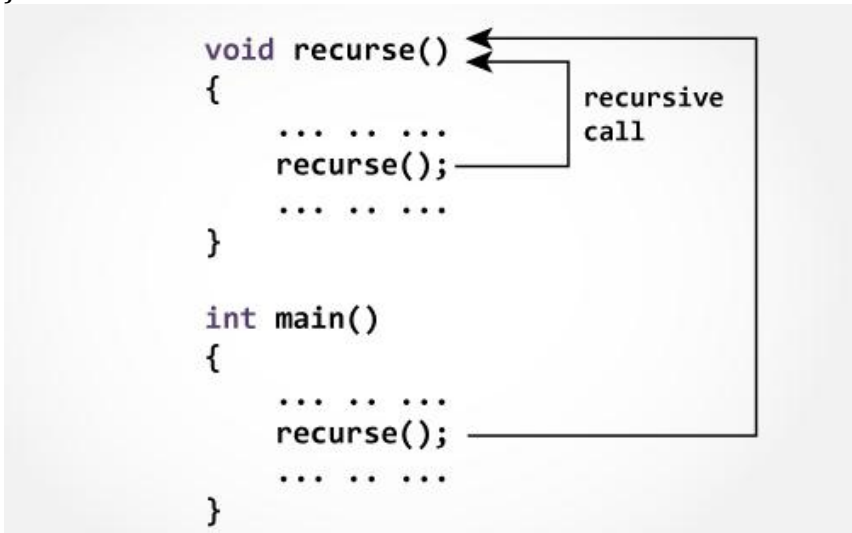
int main()
{

```

```

... ..
recurse();
... ..
}

```



The recursion continues until some condition is met to prevent it. To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

### Example: Sum of Natural Numbers Using Recursion

```

#include <stdio.h>
int sum(int n);

int main()
{
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);
}

```

```
    printf("sum=%d", result);  
}  
  
int sum(int num)  
{  
    if (num!=0)  
        return num + sum(num-1); // sum() function calls itself  
    else  
        return num;  
}
```

## Output

Enter a positive integer:

3

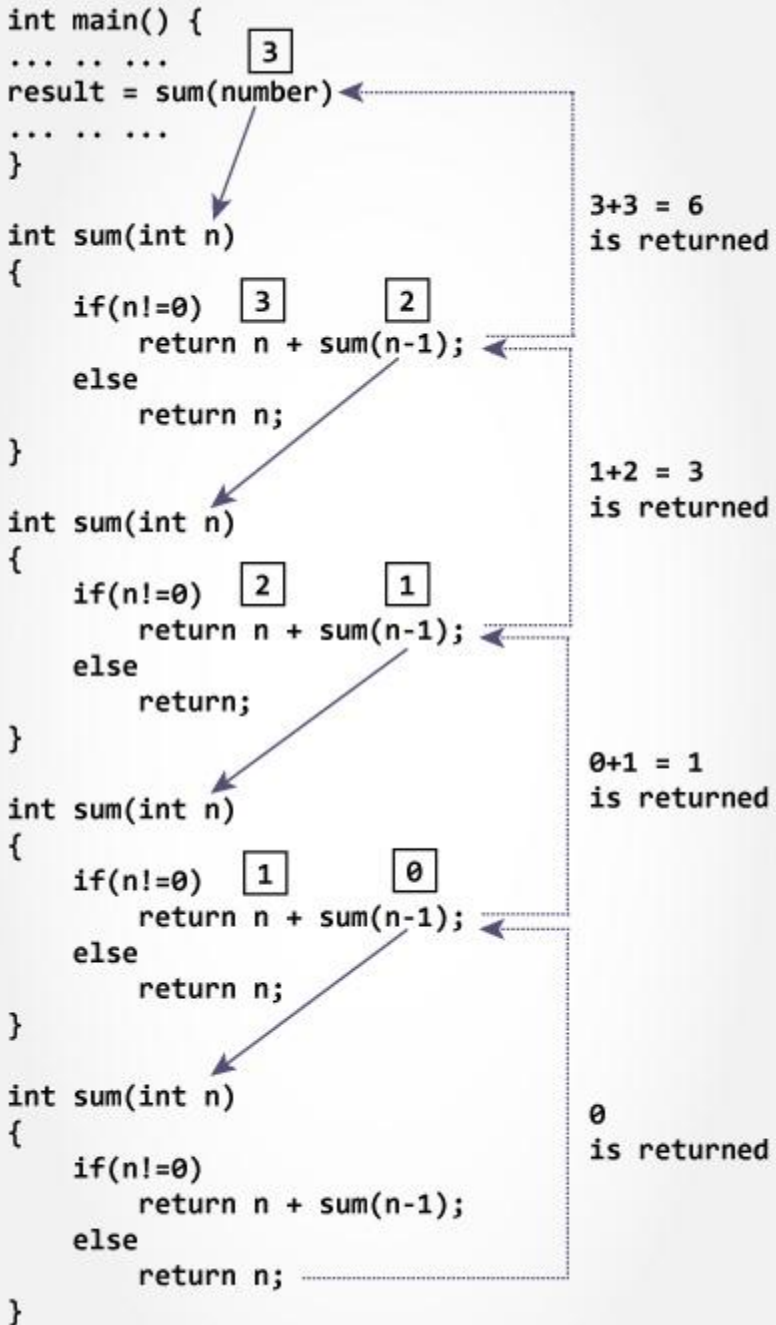
6

Initially, the `sum()` is called from the `main()` function with *number* passed as an argument.

Suppose, the value of *num* is 3 initially. During next function call, 2 is passed to the `sum()` function. This process continues until *num* is equal to 0.

When *num* is equal to 0, the if condition fails and the else part is executed returning the sum of integers to the `main()` function.





## Advantages and Disadvantages of Recursion

Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove.

If the speed of the program is vital then, you should avoid using recursion. Recursions use more memory and are generally slow. Instead, you can use loop.

## Scope and Lifetime of a variable

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And, storage class determines the scope and lifetime of a variable.

There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

## Local Variable

The variables declared inside the function are automatic or local variables.

The local variables exist only inside the function in which it is declared. When the function exits, the local variables are destroyed.

```
int main() {  
    int n; // n is a local variable to main() function  
    ... ..  
}
```

```
void func() {  
    int n1; // n1 is local to func() function  
}
```

In the above code, *n1* is destroyed when `func()` exits. Likewise, *n* gets destroyed when `main()` exits.

## Global Variable

Variables that are declared outside of all functions are known as external variables. External or global variables are accessible to any function.

### Example #1: External Variable

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

```
void display();
```

```
int n = 5; // global variable
```

```
int main()
```

```
{
```

```
    ++n; // variable n is not declared in the main() function
```

```
    display();
```

```
    return 0;
```

```
}
```

```
void display()
```

```
{
```

```
    ++n; // variable n is not declared in the display() function
```

```
    printf("n = %d", n);
```

```
}
```

## Output

```
n = 7
```

Suppose, a global variable is declared in file1. If you try to use that variable in a different file file2, the compiler will complain. To solve this problem, keyword `extern` is used in file2 to indicate that the external variable is declared in another file.

## Register Variable

The register keyword is used to declare register variables. Register variables were supposed to be faster than local variables.

However, modern compilers are very good at code optimization and there is a rare chance that using register variables will make your program faster.

Unless you are working on embedded system where you know how to optimize code for the given application, there is no use of register variables.

## Static Variable

A static variable is declared by using keyword static. For example;

```
static int i;
```

The value of a static variable persists until the end of the program.

### Example #2: Static Variable

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

```
void display();
```

```
int main()
```

```
{  
    display();  
    display();
```

```
}
```

```
void display()
```

```
{  
    static int c = 0;  
    printf("%d ",c);  
    c += 5;  
}
```

## Output

```
0 5
```

During the first function call, the value of  $c$  is equal to 0. Then, it's value is increased by 5.

During the second function call, variable  $c$  is not initialized to 0 again. It's because  $c$  is a static variable. So, 5 is displayed on the screen.

## Structure

Structure is a collection of variables of different types under a single name.

**For example:** You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables *name*, *citNo*, *salary* to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: *name1*, *citNo1*, *salary1*, *name2*, *citNo2*, *salary2*

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name *Person*, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name *Person* is a structure.

## Structure Definition in C

Keyword **struct** is used for creating a structure.

### Syntax of structure

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

**Note:** Don't forget the semicolon `;` in the ending line.

We can create the structure for a person as mentioned above as:

```
struct person
{
    char name[50];
    int citNo;
    float salary;
};
```

This declaration above creates the derived data type **struct person**.

### Structure variable declaration

When a structure is defined, it creates a user-defined type but, no storage or memory is allocated.

For the above structure of a person, variable can be declared as:

```
struct person
{
    char name[50];
    int citNo;
    float salary;
};
```

```
int main()
{
    struct person person1, person2, person3[20];
    return 0;
}
```

Another way of creating a structure variable is:

```
struct person
{
    char name[50];
    int citNo;
    float salary;
} person1, person2, person3[20];
```

In both cases, two variables *person1*, *person2* and an array *person3* having 20 elements of type **struct person** are created.

## Accessing members of a structure

There are two types of operators used for accessing members of a structure.

1. Member operator(.)
2. Structure pointer operator(->)

Any member of a structure can be accessed as:

**structure\_variable\_name.member\_name**

Suppose, we want to access salary for variable *person2*. Then, it can be accessed as:

**person2.salary**

## Example of structure

**Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet. (Note: 12 inches = 1 foot)**

```
#include <stdio.h>
struct Distance
{
    int feet;
    float inch;
} dist1, dist2, sum;

int main()
{
    printf("1st distance\n");

    // Input of feet for structure variable dist1
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);

    // Input of inch for structure variable dist1
    printf("Enter inch: ");
    scanf("%f", &dist1.inch);

    printf("2nd distance\n");
```



```
// Input of feet for structure variable dist2
printf("Enter feet: ");
scanf("%d", &dist2.feet);

// Input of feet for structure variable dist2
printf("Enter inch: ");
scanf("%f", &dist2.inch);

sum.feet = dist1.feet + dist2.feet;
sum.inch = dist1.inch + dist2.inch;

if (sum.inch > 12)
{
    //If inch is greater than 12, changing it to feet.
    ++sum.feet;
    sum.inch = sum.inch - 12;
}

// printing sum of distance dist1 and dist2
printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
return 0;
}
```

## Output

```
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
Sum of distances = 15'-5.7"
```

## Keyword typedef while using structure

Writing struct structure\_name variable\_name; to declare a structure variable isn't intuitive as to what it signifies, and takes some considerable amount of development time.

So, developers generally use typedef to name the structure as a whole. For example:

typedef struct complex

```
{  
    int imag;  
    float real;  
} comp;
```

int main()

```
{  
    comp comp1, comp2;  
}
```

Here, typedef keyword is used in creating a type *comp* (which is of type as **struct complex**).

Then, two structure variables *comp1* and *comp2* are created by this *comp* type.

### Structures within structures

Structures can be nested within other structures in C programming.

struct complex

```
{  
    int imag_value;  
    float real_value;  
};
```

struct number

```
{  
    struct complex comp;  
    int real;  
} num1, num2;
```

Suppose, you want to access *imag\_value* for *num2* structure variable then, following structure member is used.

num2.comp.imag\_value

## Passing structures to a function

There are mainly two ways to pass structures to a function:

1. Passing by value
2. Passing by reference

### Passing structure by value

A structure variable can be passed to the function as an argument as a normal variable.

If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

### C program to create a structure student, containing name and roll and display the information.

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

```
struct student
```

```
{
```

```
    char name[50];
```

```
    int roll;
```

```
};
```

```
void display(struct student stu);
```

```
// function prototype should be below to the structure  
declaration otherwise compiler shows error
```

```
int main()
```

```
{
```

```
    struct student stud;
```

```
    printf("Enter student's name: ");
```

```
    scanf("%s", &stud.name);
```

```
    printf("Enter roll number:");
```

```
    scanf("%d", &stud.roll);
```

```
    display(stud); // passing structure variable stud as  
argument
```

```
    return 0;
```

```
}
```

```
void display(struct student stu){  
    printf("Output\nName: %s",stu.name);  
    printf("\nRoll: %d",stu.roll);  
}
```

### Output

Enter student's name: Kevin Amla

Enter roll number: 149

Output

Name: Kevin Amla

Roll: 149

### Passing structure by reference

The memory address of a structure variable is passed to function while passing it by reference.

If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

### C program to add two distances (feet-inch system) and display the result without the return statement.

```
#include <stdio.h>  
struct distance  
{  
    int feet;  
    float inch;  
};  
void add(struct distance d1,struct distance d2, struct distance  
*d3);  
  
int main()  
{  
    struct distance dist1, dist2, dist3;  
  
    printf("First distance\n");  
    printf("Enter feet: ");  
    scanf("%d", &dist1.feet);
```

```
printf("Enter inch: ");
scanf("%f", &dist1.inch);

printf("Second distance\n");
printf("Enter feet: ");
scanf("%d", &dist2.feet);
printf("Enter inch: ");
scanf("%f", &dist2.inch);

add(dist1, dist2, &dist3);

//passing structure variables dist1 and dist2 by value
whereas passing structure variable dist3 by reference
printf("\nSum of distances = %d\'-%.1f'", dist3.feet,
dist3.inch);

return 0;
}
void add(struct distance d1, struct distance d2, struct distance
*d3)
{
    //Adding distances d1 and d2 and storing it in d3
    d3->feet = d1.feet + d2.feet;
    d3->inch = d1.inch + d2.inch;

    if (d3->inch >= 12) { /* if inch is greater or equal to 12,
converting it to feet. */
        d3->inch -= 12;
        ++d3->feet;
    }
}
```

## Output

```
First distance
Enter feet: 12
Enter inch: 6.8
Second distance
Enter feet: 5
```

Enter inch: 7.5

Sum of distances = 18'-2.3"

In this program, structure variables *dist1* and *dist2* are passed by value to the add function (because value of *dist1* and *dist2* does not need to be displayed in main function).

But, *dist3* is passed by reference ,i.e, address of *dist3* (&*dist3*) is passed as an argument.

Due to this, the structure pointer variable *d3* inside the add function points to the address of *dist3* from the calling main function. So, any change made to the *d3* variable is seen in *dist3* variable in main function.

As a result, the correct sum is displayed in the output.

## Structure and Pointer

Structures can be created and accessed using pointers. A pointer variable of a structure can be created as below:

```
struct name {
    member1;
    member2;
    .
    .
};
```

```
int main()
{
    struct name *ptr;
}
```

Here, the pointer variable of type **struct name** is created.

## Accessing structure's member through pointer

A structure's member can be accessed through pointer in two ways:

1. Referencing pointer to another address to access memory
2. Using dynamic memory allocation

1. Referencing pointer to another address to access the memory  
Consider an example to access structure's member through pointer.

```
#include <stdio.h>
typedef struct person
{
    int age;
    float weight;
};

int main()
{
    struct person *personPtr, person1;
    personPtr = &person1;    // Referencing pointer to memory
                              address of person1

    printf("Enter integer: ");
    scanf("%d",&(*personPtr).age);

    printf("Enter number: ");
    scanf("%f",&(*personPtr).weight);

    printf("Displaying: ");
    printf("%d%f",(*personPtr).age,(*personPtr).weight);

    return 0;
}
```

In this example, the pointer variable of type **struct person** is referenced to the address of *person1*. Then, only the structure member through pointer can be accessed.

**Using -> operator to access structure pointer member**

Structure pointer member can also be accessed using -> operator.

(\*personPtr).age is same as personPtr->age

(\*personPtr).weight is same as personPtr->weight

**2. Accessing structure member through pointer using dynamic memory allocation**

To access structure member using pointers, memory can be allocated dynamically using malloc() function defined under "stdlib.h" header file.

*Syntax to use malloc()*

ptr = (cast-type\*) malloc(byte-size)

**Example to use structure's member through pointer using malloc() function.**

```
#include <stdio.h>
#include <stdlib.h>
struct person {
    int age;
    float weight;
    char name[30];
};
```

```
int main()
{
    struct person *ptr;
    int i, num;
```

```
    printf("Enter number of persons: ");
    scanf("%d", &num);
```

```
    ptr = (struct person*) malloc(num * sizeof(struct person));
    // Above statement allocates the memory for n structures with
    pointer personPtr pointing to base address */
```

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



```

{
    printf("Enter name, age and weight of the person
    respectively:\n");
    scanf("%s%d%f", &(ptr+i)->name, &(ptr+i)->age, &(ptr+i)-
    >weight);
}

printf("Displaying Infromation:\n");
for(i = 0; i < num; ++i)
    printf("%s\t%d\t%.2f\n", (ptr+i)->name, (ptr+i)->age,
(ptr+i)->weight);

return 0;
}

```

### Output

Enter number of persons: 2

Enter name, age and weight of the person respectively:

Adam

2

3.2

Enter name, age and weight of the person respectively:

Eve

6

2.3

Displaying Information:

Adam	2	3.20
------	---	------

Eve	6	2.30
-----	---	------

### Pass structure to a function

In C, structure can be passed to functions by two methods:

1. Passing by value (passing actual value as argument)
2. Passing by reference (passing address of an argument)

### Passing structure by value

A structure variable can be passed to the function as an argument as a normal variable.

If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

**C program to create a structure student, containing name and roll and display the information.**

```
#include <stdio.h>
struct student
{
    char name[50];
    int roll;
};

void display(struct student stu);
// function prototype should be below to the structure
// declaration otherwise compiler shows error

int main()
{
    struct student stud;
    printf("Enter student's name: ");
    scanf("%s", &stud.name);
    printf("Enter roll number:");
    scanf("%d", &stud.roll);
    display(stud); // passing structure variable stud as argument
    return 0;
}

void display(struct student stu){
    printf("Output\nName: %s",stu.name);
    printf("\nRoll: %d",stu.roll);
}
```

## Output

Enter student's name: Kevin Amla

Enter roll number: 149

Output

Name: Kevin Amla

Roll: 149

## Passing structure by reference

The memory address of a structure variable is passed to function while passing it by reference.

If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

## C program to add two distances (feet-inch system) and display the result without the return statement.

```
#include <stdio.h>
struct distance
{
    int feet;
    float inch;
};
void add(struct distance d1, struct distance d2, struct distance
*d3);

int main()
{
    struct distance dist1, dist2, dist3;

    printf("First distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);
    printf("Enter inch: ");
    scanf("%f", &dist1.inch);

    printf("Second distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist2.feet);
    printf("Enter inch: ");
    scanf("%f", &dist2.inch);

    add(dist1, dist2, &dist3);
```

```
//passing structure variables dist1 and dist2 by value
whereas passing structure variable dist3 by reference
printf("\nSum of distances = %d\`-%.1f\`", dist3.feet,
dist3.inch);
```

```
    return 0;
}
void add(struct distance d1,struct distance d2, struct distance
*d3)
{
    //Adding distances d1 and d2 and storing it in d3
    d3->feet = d1.feet + d2.feet;
    d3->inch = d1.inch + d2.inch;

    if (d3->inch >= 12) {    /* if inch is greater or equal to 12,
converting it to feet. */
        d3->inch -= 12;
        ++d3->feet;
    }
}
```

## Output

First distance

Enter feet: 12

Enter inch: 6.8

Second distance

Enter feet: 5

Enter inch: 7.5

Sum of distances = 18'-2.3"

In this program, structure variables *dist1* and *dist2* are passed by value to the add function (because value of *dist1* and *dist2* does not need to be displayed in main function).

But, *dist3* is passed by reference ,i.e, address of *dist3* (&dist3) is passed as an argument.

Due to this, the structure pointer variable *d3* inside the add function points to the address of *dist3* from the calling main function. So, any change made to the *d3* variable is seen in *dist3* variable in main function.

As a result, the correct sum is displayed in the output.

## Unions

Unions are quite similar to structures in C. Like structures, unions are also derived types.

```
union car
{
    char name[50];
    int price;
};
```

Defining a union is as easy as replacing the keyword **struct** with the keyword **union**.

## Creating Union Variables

Union variables can be created in similar manner as structure variables.

```
union car
{
    char name[50];
    int price;
} car1, car2, *car3;
```

## OR

```
union car
{
    char name[50];
    int price;
};
```

```
int main()
{
    union car car1, car2, *car3;
    return 0;
}
```

In both cases, union variables *car1*, *car2* and union pointer variable *car3* of type **union car** is created.

### Accessing members of a union

Again, the member of unions can be accessed in similar manner as structures.

In the above example, suppose you want to access *price* for union variable *car1*, it can be accessed as:

**car1.price**

Likewise, if you want to access *price* for the union pointer variable *car3*, it can be accessed as:

**(\*car3).price**

**or;**

**car3->price**

### Difference between union and structure

Though unions are similar to structure in so many ways, the difference between them is crucial to understand.

The primary difference can be demonstrated by this example:

```
#include <stdio.h>
union unionJob
{
    //defining a union
    char name[32];
    float salary;
    int workerNo;
} uJob;
```

```
struct structJob
```

```

{
    char name[32];
    float salary;
    int workerNo;
} sJob;

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

```

### Output

size of union = 32

size of structure = 40

### More memory is allocated to structures than union

As seen in the above example, there is a difference in memory allocation between union and structure.

The amount of memory required to store a structure variable is the sum of memory size of all members.



Fig: Memory allocation in case of structure

But, the memory required to store a union variable is the memory required for the largest element of an union.



**Only one union member can be accessed at a time**

In the case of structure, all of its members can be accessed at any time.

But, in the case of union, only one of its members can be accessed at a time and all other members will contain garbage values.

```
#include <stdio.h>
union job
{
    char name[32];
    float salary;
    int workerNo;
} job1;

int main()
{
    printf("Enter name:\n");
    scanf("%s", &job1.name);

    printf("Enter salary: \n");
    scanf("%f", &job1.salary);

    printf("Displaying\nName :%s\n", job1.name);
    printf("Salary: %.1f", job1.salary);

    return 0;
}
```

**Output**

```
Enter name
Hillary
Enter salary
1234.23
Displaying
Name: f%Bary
Salary: 1234.2
```



**Note:** You may get different garbage value or empty string for the name.

Initially in the program, *Hillary* is stored in `job1.name` and all other members of *job1*, i.e. `salary`, `workerNo`, will contain garbage values.

But, when user enters the value of `salary`, 1234.23 will be stored in `job1.salary` and other members, i.e. `name`, `workerNo`, will now contain garbage values.

Thus in the output, *salary* is printed accurately but, *name* displays some random string.

### **Passing Union to a Function**

Union can be passed in similar manner as structures in C programming.

### **Dynamic Memory Allocation**

In C, the exact size of array is unknown until compile time, i.e., the time when a compiler compiles your code into a computer understandable language. So, sometimes the size of the array can be insufficient or more than required.

Dynamic memory allocation allows your program to obtain more memory space while running, or to release it if it's not required.

In simple terms, Dynamic memory allocation allows you to manually handle memory space for your program.

Although, C language inherently does not have any technique to allocate memory dynamically, there are 4 library functions under "`stdlib.h`" for dynamic memory allocation.

Function	Use of Function
<u>malloc()</u>	Allocates requested size of bytes and returns a pointer first byte of allocated space
<u>calloc()</u>	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
<u>free()</u>	deallocate the previously allocated space
<u>realloc()</u>	Change the size of previously allocated space

## malloc()

The name malloc stands for "**memory allocation**".

The function **malloc()** reserves a block of memory of specified size and return a **pointer** of type void which can be casted into pointer of any form.

### Syntax of malloc()

**ptr = (cast-type\*) malloc(byte-size)**

Here, **ptr** is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

**ptr = (int\*) malloc(100 \* sizeof(int));**

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

## calloc()

The name calloc stands for "**contiguous allocation**".

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

### Syntax of calloc()

**ptr = (cast-type\*)calloc(n, element-size);**

This statement will allocate contiguous space in memory for an array of **n** elements.

For example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

## **free()**

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

## **Syntax of free()**

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by **ptr**.

## **Example #1: Using C malloc() and free()**

**Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.**

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

```
#include <stdlib.h>
```

```
int main()
```

```
{
```

```
    int num, i, *ptr, sum = 0;
```

```
    printf("Enter number of elements: ");
```

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

```
    ptr = (int*) malloc(num * sizeof(int)); //memory allocated  
using malloc
```

```
    if(ptr == NULL)
```

```
{
```

```
    printf("Error! memory not allocated.");
```

```
    exit(0);
```

```
}

printf("Enter elements of array: ");
for(i = 0; i < num; ++i)
{
    scanf("%d", ptr + i);
    sum += *(ptr + i);
}

printf("Sum = %d", sum);
free(ptr);
return 0;
}
```

### **Example #2: Using C calloc() and free()**

**Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.**

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

int main()
{
    int num, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &num);

    ptr = (int*) calloc(num, sizeof(int));
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements of array: ");
    for(i = 0; i < num; ++i)
    {
```

```

        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}

```

## realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using `realloc()`.

### Syntax of realloc()

**`ptr = realloc(ptr, newsize);`**

Here, **ptr** is reallocated with size of `newsize`.

### Example #3: Using realloc()

```

#include <stdio.h>
#include <stdlib.h>

```

```

int main()
{
    int *ptr, i, n1, n2;
    printf("Enter size of array: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Address of previously allocated memory: ");
    for(i = 0; i < n1; ++i)
        printf("%u\t", ptr + i);

    printf("\nEnter new size of array: ");
    scanf("%d", &n2);
    ptr = realloc(ptr, n2);
}

```

```
    for(i = 0; i < n2; ++i)
        printf("%u\t", ptr + i);
    return 0;
}
```