CP MODULE 4

Functions – function definition and function prototype. Function call by value and call by reference. Pointer to a function –. Recursive functions

FUNCTION

- A function is a **self-contained program segment** that carries out some specific, well-defined **task**.
- Function is used to **<u>*name* a group of statements.</u>**
- Every C program consists of one or more functions.
    - One of these functions must be called main().
- Execution of the C program will always start from main() function.
- Other user defined functions are subordinate of main()

- If a program contains many functions, their definitions can be written in any order.
- one *function definition cannot be embedded* within another function definition.
- A function can be called from another function.
    - When a function is accessed (i.e., whenever the function is "called") it will carry out its intended action (execute)
    - Once the function has carried out its intended action, control will be returned to the point from which the function was accessed(called).
- Information is passed to the function via special identifiers called arguments (also called ***parameters***), and result is returned via the ***return*** statement.

Advantages of function

1. Help to **break down program into different parts** or modules.
2. Function help to **reuse** the code.Function helps to **avoid redundant code**.
    a. The repeated instructions can be placed within a single function, which can then be accessed whenever it is needed.
3. Make program **simple**
    a. Improves *logical clarity* of the code
4. **easier to write and debug**,
5. Improves **portability**
    a. programs can be written that are independent of system-dependent features.

Elements of user defined function

There are three parts for a user defined functions:

1. Function declaration – also known as function prototype or function signature.
2. Function definition – implementation of function
3. Function call(use function)– known as function call or function invocation.(to execute the function)

FUNCTION DEFINITION

A function definition has two principal components:

- the first line (including the argument declarations),(**function header**)
- the **body of the function.**- compound statement enclosed inside curly brackets { and }

The first line of a function definitioncontains

- the data type of the value returned by the function,
- function name, and (optionally) a set of arguments, separated by commas and enclosed in parentheses.

The first line of function definition can be written as

data- type name(type1 arg1, type2 arg2, . . ., typen arg n)

where data - type represents the data type of the item that is returned by the function, name represents the function name, and type I,type 2, . . . , type n represent the data types of the arguments arg I , arg 2, . . , arg n.

(The data types are assumed to be of type int if they are not shown explicitly.)

- The arguments in function definition are called **formal arguments**
- The corresponding arguments in the function reference(call) are called **actual arguments.**
- The names of the formal arguments need not be the same as the names of the actual arguments in the calling portion of the program.(because they are local)

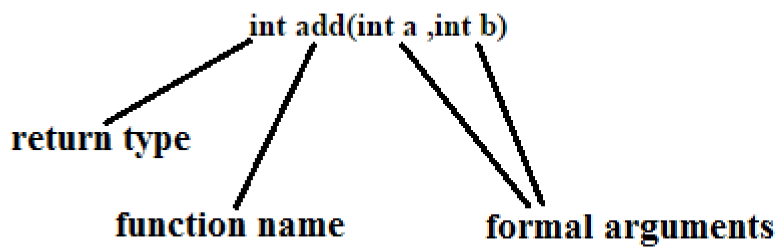
Body of the function

The remainder of the function enclosed within curly braces{ and } after function definition is a ***compound statement***. This compound statement is sometimes referred to as th**e body of the function.** The body of the function defines the action to be taken by the function

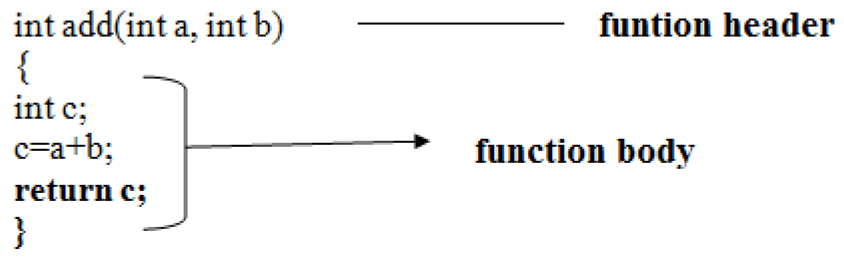
- compound statement can contain expression statements, other compound statements, control statements, and so on.
- It may include one or more return statements, in order to return a value to the calling portion of the program.
- Return statement is optional if function does not return value to the calling portion of the program
    - if no value is returned to calling portion, sometimes as the last statement in function body we write
      return;

Example function add with two integer arguments to find the sum. The function should return the sum of numbers.

Here first line(function header) of function definition can be written as



The return type is int because sum of two integers is integer. We have to return the sum and since the data type of return value (sum) is int so return type in function header should be int.
Function definition can be written as



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

return c statement means the value of c(sum of and b) is returned to the function calling this function(add).

E.g function to add two numbers

Function with **<u>no</u>** <u>return value</u> and <u>accept input from main function</u>	Function with <u>return value</u> and <u>accept input from main function</u>	Function with **<u>no</u>** <u>return value</u> and **<u>does not</u>** <u>accept input from main function</u>	Function with <u>return value</u> and **<u>does not</u>** <u>accept input from main function</u>

#include <stdio.h>
void add(inta ,int b)
{
int c;
c=a+b;
printf("sum=%d",c);
}
	#include <stdio.h>
	int add(inta ,int b)
	{
	int c;
	c=a+b;
	return c;
	}
		#include <stdio.h>
		void add()
		{
		int a,b,c;
		printf("Enter a");
		scanf("%d",&a);
		printf("Enter b");
		scanf("%d",&b);
		c=a+b;
		printf("sum=%d",c);
			#include <stdio.h>
			int add()
			{
			int a,b,c;
			printf("Enter a");
			scanf("%d",&a);
			printf("Enter b");
			scanf("%d",&b);
			c=a+b;
			return c;

main()
{
int x,y;
printf("Enter a");
scanf("%d",&x);
printf("Enter b");
scanf("%d",&y);
add(x,y);
}
	main()
	{
	int x,y,s;
	printf("Enter a");
	scanf("%d",&x);
	printf("Enter b");
	scanf("%d",&y);
	sum=add(x,y);
	printf("sum=%d",s);
	}
		}
		
		main()
		{
		add();
		}
			}
			
			main()
			{
			
			s=add();
			printf("sum=%d",s);
			}

In function header starts with void means that the function does not return a value to the calling function. If function header starts with int, it means that function returns an integer value.

FUNCTION CALL

- Just defining a function is not sufficient. In order to use(execute) the function, we need to access it. This is known as calling the function(function call).
- A function can be called by specifying its name, followed by the list of arguments in parentheses and separated by commas.
- If the function call does not require any arguments, an empty pair of parentheses must follow the name of the function to be called,
- If the function is defined before calling it, then function prototype(declaration) is not needed. Otherwise we have to write function prototype.
- If function returns a value that value may be stored when the function is called.

FUNCTION PROTOTYPE

- If function call comes before function definition , then function prototype is needed.
- When users develop their own functions, the compilers should be informed about these new functions. This is done through **function declaration(function prototype)**.
- Syntax of function prototype is almost same as function header in function definition, but this should terminate with a semicolon.
    - Argument variables are optional
    - Data type of arguments should be there

Function prototype contains the following elements.
→ function name
→ return type – data type of the result produced by the function
→ list of argument types – data types of the arguments in the function(arguments are optional but their data types should be specified)

Example

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

```
void twice(int a);        //function prototype ends with semicolon
                          //here function twice is called before the function twice is defined .
                          //so prototype is needed
main()
{
int a=5;
twice(a);        // call function named twice
}

void twice(int a)        //function definition of twice
{
int b=a*2;
printf("\nTwice a=%d",b);
}
```

Example 2

```
# include<stdio.h>
void sum(int , int);    //function prototype ends with semicolon
                //here function twice is called before the function twice is defined .
                //so prototype is needed/ /argument data type is needed. Argument name is
                optional
main()
{
int a=5,b=2;
sum(a,b);        // call function named twice
}

void sum(int a,int b) //function definition of twice
{
int c=a+b;
printf("\nSum=%d",c);
}
```

<u>Example </u> function with return value

In the following program add function is defined first, then only it is called. So function prototype is not needed fo add function.

```
#include <stdio.h>
int add(inta ,int b)       // function header in function definition. This returns an integer
{
int c;
c=a+b;
return c;
}

main()
{
int x,y,s;
printf(“Enter a”);
scanf(“%d”,&x);
printf(“Enter b”);
scanf(“%d”,&y);
sum=add(x,y);           //function call// x and y are actual arguments
// Since add function return the value it is stored in sum
printf(“sum=%d”,s);
}
```

Example function without return value

```
#include <stdio.h>
void add(int a, int b)  // function header // function does not
{
int c;
c=a+b;
printf(“sum=%d”,c);
}

main()
{
int x,y;
printf(“Enter x”);
scanf(“%d”,&x);
printf(“Enter y”);
scanf(“%d”,&y);

add(x,y);        // function call
}
```

FUNCTION CALL BY VALUE(Pass by value) AND CALL BY REFERENCE(Pass by reference).

When a single value is passed to a function via an actual argument, the value of the actual argument is copied into the function. Therefore, the value of the corresponding formal argument can be altered within the functions, but the value of the actual argument within the calling routine will not change.

This procedure for

passing the value of an argument to a function is known as passing by value.

CAll by value	**Call by reference**
Ihe value of the actual argument in function call is copied into the formal argument in function definition. So if formal argument changes, actual argument does not change	Action argument is a reference and formal argument is a pointer. So if formal argument is changed actual argument also changes
Thus, passing by value is restricted to a one-way transfer of information. When an argument is passed by value, the data item is copied to the function. Thus, any alteration made to the data item within the function is not carried over into the calling routine (	When an argument is passed by reference, however (i.e., when a pointer is passed to a function), the address of a data item is passed to the function. The contents of that address can be accessed freely, either within the function or within the calling routine. Moreover, any change that is made to the data item (i.e., to the contents of the address) will be recognized in both the function and the calling routine.
	Moreover, any change that is made to the data item (i.e., to the contents of the address) will be recognized in both the function and the calling routine.

E.g. Swap two numbers using call by value

```
#include<stdio.h>

void swap(int a, int b)
{
 int c;
 c=a;
 a=b;
 b=c;
 printf("\nInside function swap");
 printf("\na=%d b=%d ",a,b);
}
```

```
main()
{
int a=2,b=3;

printf("\nBefore calling swap");
printf("\na=%d b=%d ",a,b);
swap(a,b);
printf("\nAfter calling swap");
printf("\na=%d b=%d ",a,b);

}
```

E.g. **Swap two numbers using call by reference**

```
#include<stdio.h>

void swap(int *a, int *b)
{
        int c;
        c=*a;
        *a=*b;
        *b=c;
        printf("\nInside function swap");
        printf("\na=%d b=%d ",*a,*b);
}
main()
{
int a=2,b=3;

printf("\nBefore calling swap");
printf("\na=%d b=%d ",a,b);
swap(&a,&b);
printf("\nAfter calling swap");
printf("\na=%d b=%d ",a,b);

}
```

POINTER TO A FUNCTION
(PASSING FUNCTIONS TO OTHER FUNCTIONS)

- A pointer to one function can be passed to another function as an argument.
- This allows one function to be transferred to another, as though the first function were a variable.
- the first function can be called **guest function,** and the second function as the **host function**.
    - the guest is passed to the host,

- When a host function accepts the name of a guest function as an argument, the formal argument declaration must identify that ***argument*** as <u>*a pointer to the guest function*</u>.

<u>Formal argument declaration of host function</u>

- If guest function has ***no arguments***, the <u>*formal argument declaration of host function*</u> can be written as
  data- type (* function-name) ( )
- If guest function has arguments then <u>*formal argument declaration of host function*</u> can be written as
  data-type (* function-name) (type1 , type 2, . . .)
  or as
  data-type (* function-name) (type1 arg1 , type2 arg2, . . .)

  where ***data-type*** refers to the data type of the quantity returned by the guest and ***function-name*** is the name of the guest. ***type I , type 2, . . .*** refer to the data types of the arguments associated with the guest, and ***arg1 , arg 2***, . . . refer to the names of the arguments associated with the guest.

<u>**To call guest function inside host:**</u>

- The guest function can be accessed(called) within the host by means of the indirection operator.
- To do so, the indirection operator* must precede the guest function name (i.e., the formal argument).
- Both the <u>indirection operator and the guest function name must be enclosed in parentheses</u>;

(* function-name) (argument 1, argument 2, . . . , argument n) ;
where ***function-name*** is the name of the guest.,argument1, argument 2, . . . , argument n refer to the arguments that are required in the <u>**function call**</u>.

<u>**Function declaration for the host function**</u>

The function declaration for the host function can be written as

```
funct-data-type  funct-name(arg-data-type  (*)(type 1,  type 2, . . . ),
                            |←——        pointer to guest function        ——→|

                                          data types of other funct args);
```

When ***full function prototyping*** is used, the **<u>function declaration for the host function</u>** is expanded as follows.

```
funct-data-type  funct-name

           (arg-data-type  (*pt-var)(type 1  arg 1,  type 2  arg 2, . . . ),
           |←————          pointer to guest function              ————→|

                           data types and names of other funct args);
```

- where **funct-data- type** refers to the ***data type*** of the quantity *returned by the host function*;
- ***funct-name*** refers to the *name of the host function*;
- **arg-data- type** refers to the data *type of the quantity returned by the guest function,* and **type1, type 2** . . . refer to the *data types of guest function's arguments.*
- **pt-var** refers to the pointer variable pointing to the guest function,

Example Write a function twice with one integer argument a that returns twice the value of a. Write a host function callfun and pass a pointer to twice function and invoke it.

```
#include<stdio.h>

int twice(int a)
{
a=a*2;
return a;
 }

void callfun(int(*ptrf)(int a))            //host function with argument pointer to function
{
int a;
printf("\nEnter a\n");
scanf("%d",&a);
a= (*ptrf)(a);                             //calling guest function
printf("\ncallfunction a=%d",a);
}
```

```
main()
{

callfun(twice);     // calling host function callfun with argument guest function twice
}
```

Here callfun is the host function and twice is the guest function. ptrf is a pointer to guest function.
When callfun(twice) is executed ptrf will point to twice function

RECURSIVE FUNCTION

Recursion is a process by which a **function calls itself repeatedly**, until some specified condition has been satisfied.
To solve a problem recursively, two conditions must be satisfied.

- First, the problem must be written in a recursive form
- second, the problem statement must include a stopping condition.

n! = 1 x 2 x 3 x . + * x n, where n is the specified positive integer

We can also write
n! = n x (n - 1)!
0!=1
1!=1

- When a recursive program is executed, the recursive function calls are not executed immediately.
  - Rather, they are placed on a stack until the condition that terminates the recursion is encountered.
- The function calls are then executed in reverse order, as they are "popped" off the stack. Thus, when evaluating a factorial recursively, the function calls will proceed in the following order.

n! = n x (n - l)!
(n - l)! = (n - 1) x (n -2)!
(n -2)! = (n - 2) x (n - 3)!
2! =2 x l!
The actual values will then be returned in the following reverse order.
l! = 1
2! = 2 x l! = 2 x 1 = 2
3! = 3 x 2! = 3 x 2 =6
4! = 4 x 3! = 4 x 6 = 24
n!= n x (n- l)! = * - -
E.g
5!

5! Is put into stack
It is taken out

5*4! Is put into stack
4! Is taken out 4!=4*3! is put into stack
3! Is taken out 3!=3*2! is put into stack
2! Is taken out 2!=2*1! is put into stack
1! Is taken out 1!=1*0! is put into stack
0! Is taken out 0!=1 is put into stack
Then popped in reverse order
1*1*2*3*4*5=120

Example programs
Factorial,sumofdigit,nth fibonacci

EXAMPLE programs
Swap two variable using temporary variable(without function for swap)

```
#include<stdio.h>
main()
{
int a=2,b=3,c;
printf("\nBefore calling swap");
printf("\na=%d b=%d ",a,b);
c=a;
a=b;
b=c;

printf("\nAfter swap");
printf("\na=%d b=%d ",a,b);
}
```

Swap two variable without using temporary variable(without function for swap)

```
#include<stdio.h>
main()
{
int a=2,b=3;

printf("\nBefore calling swap");
printf("\na=%d b=%d ",a,b);
a=a+b;
b=a-b;
a=a-b;

printf("\nAfter swap");
printf("\na=%d b=%d ",a,b);


}
```