Functions and Strings

By
Dr. Jyoti Ranjan Nayak
Asst. Prof.
Institute of Technical Education and Research
(Siksha ‘O’ Anusandhan University)

Functions

Introduction Function:

- ➢ A function is a self contained block of code that performs a certain task/job.
- ➢ The programming statements of a function are enclosed within { } braces, having certain meanings and performing certain operations.
- ➢ A function can be called multiple times to provide reusability and modularity to the C++ program. In other words, we can say that the collection of functions creates a program.

Advantage of functions in C++

i. By using functions, we can avoid rewriting same logic/code again and again in a program.

ii. We can call C++ functions any number of times in a program and from any place in a program.

iii. We can track a large C++ program easily when it is divided into multiple functions.

iv. Reusability is the main achievement of C++ functions.

v. However, Function calling is always a overhead in a C++ program.

Types of function

There are two types of function in C++ programming:

1. Standard library functions
2. User-defined functions

1. Standard library functions

The standard library functions are built-in functions in C++ programming.

These functions are defined in header files. For example,

- ➢ The cout() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the iostream header file.
- ➢ The sqrt() function calculates the square root of a number. The function is defined in the math.h header file.

2. User-defined functions

- A function is a block of code that can be used to perform a specific action. C++ allows users to create their own functions called user-defined functions.
- A user-defined function can perform specific actions defined by users based on given inputs and deliver the required output.
- In User-defined functions, the user can give any name to the functions except the name of keywords.
- It follows three main parts:
  - A. function declaration
  - B. function definition
  - C. function call

A. Function declaration/ Function prototype:

- A function must be declared globally in a cpp program to tell the compiler about the function name, function parameters, and return type.
- A function declaration gives information to the compiler that the function may later be used in the program.

Syntax:

return_type function_name (type1 arg1, type2 arg2, ... typeN argN);

or

return_type function_name (type1 , type2 , ... typeN);

Example:

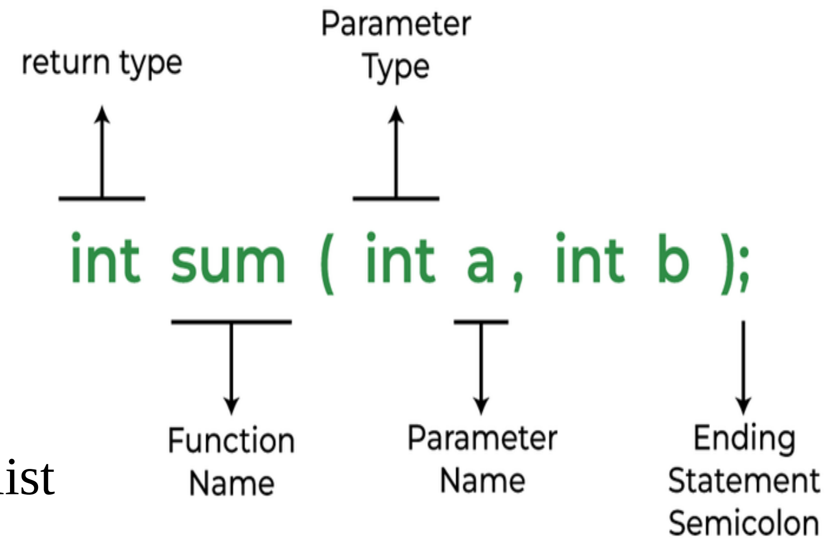
add(); //Return type and parameter list are not present

int add(int, int); //int is the return type and int, int is the parameter-type list

int add(int, float);* //int* is the return type and int, float is the parameter-type list

int add(int a,int b); //Parameter list contains the names of parameters, i.e. a and b

int add(int, int b); //Combination of abstract and complete parameter declaration



B. Function definition:

- Once the function has been called, the function definition contains the actual statements that will be executed. All the statements of the function definition are enclosed within { } braces.
- 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.
- If the function call is present after the function definition, we can skip the function prototype part and directly define the function.

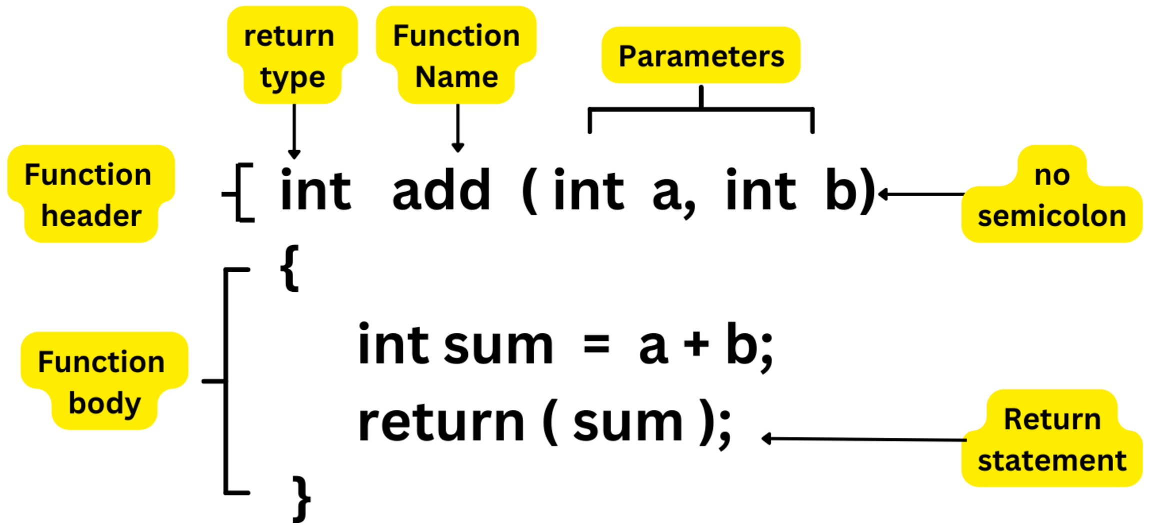
Syntax:

return_type function_name(parameter list)

{

body of the function

}



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

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.

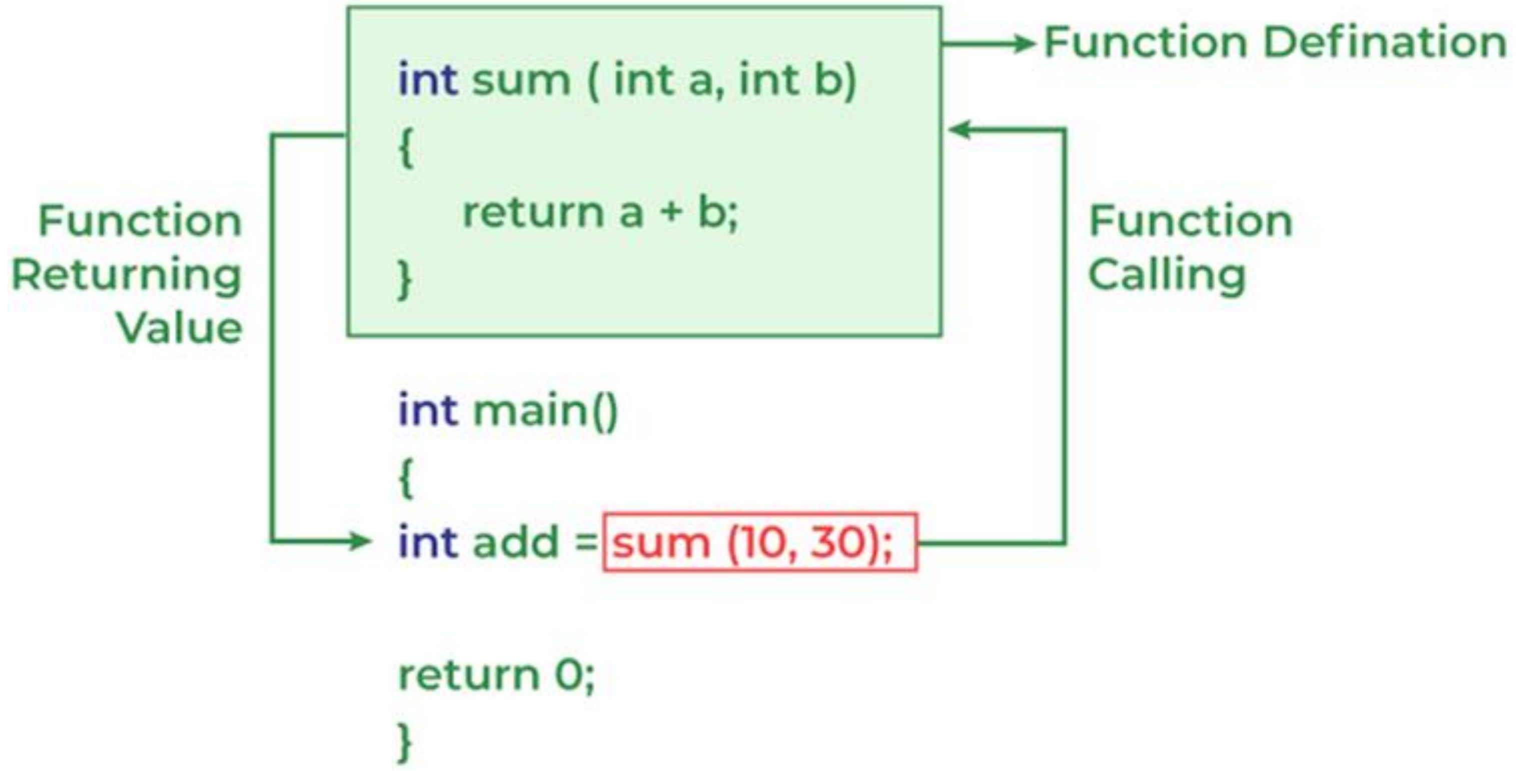
C. Function call:

- In order to transfer control to a user-defined function, we need to call it. Functions are called using their names followed by round brackets. Their arguments are passed inside the brackets.

Syntax:

function_name(arg1, arg2, ... argN);

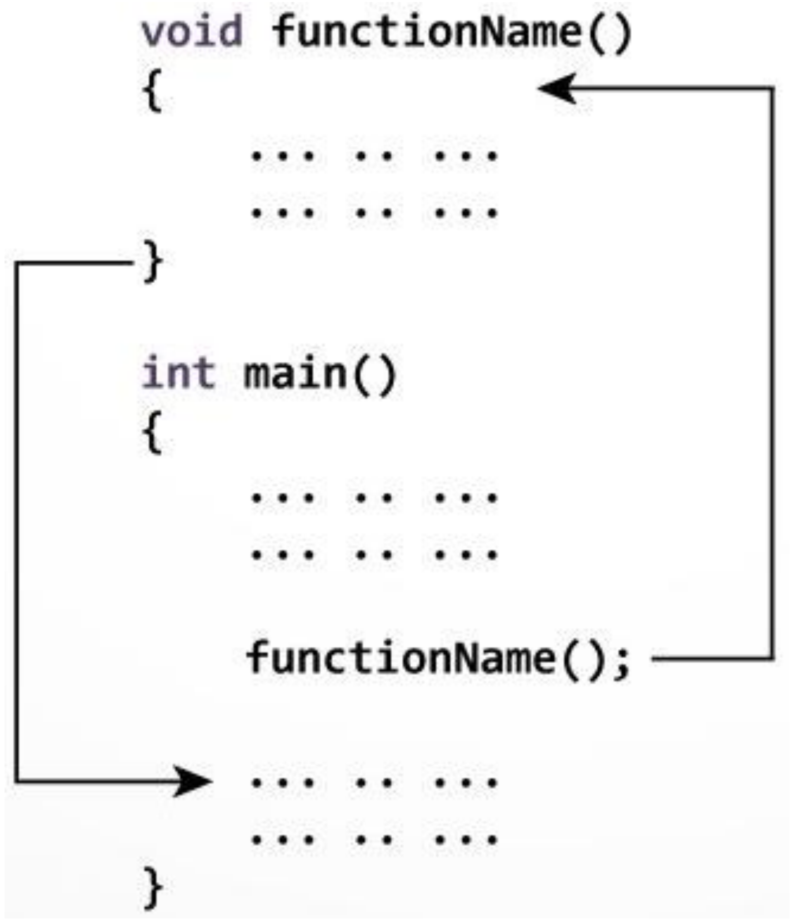
Example:



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

int main()
{
int add = sum (10, 30);

return 0;
}
```

How user-defined function works?

- The execution of a C++ program begins from the *main()* function.
- When the compiler encounters *functionName();*, control of the program jumps to *void functionName().*
- *And, the compiler starts executing the codes inside functionName().*
- *The control of the program jumps back to the main() function once code inside the function definition is executed.*



```
void functionName()
{
    ... .. ...
    ... .. ...
}

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

    functionName();

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

Example of function

```
#include<iostream>
using namespace std;
int sum(int a, int b)
{
    int c=a+b;
    return c;
}
int main()
{
    int result=sum(15,24);
    cout<<result;
    return 0;
}
```

Output: *39*

```
#include<iostream>
using namespace std;
int sum()
{
    int a, b;
    cout<<"Enter two numbers:";
    cin>>a;
    cin>>b;
    return a+b;
}
int main()
{
    int result=sum();
    cout<<result;
    return 0;
}
```

Output:

```
Enter two numbers:15
21
36
```

Categories of Functions:

A function depending an whether the arguments are present or not and whether a value is returned or not, may belong to one of following categories

i. Function with no return values, no arguments

ii. Functions with arguments, no return values

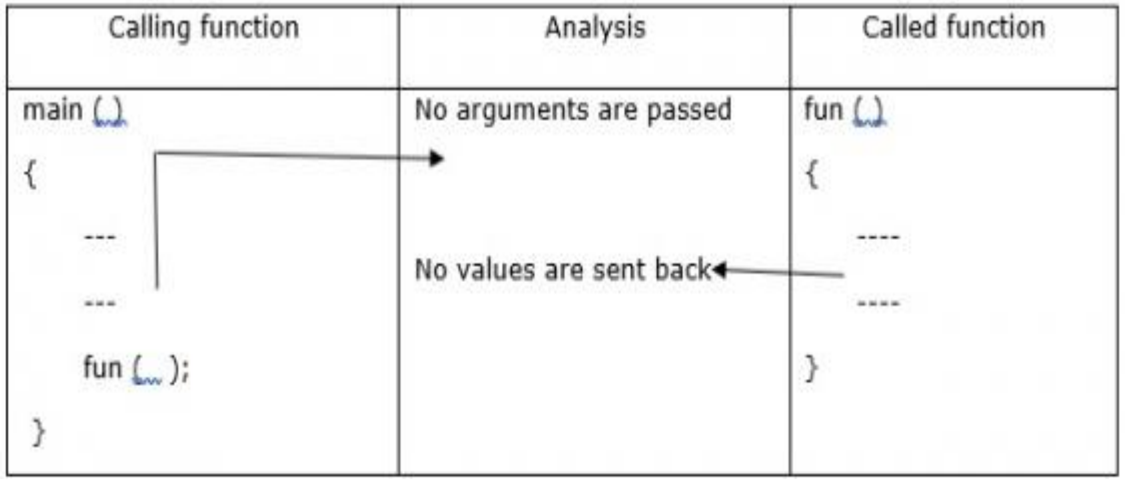
iii. Functions with arguments and return values

iv. Functions with no arguments and return values.

i. Function with no return values, no arguments

Functions that have no arguments and no return values. Such functions can either be used to display information or to perform any task on global variables.

```
#include<iostream>
using namespace std;
void sum()
{
   int a, b;
   cout<<"Enter two numbers:";
   cin>>a;
   cin>>b;
   cout<<(a+b);
}
int main()
{
   sum();
   return 0;
}
```



Calling function	Analysis	Called function

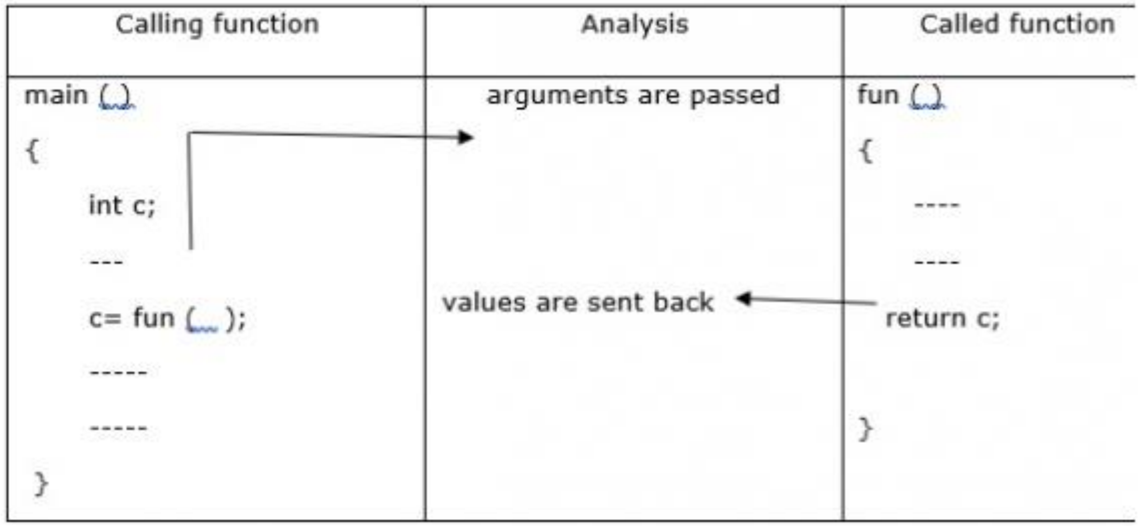
main ( )
{
--- 	
--- 	
fun ( );
}
	No arguments are passed
	
	No values are sent back
		fun ( )
		{
		---- 	
		---- 	
		}

Output:

```
Enter two numbers:10
20
30
```

ii. Functions with arguments, no return values

Functions that have arguments but no return values. Such functions are used to display or perform some operations on given arguments.



```
#include<iostream>
using namespace std;
void sum(int a, int b)
{
   cout<<(a+b);
}
int main()
{
   sum(10,20);
   return 0;
}
```

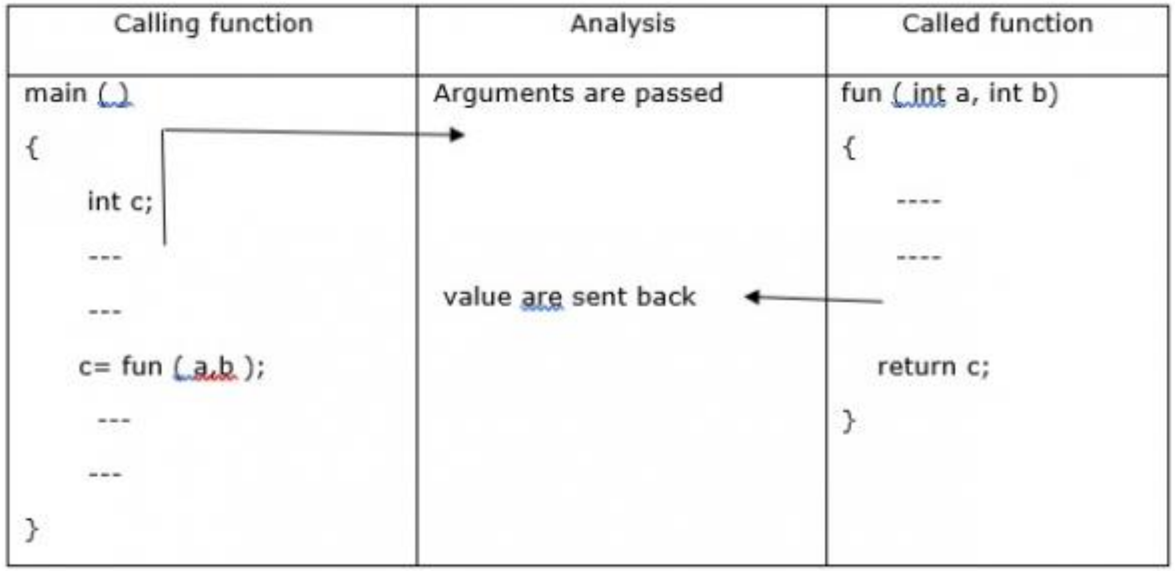
Output: 30

iii. Functions with arguments and return values

Functions that have arguments and some return value. These functions are used to perform specific operations on the given arguments and return their values to the user.

```
#include<iostream>
using namespace std;
int sum(int a, int b)
{
   int c=a+b;
   return c;
}
int main()
{
   int result=sum(15,24);
   cout<<result;
   return 0;
}
```

Output: 39

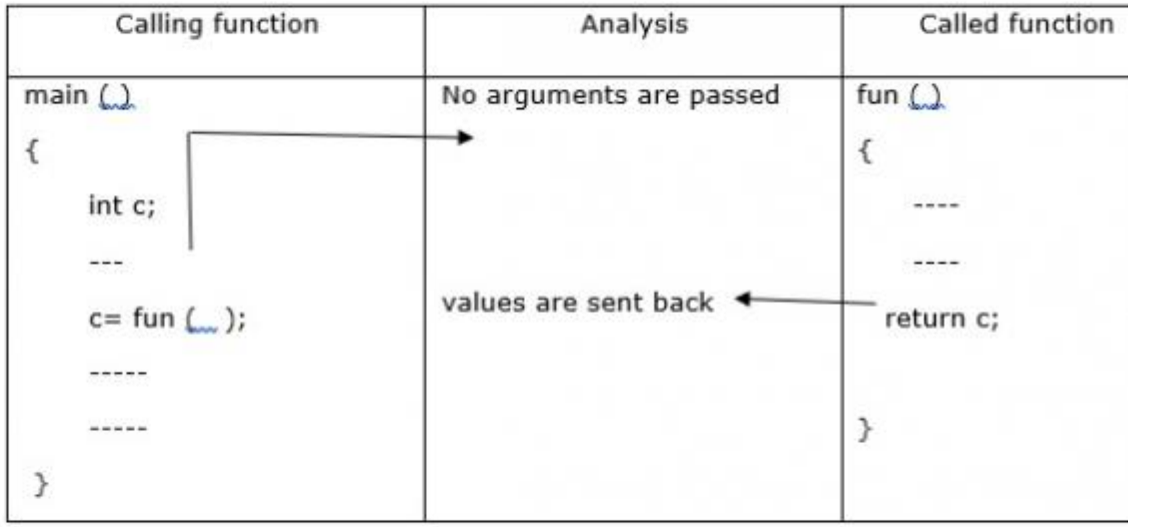


Calling function	Analysis	Called function

main ( )
{
int c;
--- 	
--- 	
c= fun (a,b);
--- 	
--- 	
}
	Arguments are passed
	
	value are sent back
		fun (int a, int b)
		{
		---- 	
		---- 	
		
		return c;
		}

iv. Functions with no arguments and return values

Functions that have no arguments but have some return values. Such functions are used to perform specific operations and return their value.



Calling function	Analysis	Called function
main ( )	No arguments are passed	fun ( )
{		{
int c;		----
---		----
c= fun ( );	values are sent back	return c;

-----		}
}		

```
#include<iostream>
using namespace std;
int sum()
{
   int a=5; int b=12;
   int c=a+b;
   return c;
}
int main()
{
   int result = sum();
   cout<<result;
   return 0;
}
```

Output: 17

Passing Parameters to Functions:

- When a function gets executed in the program, the execution control is transferred from calling-function to called function and executes function definition, and finally comes back to the calling function.
- When the execution control is transferred from calling-function to called-function it may carry one or number of data values. These data values are called as parameters.
- Let us assume that a function *B()* is called from another function *A()*. In this case, A is called the “caller function” and B is called the “called function or callee function”.
- Also, the arguments which A sends to B are called **actual arguments** and the parameters of B are called **formal arguments**.
- **Actual Parameter:** are the values that are passed to a function when it is called. They are also known as arguments. These values can be constants, variables, expressions, or even other function calls.
- **Formal Parameter:** are the variables declared in the function header that are used to receive the values of the actual parameters passed during function calls. They are also known as function parameters..

Example:

- int result = add(2, 3);

  In this function call, ***2*** and ***3*** are the ***actual parameters***, and they are passed to the function add, which takes two formal parameters.

- int add(int a, int b);

  In this function declaration, a and b are the formal parameters. They are used to receive the values of the actual parameters passed during function calls.

Difference between Actual and Formal Parameters:

i. The main difference between actual and formal parameters is that actual parameters are the values that are passed to a function when it is called, while formal parameters are the variables declared in the function header that are used to receive the values of the actual parameters passed during function calls.

ii. Actual parameters can be expressions or other function calls, while formal parameters are always variables. It means that actual parameters can be evaluated at runtime, while formal parameters are defined at compile-time.

In C++ Programming Language, there are two methods to pass parameters from calling function to called function and they are as follows...

I. Call by Value

II. Call by Reference

I. Call by Value:

- In call by value parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function.
- ***The changes made on the formal parameters does not effect the values of actual parameters.***
- That means, after the execution control comes back to the calling function, the actual parameter values remains same.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

```
#include <iostream>
using namespace std;
void swap (int a, int b);
int main ()
{
    int num1,num2;
    num1=10; num2=20;
    swap(num1,num2);
    cout<<num1<<'\t'<<num2<<endl;
    return 0;
}
```

```
void swap(int a, int b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}
```

Output:

```
10     20
```

In the above example program, the variables **num1** and **num2** are called actual parameters and the variables **a** and **b** are called formal parameters. The value of **num1** is copied into **a** and the value of num2 is copied into **b**. The changes made on variables **a** and **b** does not effect the values of **num1** and **num2**.

II. Call by Reference:

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

```
#include <iostream>
using namespace std;
void swap (int *a, int *b);
int main ()
{
   int num1,num2;
   num1=10; num2=20;
   cout<<"Before swapping num1 is "<<num1<<" num2 is "<<num2<<endl;
   swap(&num1,&num2);
   cout<<“After swapping num1 is "<<num1<<" num2 is "<<num2;
   return 0;
}
void swap(int *a, int *b)
{
   int temp;
   temp=*a;
   *a=*b;
   *b=temp;
}
```

Output:

```
Before swapping num1 is 10 num2 is 20
After swapping num1 is 20 num2 is 10
```

Summary of Storage Classes in C++

Let us now go Summarize the Storage Classes in C++:

Storage Class	Scope	Lifetime	Initial Value	Storage Location
Auto	Within the block, in which it is declared.	Within the block, in which it is declared.	Initialized with garbage value.	RAM
Register	Within the block, in which it is defined.	Same as auto, within the block in which it is declared.	Initialized with garbage value.	Register
Static	Within the block, in which it is defined.	Throughout the main program.	Zero.	RAM
Extern	Global scope, unbound by any function.	Same as static, till the end of the main program.	Zero.	RAM

Recursion:

- A function that calls itself is known as a recursive function. And, this technique is known as recursion.
- It involves defining a base case to stop the recursive calls and a recursive step that reduces the problem size until the base case is reached.

Syntax:

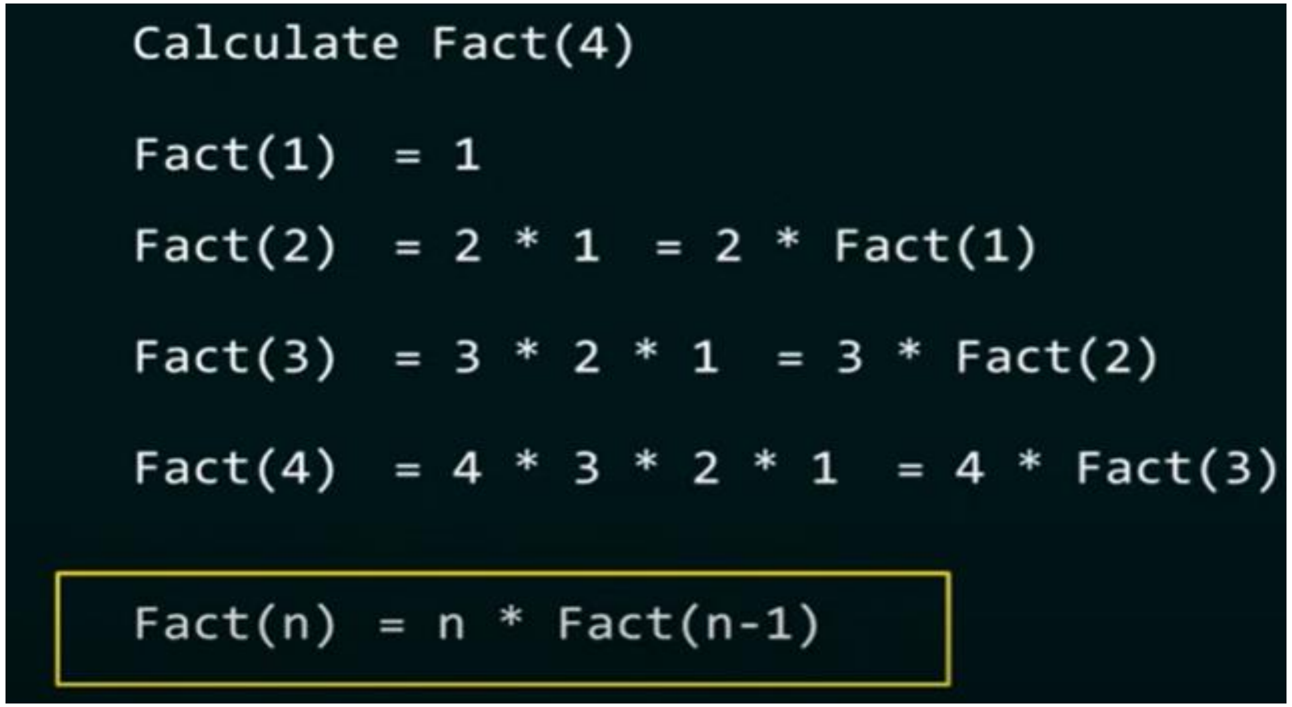
```
return_type recursive_fun(arguments)  //recursive function
{
    base_case;           // Stopping Condition
    recursive_fun(); //recursive call
}
int main()
{
   recursive_fun(arguments); //function call
 }
```

Base case: Recursion involves a base case that stops the recursive calls to prevent infinite loops, ensuring program termination.

Recursive call: The recursive call is the code executed repeatedly inside the recursive function while making recursive calls.

- It is important to define the base condition before the recursive call

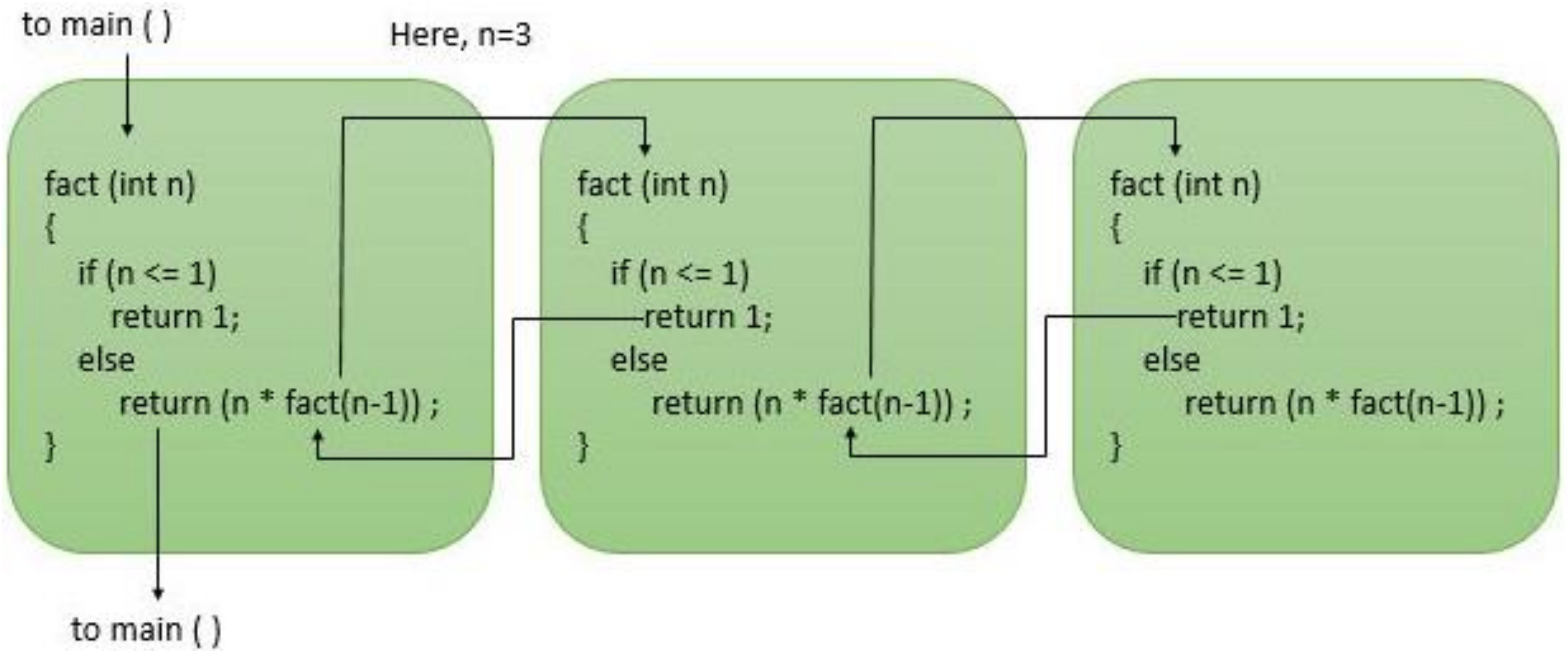
Example: *Factorial of a number using recursive function*



```
Calculate Fact(4)

Fact(1)  = 1

Fact(2)  = 2 * 1  = 2 * Fact(1)

Fact(3)  = 3 * 2 * 1  = 3 * Fact(2)

Fact(4)  = 4 * 3 * 2 * 1  = 4 * Fact(3)

Fact(n)  = n * Fact(n-1)
```

```
#include<iostream>
using namespace std;
int fact(int n)
{
    if (n==1)
    {
        return 1;
    }
    else
    {
        return n*fact(n-1);
    }
}
int main()
{
    int n=5;
    int factorial=fact(n);
    cout<<"Factorial is "<<factorial;
    return 0;
}
```

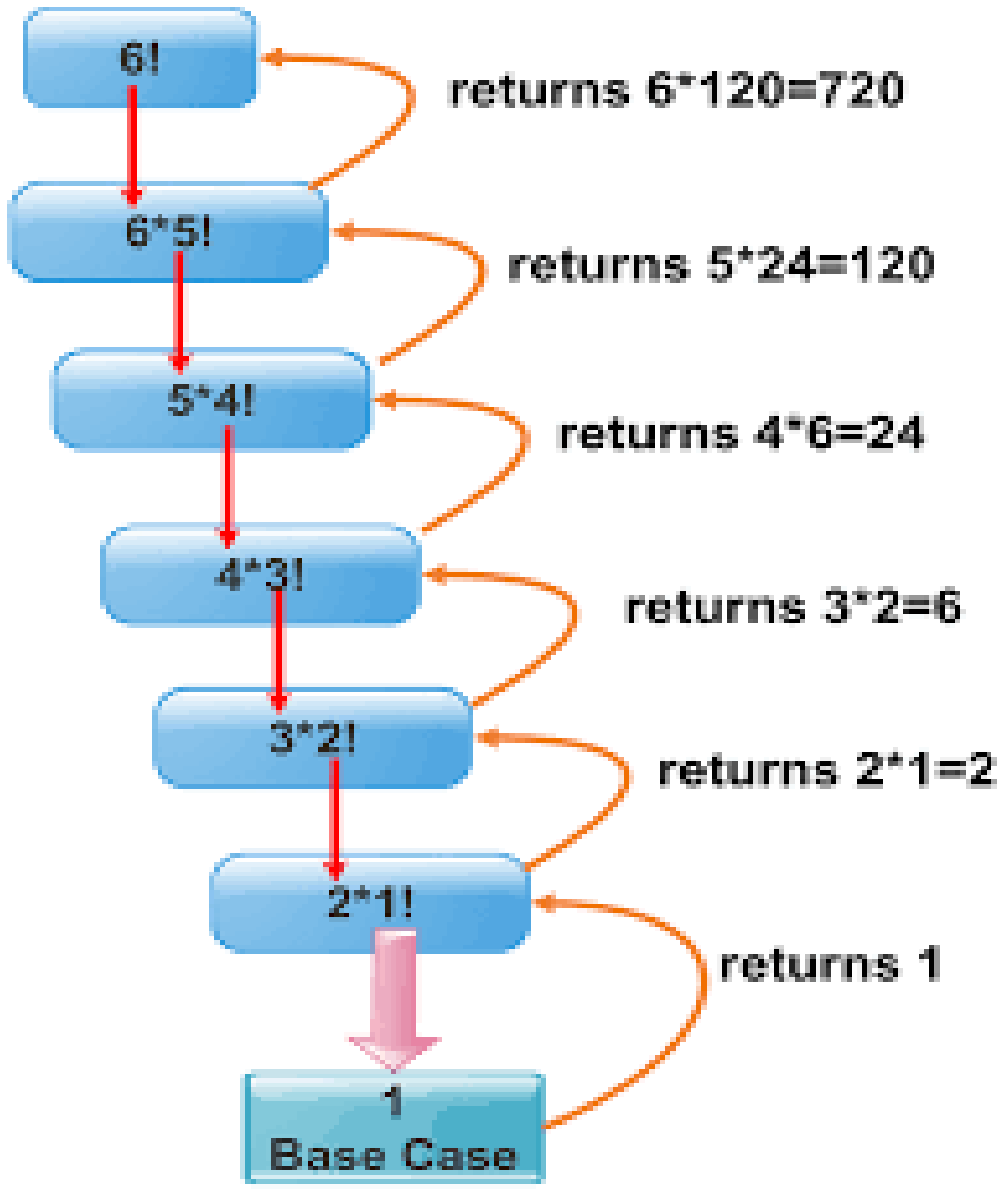
Output: *Factorial is 120*



```
fact (int n)
{
  if (n <= 1)
    return 1;
  else
     return (n * fact(n-1)) ;
}
```

```
fact (int n)
{
  if (n <= 1)
    return 1;
  else
     return (n * fact(n-1)) ;
}
```

```
fact (int n)
{
  if (n <= 1)
    return 1;
  else
     return (n * fact(n-1)) ;
}
```


6!
6*5!
5*4!
4*3!
3*2!
2*1!
1
Base Case
returns 6*120=720
returns 5*24=120
returns 4*6=24
returns 3*2=6
returns 2*1=2
returns 1

Types of Recursion in C++

There are two types of recursion in the C++ language.

a. Direct Recursion

b. Indirect Recursion

c. Tail Recursion

d. No Tail/ Head Recursion

e. Linear recursion

f. Tree Recursion

a. Direct Recursion

Direct recursion in C++ occurs when a function calls itself directly from inside. Such functions are also called direct recursive functions.

Syntax:

```
function_01()
{
    //some code
    function_01();
    //some code
}
```

Example:

```
#include <iostream>
using namespace std;
void directrecursion(int n)
{
    if (n>0)
    {
        cout<<n;
        directrecursion(n-1);
    }
}
int main()
 {
    int num=5;
    cout<<"Direct recursion:";
    directrecursion(num);
    return 0;
}
```

Output: *Direct recursion: 5 4 3 2 1*

b. Indirect Recursion

At least two functions that call each other repeatedly in a cycle constitute indirect recursion.

Syntax:

```
function_01()
{
    //some code
    function_02();
}
function_02()
{
    //some code
   function_01();
}
```

Example:

```
#include <iostream>
using namespace std;
void functionA(int n);
void functionB(int n);
int main()
{
    int num=6;
    cout<<"Indirect recursion:";
    functionA(num);
    return 0;
}
```

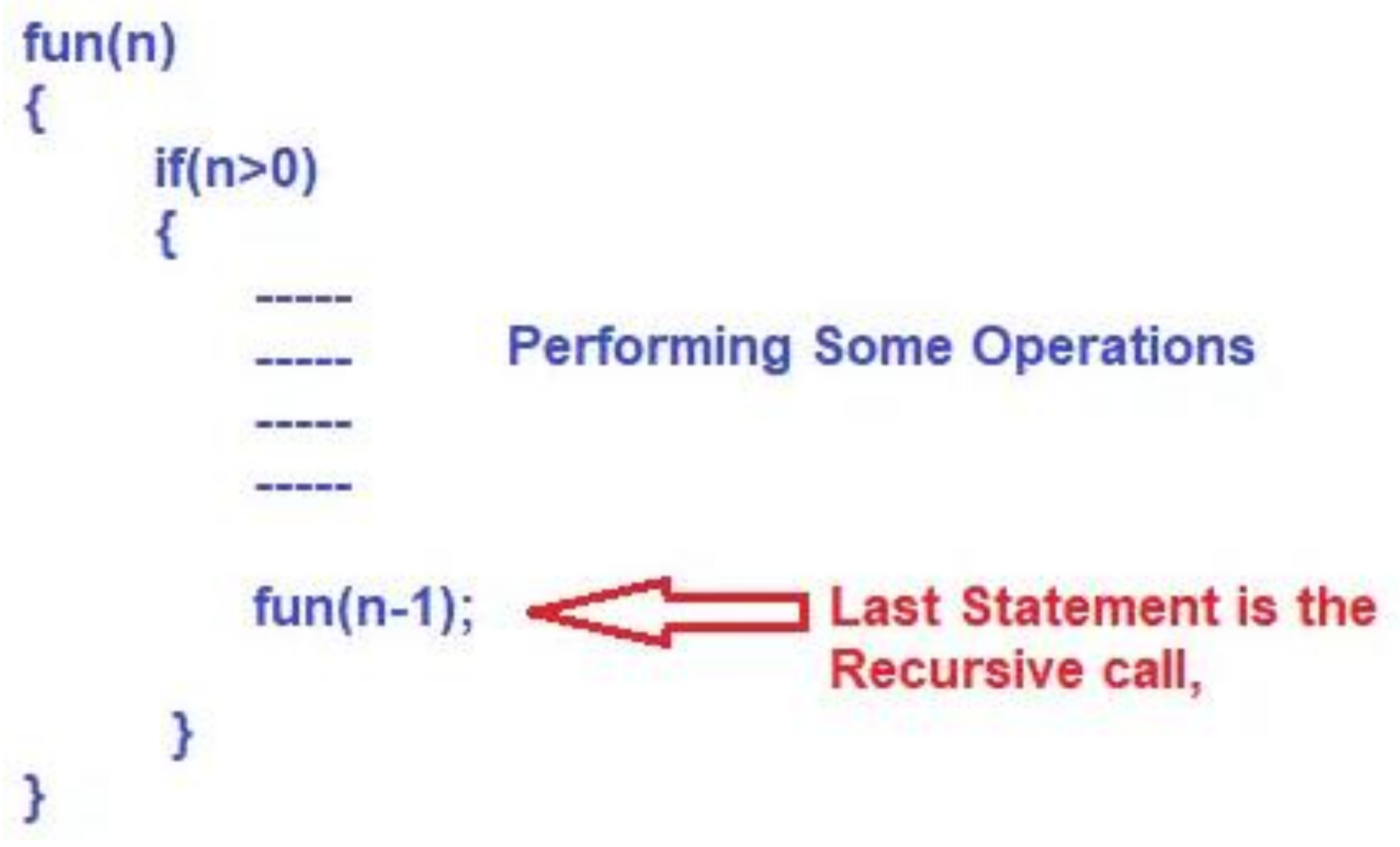
```
void functionA(int n)
{
    if (n>0)
    {
        cout<<n<<'\t’;
        functionB(n-1);
    }
}
void functionB(int n)
{
    if (n>0)
    {
        cout<<n<<'\t’;
        functionA(n/2);
    }
}
```

Output: *Indirect recursion: 6 5 2 1*

c. Tail Recursion

When a recursive function calls itself in a loop and that looping statement is the final one the function performs, the function is said to be "tail-recursive."

Syntax:



```
fun(n)
{
    if(n>0)
    {
        -----
        -----
        -----
        -----

        fun(n-1);

     }
}
```

Performing Some Operations
Last Statement is the Recursive call,

```
#include <iostream>
using namespace std;
void fun(int n);
int main()
{
    fun(3);
    return 0;
}
void fun(int n)
{
    if (n>0)
    {
        cout<<n<<'\t’;
        fun(n-1);
    }
}
```

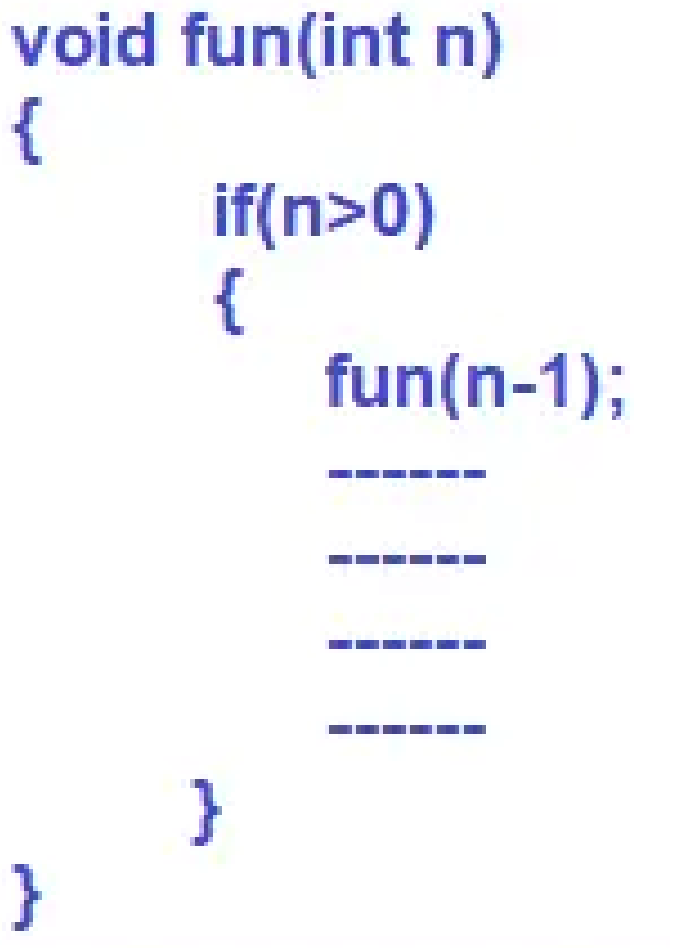
Output:

```
3       2       1
```

d. No Tail/ Head Recursion

The non-tail or head recursion of a function The initial statement in a function will be the recursive call if it does one on its own. It implies that no statement or operation should be called prior to the recursive calls.

Syntax:



```
void fun(int n)
{
        if(n>0)
        {
            fun(n-1);
            ------
            ------
            ------
            ------
        }
}
```

```
#include <iostream>
using namespace std;
void noTailRecursion(int n)
{
   if (n>0)
   {
      noTailRecursion(n-1);
      cout<<n<<'\t’;
      noTailRecursion(n-2);
   }
 }
int main()
 {
   int num=5;
   cout<<"No Tail/Head recursion: ";
   noTailRecursion(num);
   return 0;
}
```

Output:

```
No Tail/Head recursion: 1    2        3          1          4          1
          2          5          1          2          3          1
```

e. Linear Recursion

If a function only makes one call to itself each time it is executed and expands linearly as a function of the size of the problem, the function is said to be linear recursive.

```
#include <iostream>
using namespace std;
int LinearRecursion(int n)
{
    if (n==0)
    {
        return 0;
    }
    else
    {
        return n+LinearRecursion(n-1);
    }
}
```

```
int main()
 {
    int num=5;
    cout<<"Sum is: "<<LinearRecursion(num);
    return 0;
}
```

Output: Sum is: 15

f. Tree Recursion

When a recursive function in C++ calls itself more than once, the result is a branching structure that resembles a tree. This is known as tree recursion.

```
#include <iostream>
using namespace std;
void TreeRecursion(int n)
{
   if (n>0)
   {
      cout<<n<<'\t’;
      TreeRecursion(n-1);
      TreeRecursion(n-1);
   }
}
```

```
int main()
 {
   int num=3;
   cout<<"Tree Recursion: ";
   TreeRecursion(num);
   return 0;
}
```

Output:

```
Tree Recursion: 3       2       1       1       2       1       1
```

Advantages of Recursion

- **Simplicity:** Code gets cleaner and uses fewer needless function calls.
- **Elegant solution:** Helpful for resolving difficult algorithms and formula-based challenges.
- **Tractable approach**: They are useful for traversing trees and graphs because they are naturally recursive.
- **Code reusability**: Recursive functions in C++ can be used repeatedly within the program for different inputs, promoting code reusability.

Disadvantages of Recursion

- **Complexity:** It gets challenging to read and decipher the code.
- **Stack consumption**: The copies of recursive functions take up a significant amount of memory.
- **Performance**: Recursive calls can be slower and less efficient than iterative solutions due to function call overhead and repeated calculations.
- **Limited applicability**: Not all problems are well-suited for C++ recursion, and converting iterative solutions into recursive ones may not always be practical or beneficial

Strings

Fundamentals of string:

Ways to define a string in C++ are:

1. Using C-style strings
2. Using String keyword

1. Using C-style strings

- When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

Declaration:

There are two ways to declare a string in c language.

i. By char array

ii. By string literal

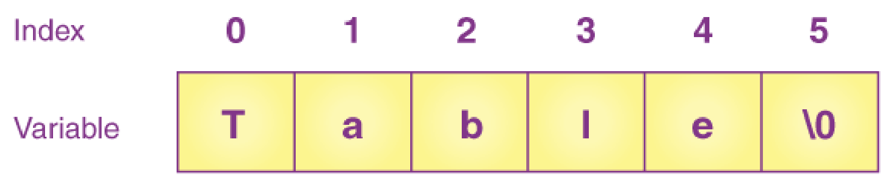
i. By char array:

Syntax: *char string_name [size_of_string]={'char1', 'char2', . . . };*

Example: char s[4]={'C', 'S', 'D','S'}; or char s[] ={'C', 'S', 'D','S'};

ii. By string literal:

Syntax: *char string_name [size_of_string]=“String”;*



Example: char s[5]=“CSDS”; or char s[]=“CSDS”;

Difference between char array and string literal

There are two main differences between char array and literal.

1. We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.
2. The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.
3. Size of string literal is one byte more than the size of char array.
4. Length of both type declarations are same.

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
    char str[]={'C','S','D','S'};
    char str1[]="CSDS";
    puts(str);
    cout<<sizeof(str)<<endl;
    puts(str1);
    cout<<sizeof(str1)<<endl;
    return 0;
}
```

Output:

```
CSDS
4
CSDS
5
```

2. Using String keyword

- C++ offers a new string class similar to the C Strings but with efficient programming capabilities. Internally, the working of the string class is the same as C Strings (by using a character array to store the characters), while the memory definition, allocation, and addition of null characters at the end are all handled by the String class itself.
- string class defined inside <string> header file. This provides many advantages over conventional C-style strings such as dynamic size, member functions, etc.
- **Syntax:**

```
string string_name = "Text";
 string string_name("Text");
```

```
#include<iostream>
using namespace std;
int main()
{
    string str1 = "Training";
    string str2("Training");
    cout << "str1 is " << str1 << endl;
    cout << "str2 is " << str2 << endl;
    return 0;
}
```

Output:

```
str1 is Training
str1 is Training
```

String Input in C++:

Methods to take a string as input are:

- cin
- getline
- stringstream

cin:

The simplest way to take string input is to use the *cin* command along with the stream extraction operator (>>).

Syntax:

```
cin>>s;
```

```
#include <iostream>
using namespace std;
int main()
{
    string s;
    cout<<"Enter String"<<endl;
    cin>>s;
    cout<<"String is: "<<s<<endl;
    return 0;
}
```

Output:

```
Enter String
Training class
String is: Training
```

- By default, the extraction operator (>>) considers white space (such as space, tab, or newline) as the terminating character.
- In the above program, only the first word, *"Training"* will be considered as the input, and the *" class"* string will be discarded because the *cin()* method will consider the white space after *"Training"* as the terminating character.

getline:

The *getline()* function is used to read a string entered by the user. The *getline()* function extracts characters from the input stream. It adds it to the string object until it reaches the delimiting character. The default delimiter is \n.

Syntax:

std::getline(cin, string_variable);

string_variable: The string where the line is stored.

```
#include <iostream>
using namespace std;
int main()
{
   string s;
   cout << "Enter String" << endl;
   getline(cin, s);
   cout << "String is: " << s << endl;
   return 0;
}
```

Output:

```
Enter String
Training class
String is: Training class
```

stringstream:

The *stringstream* class in C++ is used to take multiple strings as input at once.

Syntax:

stringstream stringstream_object(string_name);

- The *gets()* and *puts()* are declared in the header file *stdio.h*. Both the functions are involved in the input/output operations of the strings.
- The *gets()* function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string.
- The *gets()* function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered.
- The *fgets()* makes sure that not more than the maximum limit of characters are read.
- The *puts()* function is used to print the string on the console which is previously read by using gets() or scanf() function.

The <string> library has many functions that allow you to perform tasks on strings.

Function	Description
at()	Returns an indexed character from a string
length()	Returns the length of a string
size()	Alias of length(). Returns the length of a string
empty()	Checks wheter a string is empty or not
append()	Appends a string (or a part of a string) to another string
substr()	Returns a part of a string from a start index (position) and length
find()	Returns the index (position) of the first occurrence of a string or character
rfind()	Returns the index (position) of the last occurrence of a string or character
replace()	Replaces a part of a string with another string
insert()	Inserts a string at a specified index (position)
erase()	Removes characters from a string
compare()	Compares two strings

at():

at() in C++ is a built-in function of std::string class that is used to extract the character from the given index of string. If the given index is less than 0 or greater than equal to the length of string, then it throws an out_of_range exception

Syntax:

str.at(index)

```
#include<iostream>
using namespace std;
int main() {
   string str("Training class");
   cout << str.at(0) << endl;
   cout << str.at(5) << endl;
   cout << str.at(16) << endl;
   return 0;
}
```

Output:

```
T
i
terminate called after throwing an instance of
'std::out_of_range'
  what():  basic_string::at: __n (which is 16) >=
this->size() (which is 14)
```

length() or size():

It will return the length of the string.

Example of *length()*:

```
#include<iostream>
using namespace std;
int main() {
    string str("Training class");
    cout<<str.length()<<endl;
    cout<<str.size()<<endl;
    return 0;
}
```

Output:

```
14
14
```

empty():

- The *empty()* function checks whether a string is empty or not.
- It returns 1 if the string is empty, otherwise 0.

```
#include<iostream>
using namespace std;
int main() {
    string str1 = "Training Class";
    string str2 = "";
    cout << str1.empty()<<endl;
    cout << str2.empty();
    return 0;
}
```

Output:

```
0
1
```

append():

append() function can be used to do the following append operations.

- i. Append a Whole String
- ii. Append a Part of the String
- iii. Append a Character Multiple Times
- iv. Append Characters from a Given Range

Append a Whole String

append() method can be used to append the whole string at the end of the given string.

Syntax

```
str1.append(str2);
```

```
#include<iostream>
using namespace std;
int main() {
    string str1("Training class");
    string str2(" of ITER");
    str1.append(str2);
    cout << str1 << endl;
    return 0;
}
```

Output:

```
Training class of ITER
```

Append a Part of the String

append() method can also be used to append the substring starting from the particular position till the given number of characters.

Syntax:

str1.append(str2, pos, num);

```
#include<iostream>
using namespace std;
int main() {
   string str1("Training class");
   string str2(" of ITER");
   str1.append(str2, 3, 5);
   cout << str1 << endl;
   return 0;
}
```

Output:

```
Training class ITER
```

Append a Character Multiple Times

append() method is used to append the multiple characters at the end of the string.

Syntax:

str.append(num, c);

c: Character which we have to append multiple times.

num: Number of times up to which characters will append.

```
#include<iostream>
using namespace std;
int main()
{
   string str("Training class");
   str.append(3, '?');
   cout << str;
   return 0;
}
```

Output:

```
Training class???
```

Append Characters from a Given Range

append() method can also be used to append the characters from the given range at the end of the string.

Syntax:

str.append(first, last);

```
#include<iostream>
using namespace std;
int main()
{
    string str1("Training class");
    string str2("of ITER");
    str1.append(str2.begin() + 2, str2.end());
    cout << str1;
    return 0;
}
```

Output:

```
Training class ITER
```

substr():

- The substring function takes two values *pos* and *len* as an argument and returns a newly constructed string object with its value initialized to a copy of a sub-string of this object.

Syntax:

string substr (size_t pos, size_t len);

pos: *Index of the first character to be copied.*

len: *Length of the sub-string.*

size_t: *It is an unsigned integral type.*

```
#include<iostream>
using namespace std;
int main()
{
   string str1 = "Training class";
   string str2 = str1.substr(3, 5);
   cout << str2;
   return 0;
}
```

Output:

```
ining
```

find():

- string *find()* is a library function used to find the first occurrence of a sub-string in the given string.

```
#include<iostream>
using namespace std;
int main()
{
    string str1 = "Training class of ITER";
    string str2 = "of";
    cout << str1.find(str2);
    return 0;
}
```

Output:

```
15
```

rfind():

- *rfind()* function is used to locate the last occurrence of a specified substring or character within a string.

```
#include<iostream>
using namespace std;
int main()
{
    string str1 = "Training class of ITER!";
    string str2 = "class";
    int i = str1.rfind(str2);
    cout << i;
    return 0;
}
```

Output:

```
9
```

replace():

The C++ string *replace()* function is used to replace a substring of a string with another given string.

Syntax:

str1.replace(position,length,str2);

position : The position defines the starting position of the substring of str1.
length: The number of characters starting from the position to be replaced by another string.
str2: str2 is the string that will replace the substring of string str1.

```
#include<iostream>
using namespace std;
int main()
{
   string str1 = "Training class of ITER";
   string str2 = "SOA";
   cout<<str1<<'\n';
   str1.replace(18,5,str2);
   cout<<str1<<'\n';
   return 0;
}
```

Output:

```
Training class of ITER
Training class of SOA
```

insert():

- *insert()* function is used insert the characters or a string at the given position of the string.

Insert a Single Character:

insert() method can also be used to insert the single character at some given position of the string.

Syntax:

str.insert(pos, character);

```
#include<iostream>
using namespace std;
int main()
{
   string str("Trainin class");
   str.insert(str.begin() + 7, 'g');
   cout << str;
   return 0;
}
```

Output:

```
Training class
```

Insert a Single Character Multiple Times

insert() method can also be used to insert a single character multiple time at the given position in the string.

Syntax:

str.insert(pos, num, character);

```
#include<iostream>
using namespace std;
int main()
{
   string str("Training cla of ITER");
   str.insert(12, 2, 's');
   cout << str;
   return 0;
}
```

Output:

```
Training class of ITER
```

Insert Characters from the Given Range

Syntax:

str.insert(pos, first, last);

```
#include<iostream>
using namespace std;
int main()
{
   string str1("Train class of ITER");
   string str2("iingg");
   auto first = str2.begin() + 1;
   auto last = str2.end() - 1;
   str1.insert(str1.begin() + 5, first, last);
   cout << str1;
   return 0;
}
```

Output:

```
Training class of ITER
```

Insert a String

insert() method is used to insert the string at the specific position of the string.

Syntax:

str1.insert(pos, str2);

```
#include<iostream>
using namespace std;
int main()
{
   string str1("Training of ITER");
   string str2("class ");
   str1.insert(9, str2);
   cout << str1;
   return 0;
}
```

Output:

```
Training class of ITER
```

erase():

- *erase()* function is a built-in function of std::string class that is used to erase the whole or a part of the string shortening its length.

Syntax:

```
s.erase()               // Erases whole string
s.erase(idx)            // Erases all characters after idx
s.erase(idx, k)         // Erases k characters after idx
s.erase(itr)            // Erases character at itr
s.erase(first, last) // Erases character in range [first, last)
```

```
#include<iostream>
using namespace std;
int main()
{
   string str("Training of ITER");
   str.erase();
   cout << str1;
   return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{
    string str("Training of ITER");
    str.erase(4);
    cout << str;
    return 0;
}
```

Output: Trai

```
#include<iostream>
using namespace std;
int main()
{
    string str("Training of ITER");
    str.erase(9,2);
    cout << str;
    return 0;
}
```

Output: Training ITER

compare():

➢ *compare()* function in C++ is used to compare a string or the part of string with another string or substring.

Compare Two Strings

The string::compare() method can be used to compare the one string with another string that is passed to it as argument.

- Returns 0 if str1 is equal to str2.
- Returns a positive integer if the length of str1 is greater than str2.
- Returns a negative integer if the length of str1 is smaller than str2.

Syntax:

str1.compare(str2);

```
#include<iostream>
using namespace std;
int main()
{
   string str1("Training");
   string str2("Training");
   if ((str1.compare(str2))==0)
      cout << "String Matched" << endl;
   else
      cout << "String Not Matched" << endl;
   return 0;
}
```

Output:

```
String Matched
```

Compare Substring with Another String

compare() method also supports the comparison between a part of the string with another string.

Syntax:

str1.compare(pos, n, str2);

```
#include<iostream>
using namespace std;
int main()
{
    string str1("Training");
    string str2("ihf");
    int r=str1.compare(2,3,str2);
    cout <<r<< endl;
    return 0;
}
```

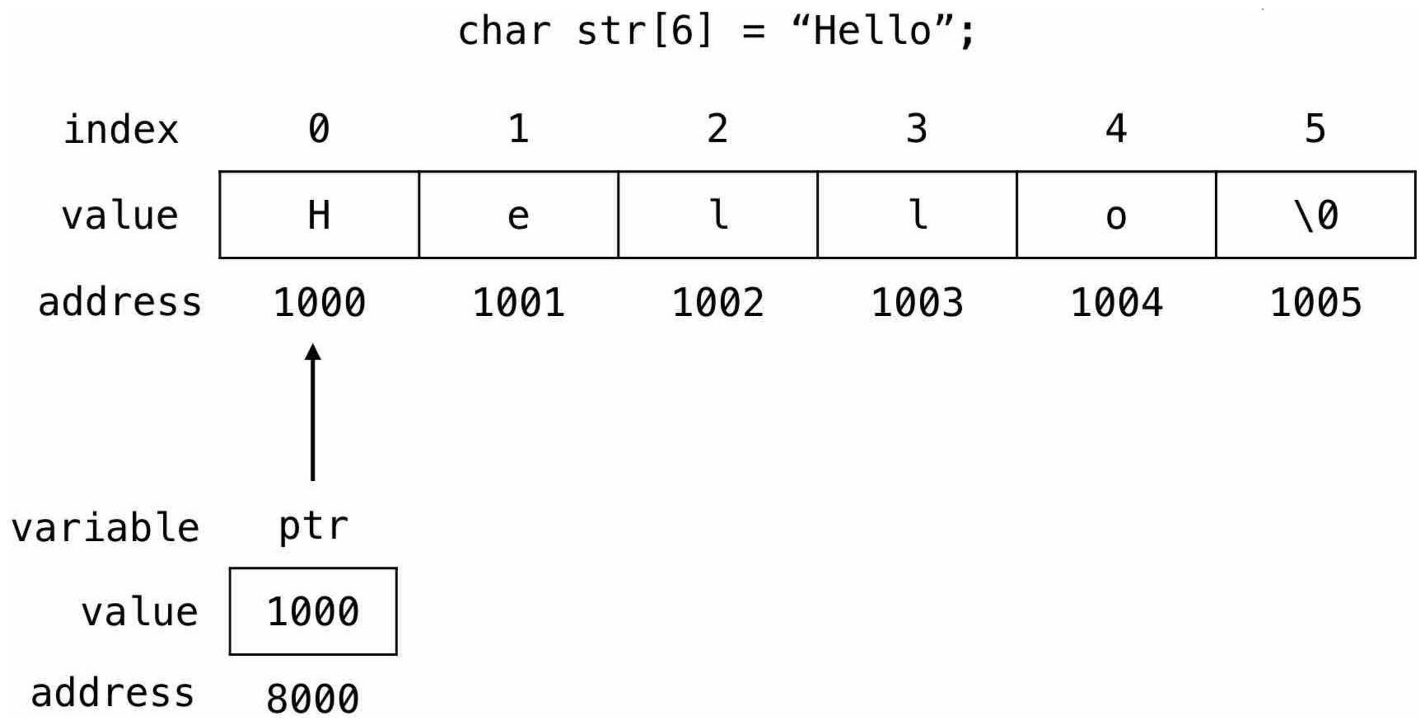
Output:

```
-1
```

Pointers and Strings:

Creating a pointer for the string

- The variable name of the string str holds the address of the first element of the array i.e., it points at the starting memory address.
- So, we can create a character pointer ptr and store the address of the string str variable in it. This way, ptr will point at the string str.
- In the following code we are assigning the address of the string str to the pointer ptr.
- char *ptr = str;
- We can represent the character pointer variable ptr as follows.



```
#include<iostream>

using namespace std;

int main()

{

    string str("Training");

    string *ptr=&str;

    cout<<*ptr<<endl;

    cout<<ptr<<endl;

    cout<<str;

    return 0;

}
```

Output:

```
Training

0x61fef0

Training
```

Program for Return maximum occurring character in the input string

Program for Remove all duplicates from the input string.

Program for Print all the duplicates in the input string.

Program for Remove characters from the first string which are present in the second string

Program for A Program to check if strings are rotations of each other or not

Program for Print reverse of a string using recursion

Program for Divide a string in N equal parts

Program for Reverse words in a given string

Program for Length of the longest substring without repeating characters

Program for Print all permutations with repetition of characters

Program for Print all interleavings of given two strings

Program for Check whether two strings are anagram of each other

Program for Longest Palindromic Substring

Program for Count words in a given string

Program for Remove “b” and “ac” from a given string

Program for Find the first non-repeating character from a stream of characters

Program for Recursively remove all adjacent duplicates

Program for Rearrange a string so that all same characters become d distance away

How do you split a string by a delimiter in C++?

How do you efficiently remove duplicate characters from a string?

How do you implement a function to check if a string is a palindrome?

How do you count the occurrences of a specific character in a string?

How do you convert a string to an integer in C++?

Thank You