

Year	Subject Title	Sem	Sub Code
2018 -19 Onwards	Core: COMPUTER FUNDAMENTALS AND C PROGRAMMING	I	18BIT13C

UNIT IV: User-Defined Functions: Introduction – Need and Elements of User-Defined Functions- Definition Return Values and their types - Function Calls – Function Declaration– Category of Functions- Nesting of Functions - Recursion – Passing Arrays and Strings to Functions - Structures and Unions.

TEXT BOOK

1. E Balagurusamy,” Computing Fundamentals & C Programming” - TataMcGraw-Hill, Second Reprint 2008, ISBN 978-0-07-066909-3.

Prepared by P.Sundari.

C - Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **strcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

The general form of a function definition in C programming language is as follows

```
return_type function_name( parameter list ) {
    body of the function
}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function –

- **Return Type** – A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** – The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two –

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts –

```
return_type function_name( parameter list );
```

For the above defined function max(), the function declaration is as follows –

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration –

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example –

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;

    /* calling a function to get max value */
```

```

ret = max(a, b);

printf( "Max value is : %d\n", ret );

return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

    /* local variable declaration */
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}

```

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result –

Max value is : 200

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function –

Sr.No.	Call Type & Description
1	<u>Call by value</u>

	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	<u>Call by reference</u> This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Function call by Value in C

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```

/* function definition to swap the values */
void swap(int x, int y) {

    int temp;

    temp = x; /* save the value of x */
    x = y;   /* put y into x */
    y = temp; /* put temp into y */

    return;
}

```

Now, let us call the function **swap()** by passing actual values as in the following example –

```

#include <stdio.h>

/* function declaration */
void swap(int x, int y);

```

```

int main () {

    /* local variable definition */
    int a = 100;
    int b = 200;

    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );

    /* calling a function to swap the values */
    swap(a, b);

    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );

    return 0;
}

```

Let us put the above code in a single C file, compile and execute it, it will produce the following result –

```

Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200

```

It shows that there are no changes in the values, though they had been changed inside the function.

Function call by reference in C

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

```

/* function definition to swap the values */
void swap(int *x, int *y) {

    int temp;
    temp = *x; /* save the value at address x */
    *x = *y; /* put y into x */
    *y = temp; /* put temp into y */

    return;
}

```

Let us now call the function **swap()** by passing values by reference as in the following example –

```

#include <stdio.h>

/* function declaration */
void swap(int *x, int *y);

int main () {

    /* local variable definition */
    int a = 100;
    int b = 200;

    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );

    /* calling a function to swap the values.
       * &a indicates pointer to a ie. address of variable a and
       * &b indicates pointer to b ie. address of variable b.
       */
    swap(&a, &b);

    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );

    return 0;
}

```

Let us put the above code in a single C file, compile and execute it, to produce the following result –

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :200

After swap, value of b :100

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {  
    recursion(); /* function calls itself */  
}  
  
int main() {  
    recursion();  
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Number Factorial

The following example calculates the factorial of a given number using a recursive function –

```
#include <stdio.h>  
  
unsigned long long int factorial(unsigned int i) {  
  
    if(i <= 1) {  
        return 1;  
    }  
    return i * factorial(i - 1);  
}
```

```
}  
  
int main() {  
    int i = 12;  
    printf("Factorial of %d is %d\n", i, factorial(i));  
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

Factorial of 12 is 479001600

Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function –

```
#include <stdio.h>  
  
int fibonacci(int i) {  
  
    if(i == 0) {  
        return 0;  
    }  
  
    if(i == 1) {  
        return 1;  
    }  
    return fibonacci(i-1) + fibonacci(i-2);  
}  
  
int main() {  
  
    int i;  
  
    for (i = 0; i < 10; i++) {  
        printf("%d\t", fibonacci(i));  
    }  
  
    return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

```
0
1
1
2
3
5
8
13
21
34
```

Structures

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.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book –

- Title
- Author
- Subject
- Book ID

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;  
    ...  
    member definition;  
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure –

```
struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} book;
```

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. The following example shows how to use a structure in a program –

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

struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

int main() {

    struct Books Book1;    /* Declare Book1 of type Book */
    struct Books Book2;    /* Declare Book2 of type Book */

    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;
```

```

/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

/* print Book2 info */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);

return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700

```

Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

```

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

struct Books {

```

```

char title[50];
char author[50];
char subject[100];
int book_id;
};

/* function declaration */
void printBook( struct Books book );

int main( ) {

    struct Books Book1;    /* Declare Book1 of type Book */
    struct Books Book2;    /* Declare Book2 of type Book */

    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;

    /* book 2 specification */
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Zara Ali");
    strcpy( Book2.subject, "Telecom Billing Tutorial");
    Book2.book_id = 6495700;

    /* print Book1 info */
    printBook( Book1 );

    /* Print Book2 info */
    printBook( Book2 );

    return 0;
}

void printBook( struct Books book ) {

    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
}

```

```
printf( "Book book_id : %d\n", book.book_id);  
}
```

When the above code is compiled and executed, it produces the following result –

```
Book title : C Programming  
Book author : Nuha Ali  
Book subject : C Programming Tutorial  
Book book_id : 6495407  
Book title : Telecom Billing  
Book author : Zara Ali  
Book subject : Telecom Billing Tutorial  
Book book_id : 6495700
```

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable –

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows –

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the → operator as follows –

```
struct_pointer->title;
```

Let us re-write the above example using structure pointer.

```
#include <stdio.h>  
#include <string.h>  
  
struct Books {  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
};  
  
/* function declaration */
```

```

void printBook( struct Books *book );
int main( ) {

    struct Books Book1;    /* Declare Book1 of type Book */
    struct Books Book2;    /* Declare Book2 of type Book */

    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;

    /* book 2 specification */
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Zara Ali");
    strcpy( Book2.subject, "Telecom Billing Tutorial");
    Book2.book_id = 6495700;

    /* print Book1 info by passing address of Book1 */
    printBook( &Book1 );

    /* print Book2 info by passing address of Book2 */
    printBook( &Book2 );

    return 0;
}

void printBook( struct Books *book ) {

    printf( "Book title : %s\n", book->title);
    printf( "Book author : %s\n", book->author);
    printf( "Book subject : %s\n", book->subject);
    printf( "Book book_id : %d\n", book->book_id);
}

```

When the above code is compiled and executed, it produces the following result –

```

Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407

```

Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples include –

- Packing several objects into a machine word. e.g. 1 bit flags can be compacted.
- Reading external file formats -- non-standard file formats could be read in, e.g., 9-bit integers.

C allows us to do this in a structure definition by putting :bit length after the variable. For example –

```
struct packed_struct {  
    unsigned int f1:1;  
    unsigned int f2:1;  
    unsigned int f3:1;  
    unsigned int f4:1;  
    unsigned int type:4;  
    unsigned int my_int:9;  
} pack;
```

Here, the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4-bit type and a 9-bit my_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case, then some compilers may allow memory overlap for the fields while others would store the next field in the next word.

Unions

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.

Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows –

```
union [union tag] {  
    member definition;  
    member definition;  
    ...  
    member definition;  
} [one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named `Data` having three members `i`, `f`, and `str` –

```
union Data {  
    int i;  
    float f;  
    char str[20];  
} data;
```

Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, `Data` type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union –

```
#include <stdio.h>  
#include <string.h>  
  
union Data {  
    int i;  
    float f;
```

```
char str[20];
};

int main() {

    union Data data;

    printf( "Memory size occupied by data : %d\n", sizeof(data));

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –
Memory size occupied by data : 20

Accessing Union Members

To access any member of a union, we use the **member access operator** (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program –

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

union Data {
    int i;
    float f;
    char str[20];
};

int main() {

    union Data data;

    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");
}
```

```
printf( "data.i : %d\n", data.i);
printf( "data.f : %f\n", data.f);
printf( "data.str : %s\n", data.str);

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions –

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

union Data {
    int i;
    float f;
    char str[20];
};

int main( ) {

    union Data data;

    data.i = 10;
    printf( "data.i : %d\n", data.i);

    data.f = 220.5;
    printf( "data.f : %f\n", data.f);

    strcpy( data.str, "C Programming");
    printf( "data.str : %s\n", data.str);
```

```
return 0;  
}
```

When the above code is compiled and executed, it produces the following result –

```
data.i : 10  
data.f : 220.500000  
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.