User defined function

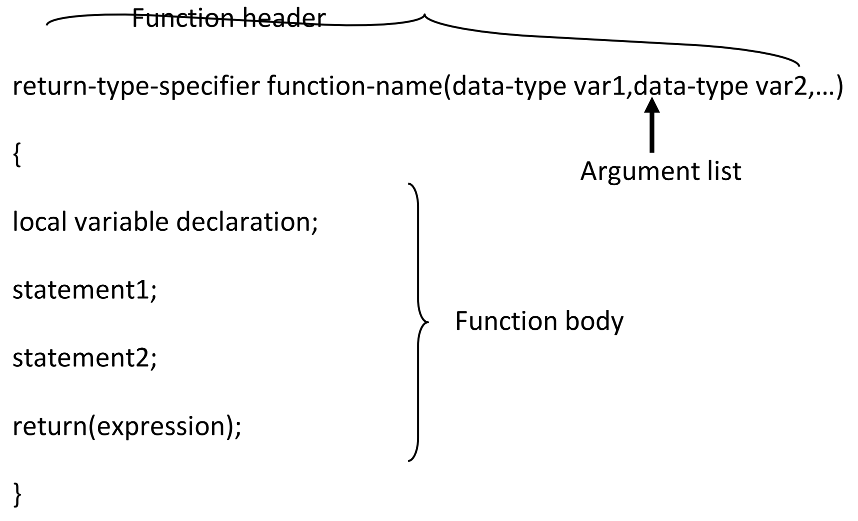
While solving a problem we use the top down approach. By this approach the complex problem is break down into smaller manageable part called module. The each module is solved separately. To manageable smaller sub program is called module or function

Function: It can be defined as a named unit of a group of program statements designed to perform a specific task and return a single value. If the function is defined by the user to do a specific task is called **user defined function.**

Advantages of user defined function

1. We can avoid the code repetition
2. **Universal use** : If a task is needed more than one program a function can develop and made available to other
3. Due to modularity, each function can develop independently and tested
4. **Team work** : One entire team will be involved to solve the problem instead of one person
5. The number of lines of code can be reduced

Structure of user defined function



```
return-type-specifier function-name(data-type var1,data-type var2,…)
{
local variable declaration;
statement1;
statement2;
return(expression);
}
```

Where

The return-type-specifier specifies the type of value that return back to the main or to the calling program.
Function-name is any valid identifier
data-type is a basic data type and is declared individually.
Local variables are the variable required in side of the function body

Statements are the task that we have to perform to solve the problem
return(expression) indicates the type of value that we are returning back to the calling function

Note : The first complete line is called function header .
The function header will not have a semicolon at the end

Example:

```
double power(float base, int expo) // function header

{  // beginning of function body

double result=1; // local varible

for(int i=0;i<expo;i++) // statement 1

result=result*base;  // statement 2

return(result);  // return expression

}  // end of function body
```

Note: The argument declared in the function header is called formal parameter and they accept the copy of original value

FUNCTION CALL

The function call is a statement used to invoke the sub program in the main and we can write this as follows
Variable-name= function-name(arguments);
OR
Variable-name= function-name();

The arguments used in the function call is called **actual argument**
For example to invoke above mentioned function we can write as
P=power(x, n); // the x and n are called the actual argument
The number and type of the argument used in the function call should match with the type and number of argument and order used in the function header and function prototype.
We can also declare a function with an *empty argument list*, as in the following example:
void display ( ) ;
In C++, this means that the function does not pass any parameters. It is identical to the statement.
void display (void) ;

Note: 1. Calling function is a function that transfer the control from it to another function by specifying the name of that function and passing arguments
2. Called function is a function that receives the call and arguments from another calling function
3. Function call is a statement that is used to call or make another function execute
4. When a function call is made the control jumps from calling function to the called function

The main() function: In C++ the main() function returns a value of type int to operating system. If the program runs successfully then the zero is returned otherwise non-zero is returned to the operating system indicating error.
If the main function is not return any value then the data type void is used.
The general form of main is

```
int main()                          void main()
{                   OR              {
statements;                         statements;
return 0;                           }
}
```

Returning a value: When a function is called the statements in the called function are executed. After the execution the function returns a value to the calling function. The return statement is used to return a value
Syntax:

return(expression); OR **return 0;**

Note:

1. The return statement is not compulsory
2. A function can have more than one return statement depending on the situation , but only one return statement is executed
   ```
   if(a>b)
   return(a);
   else
   return(b);
   ```
3. If a function declared as void do not return a value , so no need of using a return statement
4. The data type of the return value is same as that of return-type-specifier in the function header
5. If we avoid the return type then function assume the default return type to int
6. A function can return only one value or expression after evaluating
7. It is not possible to return more than one value in a single return statement

Function prototype:
The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a *template* is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself.

Function prototype is a declaration of the function that tells the program about the type of the value return by the function and the number and type of the arguments

Syntax:
return-type-specifier function-name(type , type ,….);
OR
return-type-specifier function-name(type arg1, type arg2,….);

Example:

```
float  volume( int  x,   float   y,  float  z) ;
```

Note that each argument variable must be declared independently inside the parentheses. That is, a combined declaration like

```
float  volume(int  x,  float  y,  z) ;
```

is illegal.

In a function declaration, the names of the arguments are *dummy* variables and therefore, they are optional. So we can also write the function prototype as

```
float  volume ( int,  float,  float ) ;
```

Note: The difference between the function proto type and function header is that the function prototype ends with semicolon where as the function header will not contain the semicolon

Types of argument: The arguments are the variable in function header or in function call. Depending on the place in which we are using it we categorize it as
a**) Actual argument**: These are the variable present in the function call is called actual argument or actual parameter
g=gcd(a,b); // The variable a and b is called as actual argument

b) **Formal argument**: The variable declared in the function header is called the formal parameter or formal argument
int gcd(int x, int y) // the x and y is called as formal argument

Note

1. The actual argument and formal argument should same in number and order
2. The name of the actual and formal argument can be same or different

Local variable: A variable declared inside a function or a block will have the scope only within side of that block is called local variable. These variable can not possible to access outside of that block .These variable will have the life only function created and destroy when we exit from function
Example

```
void main()
{
int x;
----
----

 }
```

Global variable: The all variable decaled out side of function and class can be accessed any where in the program . These variable will have the life till that program execution.
Example

```
int x;
void main()
{
int y; // local variable
----
----
}
```

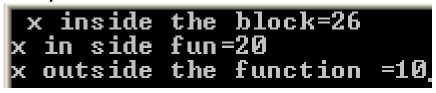
Scope of the variable: The scope of the variable refers to the part of the program where the values of variable can be used

Nested and parallel scope

The scope of the variable can be nested as shown in the following example

```
#include<iostream.h>

#include<conio.h>

void f();

void g();

int x=10;

void main()

{

clrscr();

int x=20;

{

int x=26;

cout<<" x inside the block="<<x<<endl;

}

cout<<"x in side fun="<<x<<endl;

cout<<"x outside the function ="<<::x;

getch();

}
```

Output



```
 x inside the block=26
x in side fun=20
x outside the function =10
```

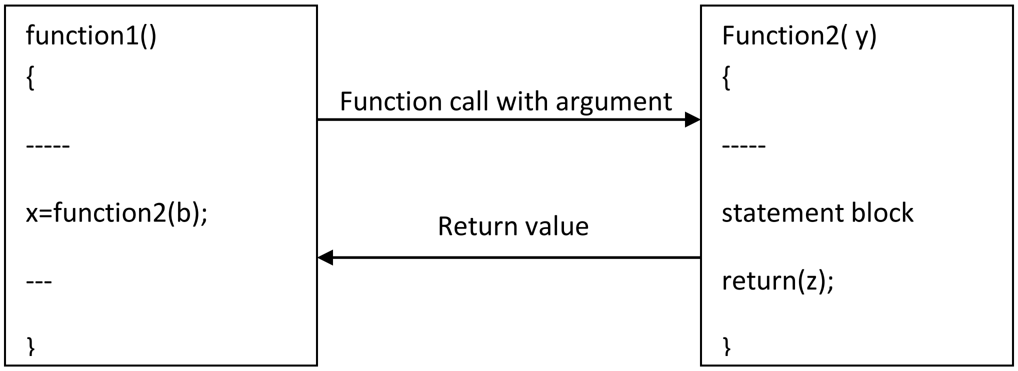
If we use the same variable name in entire program then we can access the outermost (global variable) by using **::** symbol

Types of function: The function can be categorized in to different type based on the whether we are going to pass the argument or after calculation function is returning any value back and they are

1. functions with argument and with return value
2. function with no argument and with return value
3. function with argument and no return value
4. functions with no argument and no return value
5. recursive function

The function with argument and with return value : In this method calling program pass the argument to sub program it accept it and after execution sub program return the value back to the calling program

Calling program called program

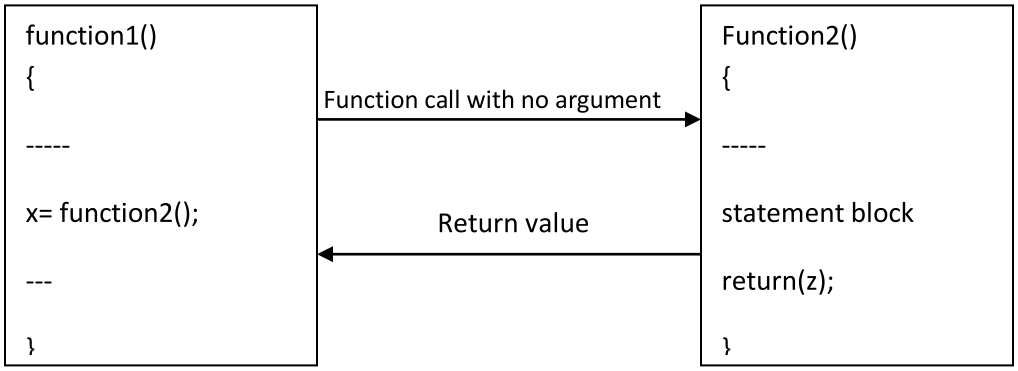


```
function1()
{
-----
x=function2(b);
---
}
```

```
Function2( y)
{
-----
statement block
return(z);
}
```

The function with no argument and with return value : In this method calling program will not pass the argument to sub program, it only call that sub program and after execution sub program return the value back to the calling program

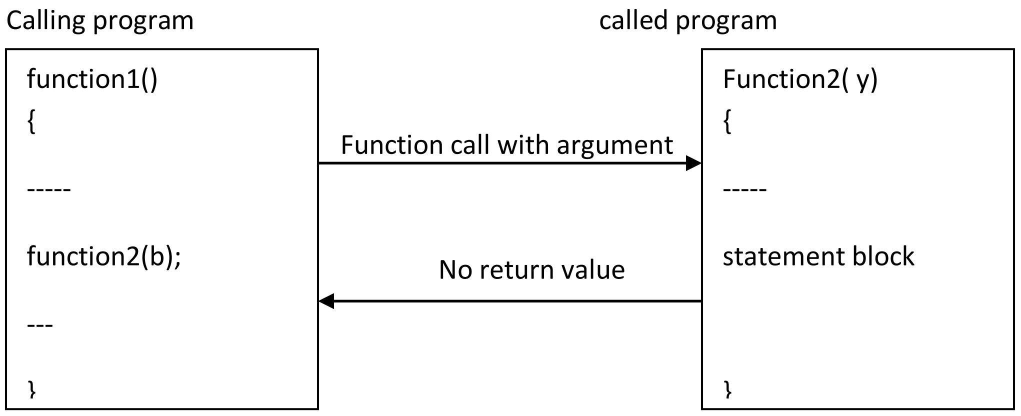
Calling program called program



```
function1()
{
-----
x= function2();
---
}
```

```
Function2()
{
-----
statement block
return(z);
}
```

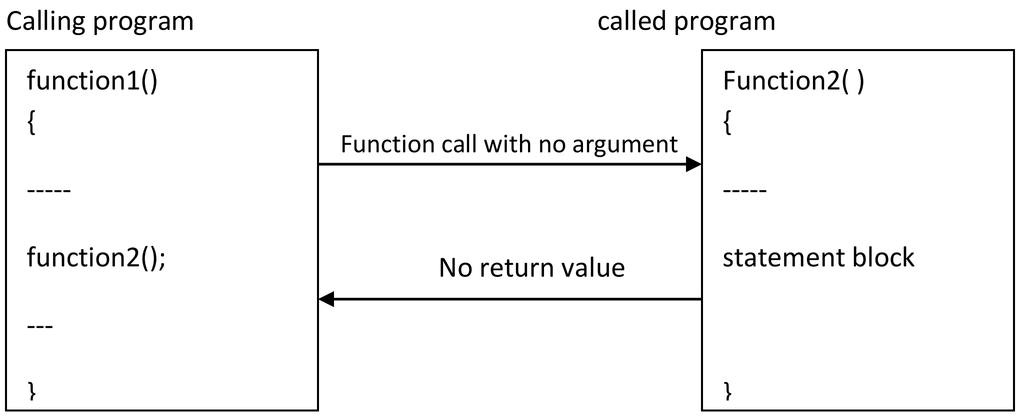
The function with argument and no return value : In this method calling program pass the argument to sub program it accept it and after execution sub program will not return the value back to the calling program



```
function1()
{
-----
function2(b);
---
}
```

```
Function2( y)
{
-----
statement block
}
```

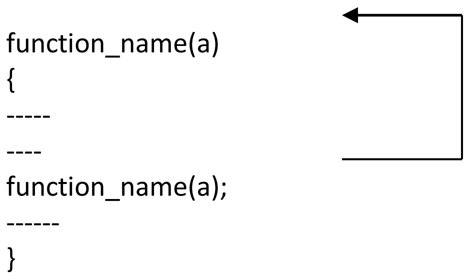
The function with no argument and with no return value : In this method calling program will not pass the argument to sub program   and after execution sub program is not return the value back to the calling program, it will return only control back



```
function1()
{
-----
function2();
---
}
```

```
Function2( )
{
-----
statement block
}
```

Recursive function :
Recursive is the technique where the function calls by itself again and again. In general the recursive function has two parts. The **Base case** which state the solution normally and call the recursive function. The second one is **recursive case** it contains the solution which is expressed in terms of a smaller version . The recursive function will have following syntax



```
function_name(a)
{
-----
----
function_name(a);
------
}
```

Passing default argument to function

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a *default value* to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype with default values:

```
float  amount ( float  principal , int  period, float  rate=0.15 ) ;
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument **rate**. A subsequent function call like

```
value = amount ( 5000 , 7 ) ;                  //  one  arguments  missing
```

Passes the value of 5000 to **principa**l and 7 to **period** and then lets function use default value of 0.15 for rate. The call

```
value  =  amount ( 5000 , 0, 0.12 ) ;                  //  no  missing  argument
```

passes an explicit value of 0.12 to **rate.**

A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have default values. It is important to note that we must add defaults from *right to left*. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int  mul ( int  I,  int  j=5,  int  k=10 ) ;                //  legal

int  mul ( int  i=5,  int  j ) ;                                      //  illegal

int  mul ( int  i=0,  int  j,  int  k=10 ) ;                //  illegal

int  mul ( int i=2,  int  j=5,  int  k=10 ) ;               //  legal
```

Advantages of providing the default arguments are:

1. We can use default arguments to add new parameters to the existing functions.
2. Default arguments can be used to combine similar functions into one.

Uses of default argument

1. These are useful in situation where some arguments always have the same value
2. It provides flexibility to the programmer
3. These are used to add new argument to the existing function
4. These are used to combine similar functions into one

Note: 1) The default argument can be used with inline function

2) Default values should be assigned only in the function prototype . it should not be repeated in the function definition

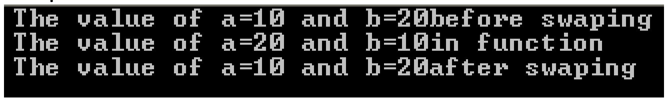
3) Default value for an argument can be global variable , global constant

4) Default values can be assigned to the arguments which does not have the matching argument in the function call

Call by value: In call by value method the parameter or argument that receives the copy of the values of the corresponding argument
The calling function sends the data to the called function through the actual parameters. The called function receives the data into its corresponding formal parameter. This is pass by value. In pass by value the copy of the data is sent by a calling function is stored in temporary storage location. The called function uses these value as the initial value of the formal parameter. If we do any changes to the formal parameter will not effect the original value stored in the actual parameter.
Example

```
#include <iostream.h>
#include <conio.h>
#include <string.h>

void main()
{
void swap(int,int);
int a=10,b=20;
clrscr();
cout<<" The value of a="<<a<< " and b="<<b<<"before swaping\n";
swap(a,b);
cout<<" The value of a="<<a<< " and b="<<b<<"after swaping\n";
getch();
}
void swap(int a, int b)
{
int t;
t=a;
a=b;
b=t;
cout<<" The value of a="<<a<< " and b="<<b<<"in function\n";
}
```

Output

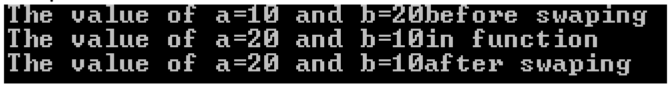


```
The value of a=10 and b=20before swaping
The value of a=20 and b=10in function
The value of a=10 and b=20after swaping
```

By reference: Some time we need to change the value of actual argument in the calling function then we have to pass the argument in the reference. Here we are passing the address of the actual argument to the formal argument instead of sending the copy of the value, so if we do any changes in the function will effect the original value
Example

```
#include <iostream.h>
#include <conio.h>
#include <string.h>

void main()
{
void swap(int&,int&);
int a=10,b=20;
clrscr();
cout<<" The value of a="<<a<< " and b="<<b<<"before swaping\n";
swap(a,b);
cout<<" The value of a="<<a<< " and b="<<b<<"after swaping\n";
getch();
}
void swap(int &a, int &b)
{
int t;
t=a;
a=b;
b=t;
cout<<" The value of a="<<a<< " and b="<<b<<"in function\n";
}
```

Output



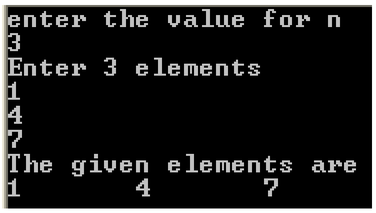
```
The value of a=10 and b=20before swaping
The value of a=20 and b=10in function
The value of a=20 and b=10after swaping
```

Passing array to the function

Normally we are passing the constant or a variable as an argument. The array is the block of the memory containing collection of elements and are stored in contiguous memory location, so if we want to pass the array values as a argument then we have to pass the address of the first location. That is if we want to pass the array then just pass the name of the array to the function as shown in the below example. Due to passing the address if we do any changes to the values in the function it will effect the original value also

```
#include <iostream.h>
#include <conio.h>
#include <string.h>

void main()
{
void display(int[],int);
int a[20],i,n;
clrscr();
cout<<"enter the value for n\n";
cin>>n;
cout<<"Enter "<<n<<" elements\n";
for(i=0;i<n;i++)
cin>>a[i];
cout<<"The given elements are \n";
display(a,n);
getch();
}
void display(int x[], int m)
{
int i;
for(i=0;i<m;i++)
cout<<x[i]<<"\t";
}
```

Output



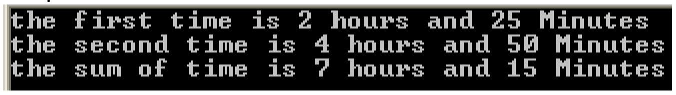
```
enter the value for n
3
Enter 3 elements
1
4
7
The given elements are
1       4       7
```

Passing structure to function : We can pass structure to function as we pass other argument. The structures are passed to the function by pass by value method

Example : To add two time periods

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
struct time
{
int h;
int m;
};
void main()
{
time sum(time,time);
void show(time);
time t1={2,25};
time t2= {4,50};
time t3;
clrscr();
cout<<"the first time is ";
show(t1);
cout<<"the second time is ";
show(t2);
t3=sum(t1,t2);
cout<<"the sum of time is ";
show(t3);
getch();
}
time sum(time t1, time t2)
{
time total;
total.h=(t1.h+t2.h)+(t1.m+t2.m)/60;
total.m=(t1.m+t2.m)%60;
return(total);
}
void show(time t)
{
cout<<t.h<<" hours and "<<t.m<<" Minutes\n";
}
```

Output



```
the first time is 2 hours and 25 Minutes
the second time is 4 hours and 50 Minutes
the sum of time is 7 hours and 15 Minutes
```