C.P.PATEL & F.H.SHAH COMMERCE COLLEGE
(MANAGED BY SARDAR PATEL EDUCATION TRUST)
BCA, BBA (ITM) & **PGDCA** PROGRAMME:
BCA Semester III -Paper Code: US03CBCA23

UNIT 3

C++ Functions

In programming, function refers to a segment that groups code to perform a specific task.
Depending on whether a function is predefined or created by programmer; there are two types of function:

(1) Library Function and (2) User-defined Function

Library Function

Library functions are the built-in function in C++ programming.
Programmer can use library function by invoking function directly; they don't need to write it themselves.

Example 1: Library Function

```
#include <iostream.h>
#include <cmath.h>
using namespace std;
int main()
{
   double number, squareRoot;
   cout << "Enter a number: ";
   cin >> number;
   // sqrt() is a library function to calculate square root
   squareRoot = sqrt(number);
   cout << "Square root of " << number << " = " << squareRoot;
   return 0;
}
```

Output

```
Enter a number: 26
Square root of 26 = 5.09902
```

In the example above, sqrt() library function is invoked to calculate the square root of a number.
Notice code #include <cmath> in the above program. Here, cmath is a header file. The function definition of sqrt()(body of that function) is present in the cmath header file.
You can use all functions defined in cmath when you include the content of file cmath in this program using #include <cmath> .
Every valid C++ program has at least one function, that is, main() function.

C++ Standard Library Function

The C++ Standard Library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output, error checking and many other useful operations. This makes the programmer's job easier, because

these functions provide many of the capabilities programmers need. The C++ Standard Library functions are provided as part of the C++ programming environment.
Header file names ending in .h are "old-style" header files that have been superseded by the C++ Standard Library header files.

C++ Standard Library header file	Explanation
<iostream.h>	Contains function prototypes for the C++ standard input and standard output functions. This header file replaces header file <iostream.h.h>.
<iomanip>	Contains function prototypes for stream manipulators that format streams of data. This header file replaces header file <iomanip.h>.
<cmath>	Contains function prototypes for math library functions. This header file replaces header file <math.h>.
<cstdlib>	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. This header file replaces header file <stdlib.h>.
<cstring>	Contains function prototypes for C-style string-processing functions. This header file replaces header file <string.h>.
<cstdio>	Contains function prototypes for the C-style standard input/output library functions and information used by them. This header file replaces header file <stdio.h>.

Mathematical Functions

Some of the important mathematical functions in header file <**cmath>** are

Function	Meaning
exp(x)	Exponential function of x (ex)
log(x)	logarithm of x
log 10(x)	Logarithm of number x to the base 10
sqrt(x)	Square root of x
pow(x, y)	x raised to the power y
abs(x)	Absolute value of integer number x

Character Functions

All the character functions require **<cctype>** header file. The following table lists the function.

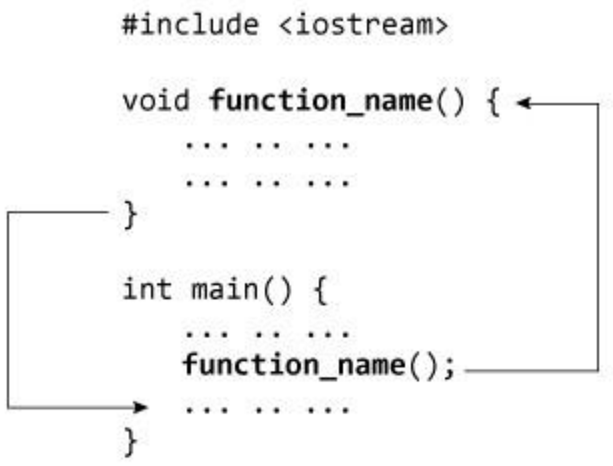
Function	Meaning
isalpha(c)	It returns True if C is an uppercase letter and False if c is lowercase.
isdigit(c)	It returns True if c is a digit (0 through 9) otherwise False.
islower(c)	It returns True if C is a lowercase letter otherwise False.
isupper(c)	It returns True if C is an uppercase letter otherwise False.
toupper(c)	It converts c to uppercase letter.
tolower(c)	It converts c to lowercase letter.

User-defined Function

C++ allows programmer to define their own function.

A user-defined function groups code to perform a specific task and that group of code is given a name(identifier).
When the function is invoked from any part of program, it all executes the codes defined in the body of function.

How user-defined function works in C Programming?



```
#include <iostream>

void function_name() {
    ... .. ...
    ... .. ...
}

int main() {
    ... .. ...
    function_name();
    ... .. ...
}
```

Consider the figure above.
When a program begins running, the system calls the main() function, that is, the system starts executing codes from main() function.
When control of the program reaches to function_name() inside main(), it moves to void function_name() and all codes inside void function_name() is executed.
Then, control of the program moves back to the main function where the code after the call to the function_name() is executed as shown in figure above.

Example 2: User Defined Function

C++ program to add two integers. Make a function add() to add integers and display sum in main() function.

```
#include <iostream.h>
using namespace std;
// Function prototype (declaration)
int add(int, int);
int main()
{
   int num1, num2, sum;
   cout<<"Enters two numbers to add: ";
   cin >> num1 >> num2;
   // Function call
   sum = add(num1, num2);
   cout << "Sum = " << sum;
   return 0;
}
// Function definition
int add(int a, int b)
{
   int add;
   add = a + b;
   // Return statement
   return add;
}
```

Output

```
Enters two integers: 8
-4
Sum = 4
```

Function prototype (declaration)

If a user-defined function is defined after main() function, compiler will show error. It is because compiler is unaware of user-defined function, types of argument passed to function and return type.

In C++, function prototype is a declaration of function without its body to give compiler information about user-defined function. Function prototype in the above example is:

```
int add(int, int);
```

You can see that, there is no body of function in prototype. Also, there are only return type of arguments but no arguments. You can also declare function prototype as below but it's not necessary to write arguments.

```
int add(int a, int b);
```

Note: It is not necessary to define prototype if user-defined function exists before main()function.

Function Call

To execute the codes of function body, the user-defined function needs to be invoked(called).

In the above program, add(num1,num2); inside main() function calls the user-defined function.

The function returns an integer which is stored in variable add.

Function Definition

The function itself is referred as function definition. Function definition in the above program is:

```
// Function definition
int add(int a,int b)
{
   int add;
   add = a + b;
   return add;
}
```

When the function is called, control is transferred to the first statement of the function body.

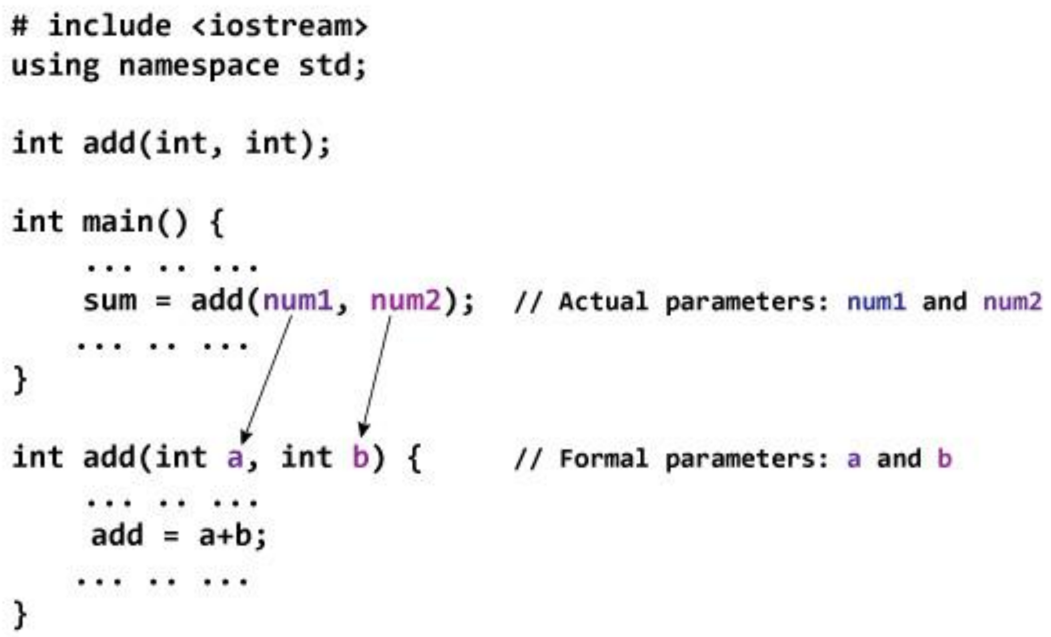
Then, other statements in function body are executed sequentially.

When all codes inside function definition is executed, control of program moves to the calling program.

Passing Arguments to Function

In programming, argument (parameter) refers to the data which is passed to a function (function definition) while calling it.

In the above example, two variables, num1 and num2 are passed to function during function call. These arguments are known as actual arguments.
The value of num1 and num2 are initialized to variables a and b respectively. These arguments a and b are called formal arguments.
This is demonstrated in figure below:



```
# include <iostream>
using namespace std;

int add(int, int);

int main() {
    ... .. ...
    sum = add(num1, num2);  // Actual parameters: num1 and num2
   ... .. ...
}

int add(int a, int b) {     // Formal parameters: a and b
    ... .. ...
     add = a+b;
   ... .. ...
}
```

Notes on passing arguments

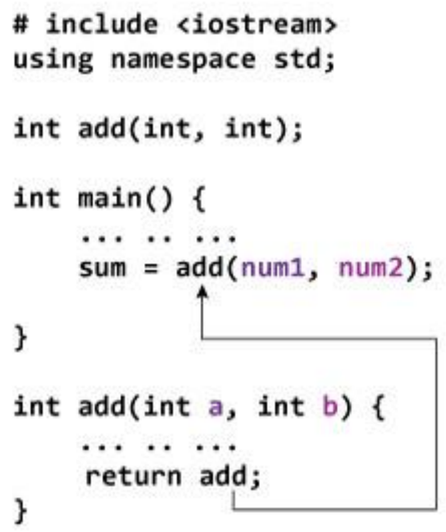
- The numbers of actual arguments and formals argument should be the same. (Exception: Function Overloading)
- The type of first actual argument should match the type of first formal argument. Similarly, type of second actual argument should match the type of second formal argument and so on.
- You may call function a without passing any argument. The number(s) of argument passed to a function depends on how programmer want to solve the problem.
- You may assign default values to the argument. These arguments are known as default arguments.
- In the above program, both arguments are of int type. But it's not necessary to have both arguments of same type.

Return Statement

A function can return a single value to the calling program using return statement.
In the above program, the value of add is returned from user-defined function to the calling program using statement below:

```
return add;
```

The figure below demonstrates the working of return statement.



```
# include <iostream>
using namespace std;

int add(int, int);

int main() {
    ... .. ...
    sum = add(num1, num2);

}

int add(int a, int b) {
    ... .. ...
     return add;
}
```

In the above program, the value of add inside user-defined function is returned to the calling function. The value is then stored to a variable sum.
Notice that the variable returned, i.e., add is of type int and sum is also of int type.
Also, notice that the return type of a function is defined in function declarator int add(int a, int b). The int before add(int a, int b) means the function should return a value of type int.
If no value is returned to the calling function then, void should be used.
Types of User-defined Functions in C++

For better understanding of arguments and return in functions, user-defined functions can be categorized as:

- 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

Consider a situation in which you have to check prime number. This problem is solved below by making user-defined function in 4 different ways as mentioned above.

Example 1: No arguments passed and no return value

```
# include <iostream.h>
using namespace std;
void prime();
int main()
{
   // No argument is passed to prime()
   prime();
   return 0;
}

// Return type of function is void because value is not returned.
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.";
    }
}
```

In the above program, prime() is called from the main() with no arguments.
prime() takes the positive number from the user and checks whether the number is a prime number or not.
Since, return type of prime() is void, no value is returned from the function.

Example 2: No arguments passed but a return value

```
#include <iostream.h>
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;
}
```

In the above program, prime() function is called from the main() with no arguments.
prime() takes a positive integer from the user. Since, return type of the function is an int, it returns the inputted number from the user back to the calling main() function.
Then, whether the number is prime or not is checked in the main() itself and printed onto the screen.

Example 3: Arguments passed but no return value

```
#include <iostream.h>
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.";
    }
}
```

In the above program, positive number is first asked from the user which is stored in the variable num.
Then, num is passed to the prime() function where, whether the number is prime or not is checked and printed.
Since, the return type of prime() is a void, no value is returned from the function.

Example 4: Arguments passed and a return value.

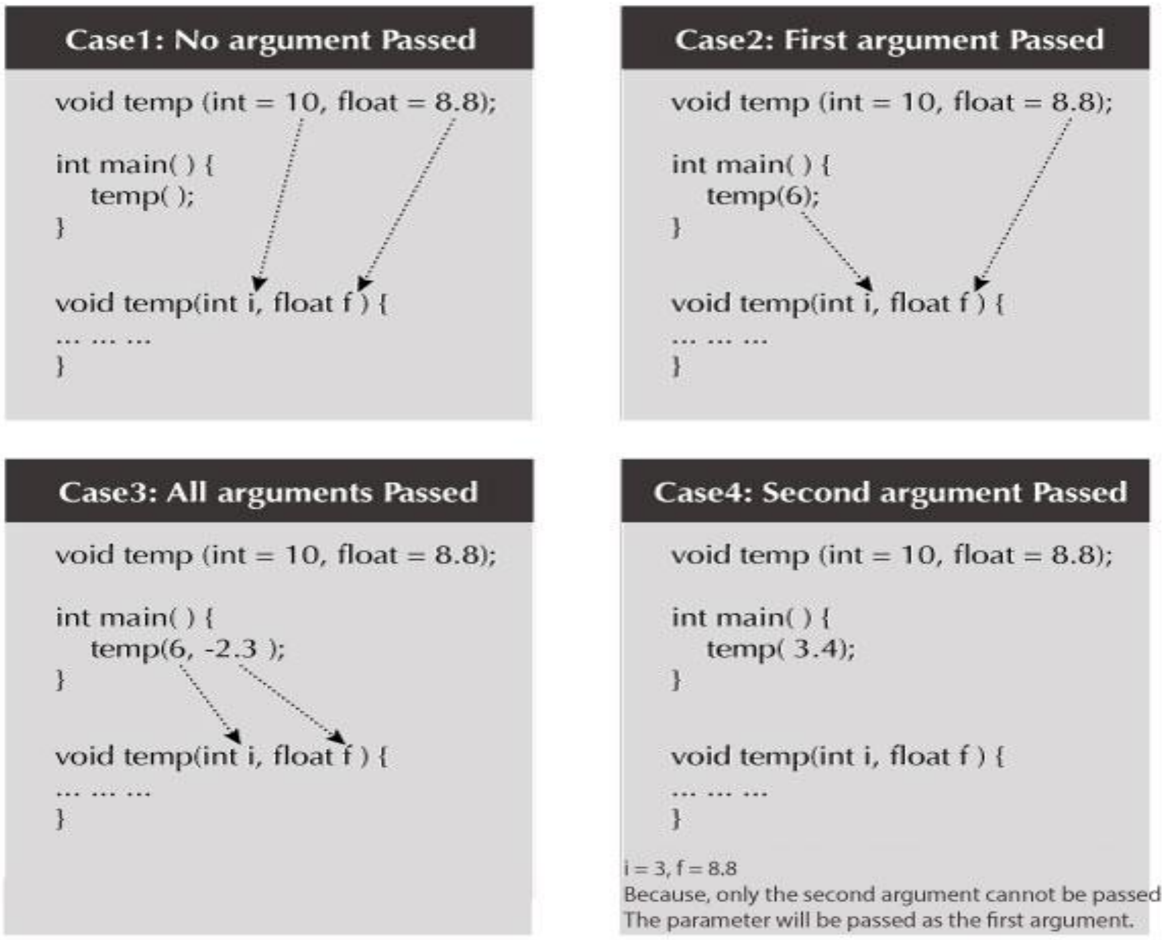
```
#include <iostream.h>
using namespace std;

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;
}
```

In the above program, a positive integer is asked from the user and stored in the variable num.
Then, num is passed to the function prime() where, whether the number is prime or not is checked.
Since, the return type of prime() is an int, 1 or 0 is returned to the main() calling function. If the number is a prime number, 1 is returned. If not, 0 is returned.
Back in the main() function, the returned 1 or 0 is stored in the variable flag, and the corresponding text is printed onto the screen.

C++ Programming Default Arguments (Parameters)

In C++ programming, you can provide default values for function parameters.
The idea behind default argument is simple. If a function is called by passing argument/s, those arguments are used by the function.
But if the argument/s are not passed while invoking a function then, the default values are used.
Default value/s are passed to argument/s in the function prototype.

Working of default arguments



Case1: No argument Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
   temp( );
}

void temp(int i, float f ) {
... ... ...
}
```

Case2: First argument Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
   temp(6);
}

void temp(int i, float f ) {
... ... ...
}
```

Case3: All arguments Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
   temp(6, -2.3 );
}

void temp(int i, float f ) {
... ... ...
}
```

Case4: Second argument Passed

```
void temp (int = 10, float = 8.8);

int main( ) {
   temp( 3.4);
}

void temp(int i, float f ) {
... ... ...
}
```

i = 3, f = 8.8
Because, only the second argument cannot be passed
The parameter will be passed as the first argument.

Figure: Working of Default Argument in C++

Example: Default Argument

// C++ Program to demonstrate working of default argument

```
#include <iostream.h>
using namespace std;

void display(char = '*', int = 1);

int main()
{
   cout << "No argument passed:\n";
   display();

   cout << "\nFirst argument passed:\n";
   display('#');

   cout << "\nBoth argument passed:\n";
   display('$', 5);

   return 0;
}

void display(char c, int n)
{
   for(int i = 1; i <= n; ++i)
   {
      cout << c;
   }
   cout << endl;
}
```

Output

```
No argument passed:
*
First argument passed:
#
Both argument passed:
$$$$$
```

In the above program, you can see the default value assigned to the arguments void display(char = '*', int = 1);.
At first, display() function is called without passing any arguments. In this case, display()function used both default arguments c = * and n = 1.
Then, only the first argument is passed using the function second time. In this case, function does not use first default value passed. It uses the actual parameter passed as the first argument c = # and takes default value n = 1 as its second argument.
When display() is invoked for the third time passing both arguments, default arguments are not used. So, the value of c = $ and n = 5.

Common mistakes when using Default argument

1. void add(int a, int b = 3, int c, int d = 4);

The above function will not compile. You cannot miss a default argument in between two

arguments.
In this case, c should also be assigned a default value.

2. void add(int a, int b = 3, int c, int d);

   The above function will not compile as well. You must provide default values for each argument after b.

   In this case, c and d should also be assigned default values.
   If you want a single default argument, make sure the argument is the last one. void add(int a, int b, int c, int d = 4);

3. No matter how you use default arguments, a function should always be written so that it serves only one purpose.
   If your function does more than one thing or the logic seems too complicated, you can use Function overloading to separate the logic better.

C++ Function Overloading

Two or more functions having same name but different argument(s) are known as overloaded functions.

Function refers to a segment that groups code to perform a specific task.
In C++ programming, two functions can have same name if number and/or type of arguments passed are different.
These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.
Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
// Error code
int test(int a) { }
double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

Example 1: Function Overloading

```
#include <iostream.h>
using namespace std;
```

```
void display(int);
void display(float);
void display(int, float);

int main() {

   int a = 5;
   float b = 5.5;

   display(a);
   display(b);
   display(a, b);

   return 0;
}

void display(int var) {
   cout << "Integer number: " << var << endl;
}

void display(float var) {
   cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Here, the display() function is called three times with different type or number of arguments. The return type of all these functions are same but it's not necessary.

Example 2: Function Overloading

```
// Program to compute absolute value
// Works both for integer and float

#include <iostream.h>
using namespace std;

int absolute(int);
float absolute(float);

int main() {
   int a = -5;
   float b = 5.5;
   cout << "Absolute value of " << a << " = " << absolute(a) << endl;
   cout << "Absolute value of " << b << " = " << absolute(b);
   return 0;
}

int absolute(int var) {
    if (var < 0)
        var = -var;
   return var;
}

float absolute(float var){
   if (var < 0.0)
       var = -var;
   return var;
}
```

Output

```
Absolute value of -5 = 5
Absolute value of 5.5 = 5.5
```

In the above example, two functions absolute() are overloaded.
Both functions take single argument. However, one function takes integer as an argument and other takes float as an argument.
When absolute() function is called with integer as an argument, this function is called:

```
int absolute(int var) {
    if (var < 0)
        var = -var;
   return var;
}
```

When absolute() function is called with float as an argument, this function is called:

```
float absolute(float var){
   if (var < 0.0)
       var = -var;
   return var;
}
```

C++ Friend Functions

WHAT IS FRIEND FUNCTION?

- The functions can be defined somewhere in the program just like a normal function but friend function is defined outside the class but it can still access all the private and protected members of the class.
- If we define a function as a friend function, then the protected and private data of a class can be accessed using the function.
- The friend can be a function, a function template or member function, a class or a class template where the entire class and all the members of the class are friends.
- It is easy to declare the friend function. The keyword friend is used while declaring the friend function.

SYNATX FOR DECLARING THE FRIEND FUNCTION:

```
class class_name
{
friend data_type function_name()  ;
…
}
#include <iostream.h>
using namespace std;
class Shape {
   double area;
   public:
      friend void printArea( Shape shape ); // friend function
      void setArea( double ar );
};
// member function
void Shape::setArea( double ar ) {
   area = ar;
}

// printArea() is not a member function of any class.
void printArea( Shape shape ) {
   /* Because printArea() is a friend of Shape, it can directly access any member of this class
*/
   cout << "Area of shape : " << shape.area <<endl;
}
 // Main function
int main() {
   Shape shape;
    // set shape area without member function
  shape.setArea(20.0);
    // using friend function to print the area.
   printArea( shape );
   return 0;
}
```

What is inline function?

The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called. Compiler replaces the definition of inline functions at compile time instead of referring function definition at runtime.
NOTE- This is just a suggestion to compiler to make the function inline, if function is big (in term of executable instruction etc) then, compiler can ignore the “inline” request and treat the function as normal function.
The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
    // function code
}
```

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:
1) If a function contains a loop. (for, while, 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 function body.
5) If a function contains switch or goto statement.

How to make function inline:
To make any function as inline, start its definitions with the keyword "inline".

Example –

```
Class A
{
 Public:
    inline int add(int a, int b)
    {
       return (a + b);
    }
};

Class A
{
 Public:
    int add(int a, int b);
};

inline int A::add(int a, int b)
{
   return (a + b);
}
```

Why to use –
In many places we create the functions for small work/functionality which contain simple and less number of executable instruction. Imagine their calling overhead each time they are being called by callers.

When a normal function call instruction is encountered, the program stores the memory address of the instructions immediately following the function call statement, loads the function being called into the memory, copies argument values, jumps to the memory location of the called function, executes the function codes, stores the return value of the function, and then jumps back to the address of the instruction that was saved just before executing the called function. Too much run time overhead.

The C++ inline function provides an alternative. With inline keyword, the compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code. Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program.

With below pros, cons and performance analysis, you will be able to understand the “why” for inline keyword

Pros (Advantages) :-
(1) It speeds up your program by avoiding function calling overhead.
(2) It save overhead of variables push/pop on the stack, when function calling happens.
(3) It save overhead of return call from a function.
(4) It increases locality of reference by utilizing instruction cache.
(5) By marking it as inline, you can put a function definition in a header file (i.e. it can be included in multiple compilation unit, without the linker complaining)
(6) Function call overhead doesn’t occur.
(7) It also saves the overhead of push/pop variables on the stack when function is called.
(8) It also saves overhead of a return call from a function.
(9) When you inline a function, you may enable compiler to perform context specific optimization on the body of function.
(10) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Cons (Disadvantages):-
(1) It increases the executable size due to code expansion.
(2) C++ in lining is resolved at compile time. Which means if you change the code of the in lined function, you would need to recompile all the code using it to make sure it will be updated.
(3) When used in a header, it makes your header file larger with information which users don’t care.
(4) As mentioned above it increases the executable size, which may cause thrashing in memory. More number of page fault bringing down your program performance.
(5) Sometimes not useful for example in embedded system where large executable size is not preferred at all due to memory constraints.
(6) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.
(7) Inline functions might cause thrashing because in lining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

When to use -
Function can be made as inline as per programmer need. Some useful recommendation are mentioned below-
1. Use inline function when performance is needed.
2. Use inline function over macros.
3. Prefer to use inline keyword outside the class with the function definition to hide implementation details.

Key Points -
1. It’s just a suggestion not compulsion. Compiler may or may not inline the functions you marked as inline. It may also decide to inline functions not marked as inline at compilation or linking time.
2. Inline works like a copy/paste controlled by the compiler, which is quite different from a

pre-processor macro: The macro will be forcibly in lined, will pollute all the namespaces and code, won't be easy to debug.
3. All the member function declared and defined within class are Inline by default. So no need to define explicitly.
4. Virtual methods are not supposed to be inlinable. Still, sometimes, when the compiler can know for sure the type of the object (i.e. the object was declared and constructed inside the same function body), even a virtual function will be in lined because the compiler knows exactly the type of the object.
5. Template methods/functions are not always in lined (their presence in an header will not make them automatically inline).

The following program demonstrates the use of use of inline function.

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

Inline function and classes:

It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword. For example:

```
class S
{
public:
   inline int square(int s) // redundant use of inline
   {
      // this function is automatically inline
      // function body
   }
};
```

The above style is considered as a bad programming style. The best programming style is to just write the prototype of function inside the class and specify it as an inline in the function definition.
For example:

```
class S
{
public:
   int square(int s); // declare the function
};

inline int S::square(int s) // use inline prefix
```

```
{

}
```

The following program demonstrates this concept:

```
#include <iostream.h>
using namespace std;
class operation
{
   int a,b,add,sub,mul;
   float div;
public:
   void get();
   void sum();
   void difference();
   void product();
   void division();
};
inline void operation :: get()
{
   cout << "Enter first value:";
   cin >> a;
   cout << "Enter second value:";
   cin >> b;
}

inline void operation :: sum()
{
   add = a+b;
   cout << "Addition of two numbers: " << a+b << "\n";
}

inline void operation :: difference()
{
   sub = a-b;
   cout << "Difference of two numbers: " << a-b << "\n";
}

inline void operation :: product()
{
   mul = a*b;
   cout << "Product of two numbers: " << a*b << "\n";
}

inline void operation ::division()
{
   div=a/b;
   cout<<"Division of two numbers: "<<a/b<<"\n" ;
}
```

```
int main()
{
   cout << "Program using inline function\n";
   operation s;
   s.get();
   s.sum();
   s.difference();
   s.product();
   s.division();
   return 0;
}
```

Output:

```
Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3
```

Virtual Function in C++

A virtual function a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

```
// CPP program to illustrate
// concept of Virtual Functions
#include<iostream.h>
using namespace std;

class base
{
public:
   virtual void print ()
   { cout<< "print base class" <<endl; }

   void show ()
   { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
   void print ()
   { cout<< "print derived class" <<endl; }

   void show ()
   { cout<< "show derived class" <<endl; }
};

int main()
{
   base *bptr;
   derived d;
   bptr = &d;

   //virtual function, binded at runtime
   bptr->print();

   // Non-virtual function, binded at compile time
   bptr->show();
}
```

Output:

```
print derived class
show base class
```

Why Inheritance?

Reusability is one of the important characteristics of Object Oriented Programming (OOP). Instead of trying to write programs repeatedly, using existing code is a good practice for the programmer to reduce development time and avoid mistakes. In C++, reusability is possible by using Inheritance.

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

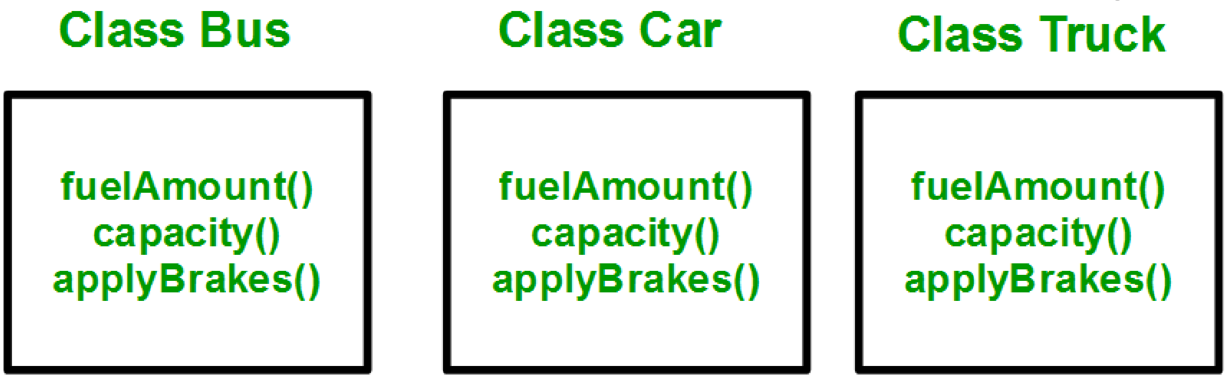
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

The article is divided into following subtopics:

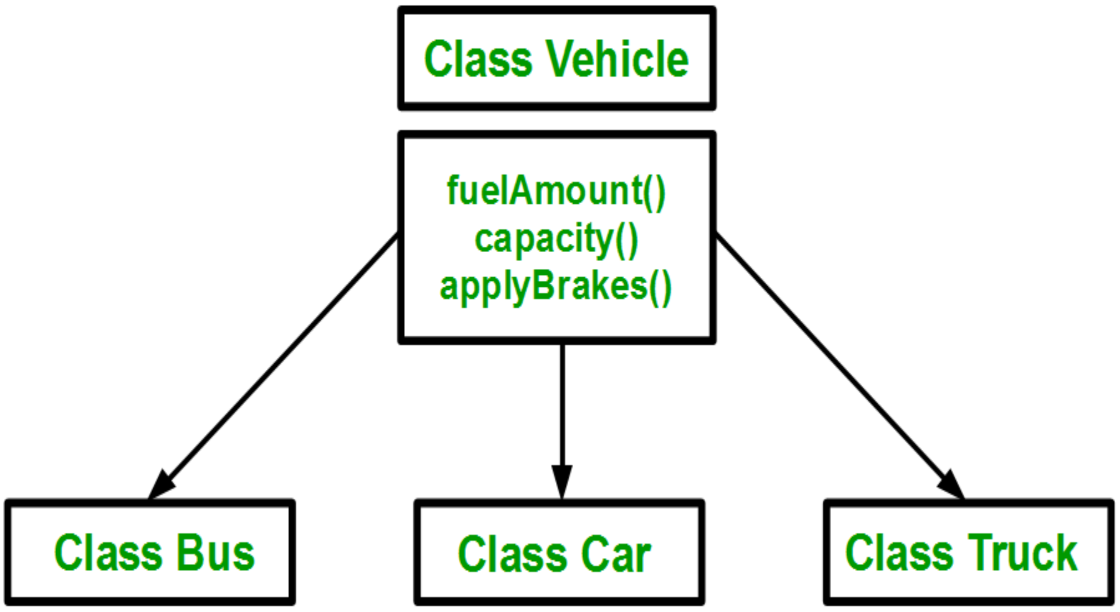
1. Why and when to use inheritance?
2. Modes of Inheritance
3. Types of Inheritance

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



You can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

What is Inheritance?

The technique of deriving a new class from an old one is called inheritance. The old class is referred to as base class and the new class is referred to as derived class or subclass. Inheritance concept allows programmers to define a class in terms of another class, which makes creating and maintaining application easier. When writing a new class, instead of writing new data member and member functions all over again, programmers can make a bonding of the new class with the old one that the new class should inherit the members of the existing class. A class can get derived from one or more classes, which means it can inherit data and functions from multiple base classes.

syntax:

```
class derived-class: visibility-mode base-class
```

Visibility mode is used in the inheritance of C++ to show or relate how base classes are viewed with respect to derived class. When one class gets inherited from another, visibility mode is used to inherit all the public and protected members of the base class. Private members never get inherited and hence do not take part in visibility. By default, visibility mode remains "private".

What are Base class and Derived class?

The existing class from which the derived class gets inherited is known as the base class. It acts as a parent for its child class and all its properties i.e. public and protected members get inherited to its derived class.

A derived class can be defined by specifying its relationship with the base class in addition to its own details, i.e. members.

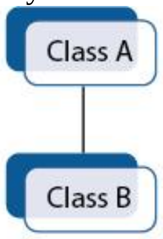
<u>The general form of defining a derived class is:</u>

```
class derived-class_name : visivility-mode base-class_name
{
 . . . .  // members of the derived class
 . . . .
};
```

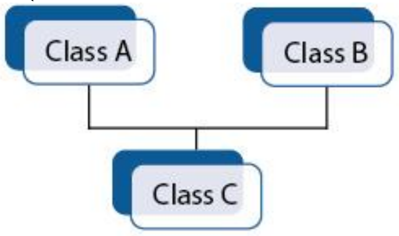
Forms of Inheritance

<u>C++ offers five types of Inheritance. They are:</u>

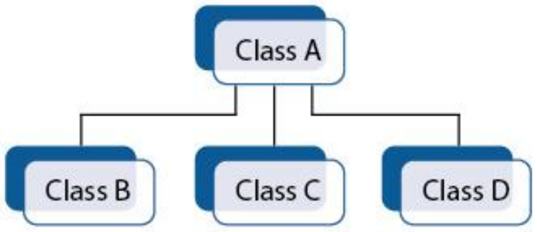
- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance (also known as Virtual Inheritance)



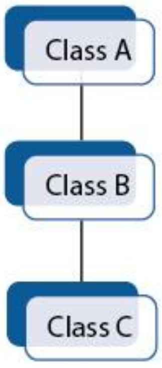
Single Inheritance



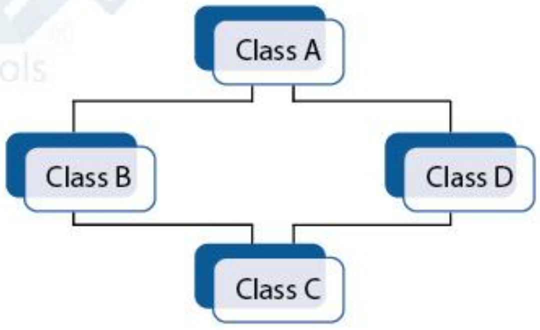
Multiple Inheritance



Hierarchical Inheritance



Multilevel Inheritance



Hybrid Inheritance

Single Inheritance

In single inheritance, there is only one base class and one derived class. The Derived class gets inherited from its base class. This is the simplest form of inheritance. In the above figure, fig(a) is the diagram for single inheritance.

Multiple Inheritance

In this type of inheritance, a single derived class may inherit from two or more base classes. In the above list of figures, fig(b) is the structure of Multiple Inheritance.

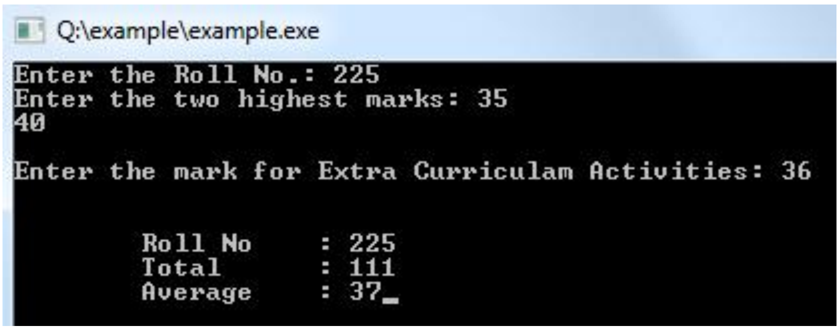
Program for Multiple Inheritance:

Example:

```
#include <iostream.h>
using namespace std;

class stud {
protected:
   int roll, m1, m2;

public:
   void get()
   {
      cout << "Enter the Roll No.: "; cin >> roll;
      cout << "Enter the two highest marks: "; cin >> m1 >> m2;
   }
};
class extracurriculam {
protected:
   int xm;

public:
   void getsm()
   {
      cout << "\nEnter the mark for Extra Curriculam Activities: "; cin >> xm;
   }
};
class output : public stud, public extracurriculam {
   int tot, avg;

public:
   void display()
   {
      tot = (m1 + m2 + xm);
      avg = tot / 3;
      cout << "\n\n\tRoll No    : " << roll << "\n\tTotal      : " << tot;
      cout << "\n\tAverage    : " << avg;
   }
};
int main()
```

```
{
   output O;
   O.get();
   O.getsm();
   O.display();
}
```

Output:



```
Q:\example\example.exe
Enter the Roll No.: 225
Enter the two highest marks: 35
40

Enter the mark for Extra Curriculam Activities: 36


        Roll No     : 225
        Total       : 111
        Average     : 37_
```

Hierarchical Inheritance

In this type of inheritance, multiple derived classes get inherited from a single base class. In the above list of figures, fig(c) is the structure of Hierarchical Inheritance.

Syntax:

```
class base_classname {
   properties;
   methods;
};
class derived_class1 : visibility_mode base_classname {
   properties;
   methods;
};
  class derived_class2 : visibility_mode base_classname {
   properties;
   methods;
};
   ... ... ...
   ... ... ...
  class derived_classN : visibility_mode base_classname {
   properties;
   methods;
};
```

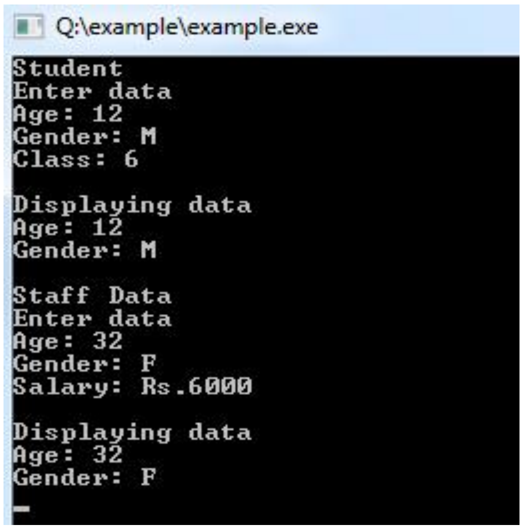
Program for Hierarchical Inheritance

Example:

```
#include <iostream.h>
#include <string.h>
using namespace std;
class member {
   char gender[10];
```

```
    int age;

public:
    void get()
    {
        cout << "Age: "; cin >> age;
        cout << "Gender: "; cin >> gender;
    }
    void disp()
    {
        cout << "Age: " << age << endl;
        cout << "Gender: " << gender << endl;
    }
};
class stud : public member {
    char level[20];

public:
    void getdata()
    {
        member::get();
        cout << "Class: "; cin >> level;
    }
    void disp2()
    {
        member::disp();
        cout << "Level: " << level << endl;
    }
};
class staff : public member {
    float salary;

public:
    void getdata()
    {
        member::get();
        cout << "Salary: Rs."; cin >> salary;
    }
    void disp3()
    {
        member::disp();
        cout << "Salary: Rs." << salary << endl;
    }
};
int main()
{
    member M;
    staff S;
    stud s;
    cout << "Student" << endl;
    cout << "Enter data" << endl;
    s.getdata();
    cout << endl
        << "Displaying data" << endl;
    s.disp();
    cout << endl
        << "Staff Data" << endl;
    cout << "Enter data" << endl;
    S.getdata();
    cout << endl
        << "Displaying data" << endl;
    S.disp();
}
```

Output:



Q:\example\example.exe

```
Student
Enter data
Age: 12
Gender: M
Class: 6

Displaying data
Age: 12
Gender: M

Staff Data
Enter data
Age: 32
Gender: F
Salary: Rs.6000

Displaying data
Age: 32
Gender: F
_
```

Multilevel Inheritance

The classes can also be derived from the classes that are already derived. This type of inheritance is called multilevel inheritance.

Program for Multilevel Inheritance

Example:

```
#include <iostream.h>
using namespace std;

class base {
public:
    void display1()
    {
        cout << "\nBase class content.";
    }
};
class derived : public base {
public:
    void display2()
    {
        cout << "1st derived class content.";
    }
};

class derived2 : public derived {
    void display3()
    {
        cout << "\n2nd Derived class content.";
    }
};

int main()
{
    derived2 D;
    //D.display3();
    D.display2();
    D.display1();
}
```

Hybrid Inheritance

This is a Mixture of two or More Inheritance and in this Inheritance, a Code May Contains two or Three types of inheritance in Single Code. In the above figure, the fig(5) is the diagram for Hybrid inheritance.

Constructor and Destructor in Inheritance

Invocation of constructors and destructors depends on the type of inheritance being implemented. We have presented you the sequence in which constructors and destructors get called in single and multiple inheritance.

1. **Constructor and destructor in single inheritance**

- Base class constructors are called first and the derived class constructors are called next in single inheritance.
- Destructor is called in reverse sequence of constructor invocation i.e. The destructor of the derived class is called first and the destructor of the base is called next.

```
//eg of constructor and destructor in single inheritance
#include<iostream.h>
using namespace std;
class base
{
public:
    base()
    {
        cout<<"base class constructor"<<endl;
    } ~base()
    {
        cout<<"base class destructor"<<endl;
    }
};
```

```
class derived:public base
{
public:
   derived()
   {
      cout<<"derived class constructor"<<endl;
   } ~derived()
   {
      cout<<"derived class destructor"<<endl;
   }
};
int main()
{
   derived d;
   return 0;
}
```

The output illustrates the sequence of call on constructor and destructor in single inheritance.

2. Constructor and destructor in multiple inheritance

- Constructors from all base class are invoked first and the derived class constructor is called.
- Order of constructor invocation depends on the order of how the base is inherited.
- For example:

```
class D: public B, public C
{
   //…
 }
```

Here, B is inherited first, so the constructor of class B is called first and then constructor of class C is called next.

- However, the destructor of derived class is called first and then destructor of the base class which is mentioned in the derived class declaration is called from last towards first in sequentially.

```
//example of constructor and destructor in multiple inheritance
#include<iostream.h>
using namespace std;
class base_one
{
public:
   base_one()
   {
      cout<<"base_one class constructor"<<endl;
   }
   ~base_one()
   {
      cout<<"base_one class destructor"<<endl;
   }
};
class base_two
{
public:
   base_two()
   {
      cout<<"base_two class constructor"<<endl;
   }
   ~base_two()
   {
      cout<<"base_two class destructor"<<endl;
   }
};
class derived:public base_one, public base_two
{
public:
   derived()
   {
      cout<<"derived class constructor"<<endl;
   }
   ~derived()
   {
      cout<<"derived class destructor"<<endl;
   }
};
int main()
{
   derived d;
   return 0;
}
```