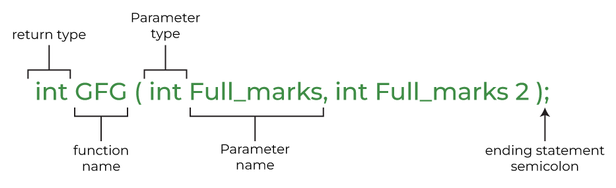
**Functions:**

A function is a **set of statements** that **takes input**, **does** some **specific computation**,and **produces output**. The idea is to put some commonly or repeatedlydone tasks together to make a **function**so that instead of writing the same code again and again for different inputs, we can call this function.

In simple terms, a function is a block of code that runs only when it is called.

**Syntax:**



**Example:** // C++ Program to demonstrate working of a function

#include <iostream>

using namespace std;

int max(int x, int y)

{

    if (x > y)

        return x;

    else

        return y;

}

int main()

{

    int a = 10, b = 20;

     int m = max(a, b);

     cout << "m is " << m;

    return 0;

}

**Output:**

m is 20

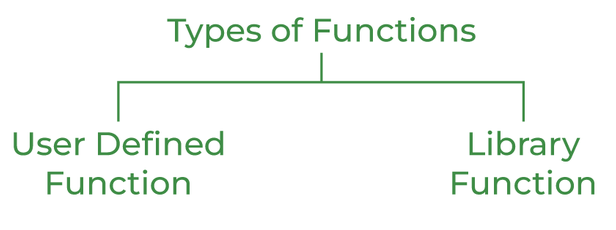
## ****Why Do We Need Functions?****

* Functions help us in ***reducing code redundancy***. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to make changes in only one place if we make changes to the functionality in future.
* Functions make code ***modular***. Consider a big file having many lines of code. It becomes really simple to read and use the code,  if the code is divided into functions.
* Functions provide ***abstraction***. For example, we can use library functions without worrying about their internal work.

**Essentials of Function:**

1. Function Declaration
2. Function Definition
3. Function Call

**Types of Functions**



### **User Defined Function**

User-defined functions are user/customer-defined blocks of code specially customized to reduce the complexity of big programs. They are also commonly known as “***tailor-made functions***” which are built only to satisfy the condition in which the user is facing issues meanwhile reducing the complexity of the whole program.

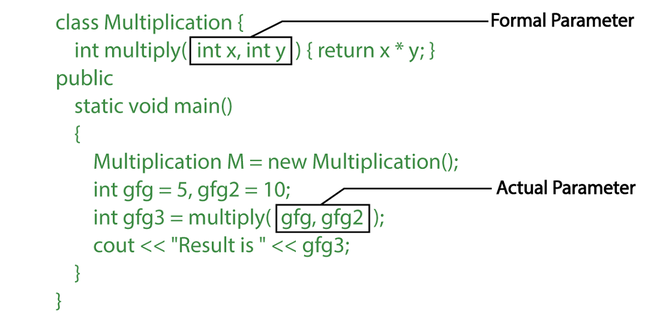
### **Library Function**

Library functions are also called “***built-in Functions***“. These functions are part of a compiler package that is already defined and consists of a special function with special and different meanings. Built-in Function gives us an edge as we can directly use them without defining them whereas in the user-defined function we have to declare and define a function before using them. 

**For Example:**sqrt(), setw(), strcat(), etc.

## Parameter Passing to Functions

The parameters passed to the function are called ***actual parameters***. For example, in the program below, 5 and 10 are actual parameters.   
The parameters received by the function are called ***formal parameters***. For example, in the above program x and y are formal parameters.



**There are two most popular ways to pass parameters:**

1. ***Pass by Value or Call by Value:*** In this parameter passing method, values of actual parameters are copied to the function’s formal parameters. The actual and formal parameters are stored in different memory locations so any changes made in the functions are not reflected in the actual parameters of the caller.
2. ***Pass by Reference or Call by Reference:*** Both actual and formal parameters refer to the same locations, so any changes made inside the function are **reflected in the actual parameters** of the caller.

**Example-1:** ***Pass by Value or Call by Value***

#include<iostream>

using namespace std;

void my\_function(int x) {

x = 50;

cout << "Value of x from my\_function: " << x << endl;

}

main() {

int x = 10;

my\_function(x);

cout << "Value of x from main function: " << x;

}

**Output:**

Value of x from my\_function: 50

Value of x from main function: 10

**Example-2:** ***Pass by Value or Call by Value***

#include <iostream>

using namespace std;

void swap(int x, int y) {

int temp = x;

x = y;

y = temp;

}

int main() {

int a = 40;

int b = 50;

cout << "Before swap: a = " << a << " b = " << b << endl;

swap(a, b);

cout << "After swap: a = " << a << " b = " << b << endl;

return 0;

}

Output:

Before swap: a = 40 b = 50

After swap: a = 40 b = 50

**Example 1:** ***Pass by Reference or Call by Reference:***

#include <iostream>

using namespace std;

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int main() {

int x = 5;

int y = 10;

cout << "Before swap: x = " << x << " , y = " << y << endl;

swap(&x, &y);

cout << "After swap: x = " << x << ", y = " << y << endl;

return 0;

}

Output:

Before swap: x = 5, y = 10

After swap: x = 10, y = 5

**Inline Functions:**

C++ provides inline functions to **reduce the function call overhead**. An inline function is a function that is **expanded in line** **when it is called**. When the inline function is called **whole code of the inline function gets inserted or substituted** at the point of the inline function call. This substitution is performed by the C++ compiler at **compile time**. An inline function may increase **efficiency** if it is **small**.

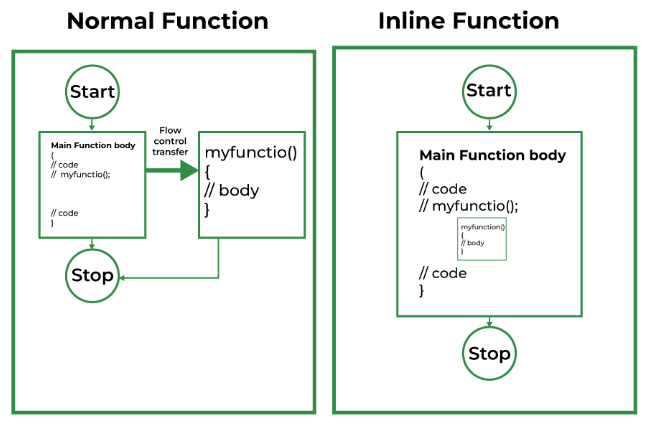
**Syntax:**

**inline** return-type function-name(parameters)

{

// function code

}



**Remember, inlining is only a request to the compiler, not a command. The compiler can ignore the request for inlining.**

**The compiler may not perform inlining in such circumstances as:**

1. If a function contains a **loop**. (*for, while and do-while*)
2. If a function contains **static** **variables**.
3. If a function is **recursive**.
4. If a function **return type is other than void**, and the return statement doesn’t exist in a function body.
5. If a function contains a **switch or goto statement**.

### **Why Inline Functions are Used?**

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack, and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register, and returns control to the calling function. This can become overhead if the execution time of the function is less than the switching time from the caller function to called function (callee).

For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function’s code. This overhead occurs for small functions because the execution time of a small function is less than the switching time.

## ****Inline functions Advantages:****

1. Function call overhead doesn’t occur.
2. It also saves the overhead of push/pop variables on the stack when a function is called.
3. It also saves the overhead of a return call from a function.
4. When you inline a function, you may enable the compiler to perform context-specific optimization on the body of the function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of the calling context and the called context.
5. An inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function called preamble and return.

**Example:**

#include <iostream>

using namespace std;

inline int cube(int s) { return s \* s \* s; }

int main()

{

    cout << "The cube of 3 is: " << cube(3) << "\n";

    return 0;

}

**Output**

The cube of 3 is: 27

**Example 2:**

#include<iostream>

using namespace std;

inline int setNum()

{ return 10;

}

int main()

{ int num ;

num = setNum(); **// setNum() will be replaced by definition of inline function**

cout << " The inline function returned: " << num ;

cout << "\n\n";

return 0 ;

}

**Output:**

The inline function returned: 10

**C++ User-defined Function Types**

* Function with no argument and no return value
* Function with no argument but return value
* Function with argument but no return value
* Function with argument and return value

**Example 1: No arguments passed and no return value**

# include <iostream>

using namespace std;

void prime();

int main()

{

prime();

return 0;

}

void prime()

{

int num, i, flag = 0;

cout << "Enter a positive integer enter to check: ";

cin >> num;

for(i = 2; i <= num/2; ++i)

{

if(num % i == 0)

{

flag = 1;

break;

}

}

if (flag == 1)

{

cout << num << " is not a prime number.";

}

else

{

cout << num << " is a prime number.";

}

}

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

#include <iostream>

using namespace std;

int prime();

int main()

{

int num, i, flag = 0;

// No argument is passed to prime()

num = prime();

for (i = 2; i <= num/2; ++i)

{

if (num%i == 0)

{

flag = 1;

break;

}

}

if (flag == 1)

{

cout<<num<<" is not a prime number.";

}

else

{

cout<<num<<" is a prime number.";

}

return 0;

}

// Return type of function is int

int prime()

{

int n;

printf("Enter a positive integer to check: ");

cin >> n;

return n;

}

**Example 3: Arguments passed but no return value**

#include <iostream>

using namespace std;

void prime(int n);

int main()

{

int num;

cout << "Enter a positive integer to check: ";

cin >> num;

// Argument num is passed to the function prime()

prime(num);

return 0;

}

// There is no return value to calling function. Hence, return type of function is void. \*/

void prime(int n)

{

int i, flag = 0;

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

{

if (n%i == 0)

{

flag = 1;

break;

}

}

if (flag == 1)

{

cout << n << " is not a prime number.";

}

else {

cout << n << " is a prime number.";

}

}

**Example 4: Arguments passed and a return value.**

#include <iostream>

using namespace std;

int prime(int n);

int main()

{

int num, flag = 0;

cout << "Enter positive integer to check: ";

cin >> num;

// Argument num is passed to check() function

flag = prime(num);

if(flag == 1)

cout << num << " is not a prime number.";

else

cout<< num << " is a prime number.";

return 0;

}

/\* This function returns integer value. \*/

int prime(int n)

{

int i;

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

{

if(n % i == 0)

return 1;

}

return 0;

}

**Default Arguments in C++**

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn’t provide a value for the argument. In case any value is passed, the default value is overridden.

Example:

// CPP Program to demonstrate Default Arguments

#include <iostream>

using namespace std;

// A function with default arguments, it can be called with

// 2 arguments or 3 arguments or 4 arguments.

int sum(int x, int y, int z = 0, int w = 0) //assigning default values to z, w as 0

{

    return (x + y + z + w);

}

int main()

{

    cout << sum(10, 15) << endl;

    cout << sum(10, 15, 25) << endl;

    cout << sum(10, 15, 25, 30) << endl;

    return 0;

}

**Output:**

25

50

80

**Pointers and Reference**

**Pointers:** A pointer is a variable that holds the memory address of another variable. A pointer needs to be dereferenced with the **\*** operator to access the memory location it points to.

[**References**](https://www.geeksforgeeks.org/references-in-c/)**:** A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer, is also implemented by storing the address of an object.

**Syntax:**

int i = 3;

// A pointer to variable i or "stores the address of i"

int \*ptr = &i;

// A reference (or alias) for i.

int &ref = i;

**Differences**:

1.**Initialization:** A pointer can be initialized in this way:

int a = 10;  
int \*p = &a;  
// OR   
int \*p;  
p = &a;

We can declare and initialize pointer at same step or in multiple line.

2. While in references,

int a = 10;  
int &p = a; // It is correct  
// but  
int &p;  
p = a; // It is incorrect as we should declare and initialize references at single step

**NOTE:**This difference may vary from compiler to compiler. The above difference is with respect to Turbo IDE.

3. **Reassignment:** A pointer can be re-assigned. This property is useful for the implementation of data structures like a linked list, a tree, etc. See the following example:

int a = 5;  
int b = 6;  
int \*p;  
p = &a;  
p = &b;

4. On the other hand, a reference cannot be re-assigned, and must be assigned at initialization.

int a = 5;  
int b = 6;  
int &p = a;  
int &p = b; // This will throw an error of "multiple declaration is not allowed"  
  
// However it is valid statement,  
int &q = p;

5. **Memory Address:** A **pointer** has its **own memory address** and size on the stack, whereas a **reference** **shares the same memory address** with the original variable and takes up no space on the stack.

int &p = a;  
cout << &p << endl << &a;

6. **NULL value:**A pointer can be assigned NULL directly, whereas a reference cannot be. The constraints associated with references (no NULL, no reassignment) ensure that the underlying operations do not run into an exception situation.

7. **Indirection:** You can have a **pointer to pointer** (known as a double pointer) offering extra levels of indirection, whereas references only offer one level of indirection. For example,

In Pointers,  
int a = 10;  
int \*p;  
int \*\*q; // It is valid.  
p = &a;  
q = &p;  
  
// Whereas in references,  
int &p = a;  
int &&q = p; // It is reference to reference, so it is an error

8. **Arithmetic operations:** Various arithmetic operations can be performed on pointers, whereas there is no such thing called Reference Arithmetic.

**Example 1: Pointer**

// C++ program to demonstrate use of pointers in C++;

#include <iostream>

using namespace std;

int main()

{

    int x = 10; // variable declared

    int\* myptr; // pointer variable

    // storing address of x in pointer myptr

    myptr = &x;

    cout << "Value of x is: ";

    cout << x << endl;

    // print the address stored in myptr pointer variable

    cout << "Address stored in myptr is: ";

    cout << myptr << endl;

    // printing value of x using pointer myptr

    cout << "Value of x using \*myptr is: ";

    cout << \*myptr << endl;

    return 0;

}

**Output:**

Value of x is: 10

Address stored in myptr is: 0x7ffd2b32c7f4

Value of x using \*myptr is: 10

**Example 2: Reference**

#include <iostream>

using namespace std;

int main()

{

    int y = 10;

    int& myref = y;

    // changing value of y to 30

    y = 30;

    cout << "value of y is " << y << endl;

    cout << "value of myref after change in value of y is: "<< myref << '\n';

    return 0;

}

Output:

value of y is 30

value of myref after change in value of y is: 30

**Dynamic Memory Management**

Dynamic memory management or allocation in C/C++ refers to performing memory allocation manually by a programmer. Dynamically allocated memory is allocated on **Heap,** and non-static and local variables get memory allocated on **Stack.**

**How is memory allocated/deallocated in C++?**   
**C uses the malloc() and calloc()** function to allocate memory dynamically at run time and uses a free() function to free dynamically allocated memory. **C++** supports these functions and also has two operators **new** and **delete,** that perform the task of allocating and freeing the memory in a better and easier way.

So in C++, Dynamic memory management is possible by using

1. **malloc**
2. **calloc**
3. **new**
4. **delete**

**new Operator**

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

**Syntax to use new operator**

pointer-variable = new data-type;

**Ex:**

// Pointer initialized with NULL

// Then request memory for the variable

int \*p = NULL;

p = new int;

OR

// Combine declaration of pointer

// and their assignment

int \*p = new int;

**Example:**

#include <iostream>

#include <memory>

using namespace std;

int main()

{

// pointer to store the address returned by the new

int\* ptr;

// allocating memory for integer

ptr = new int;

// assigning value using dereference operator

\*ptr = 10;

// printing value and address

cout << "Address: " << ptr << endl;

cout << "Value: " << \*ptr;

return 0;

}

**Output:**

Address: 0x162bc20

Value: 10

**delete operator**

Since it is the programmer’s responsibility to deallocate dynamically allocated memory, programmers are provided delete operator in C++ language.

**Syntax:**

// Release memory pointed by pointer-variable

delete pointer-variable;

Here, the pointer variable is the pointer that points to the data object created by new.

**Ex:**

delete p;

**Example:**

#include <iostream>

using namespace std;

int main () {

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

\*pvalue = 29494.99; // Store value at allocated address

cout << "Value of pvalue : " << \*pvalue << endl;

delete pvalue; // free up the memory.

return 0;

}