CHAPTER-3

Classes and Object:

Defining class with functions and data members

Access specifier: private Vs. public

Creating & deleting objects by using new and delete operators respectively,

Array of Objects, Objects as function argument

Static Data members and member functions

Friend function, friend class

Function with default arguments, function overloading

C++ Classes and Objects

- ✓ Class: The building block of C++ that leads to Object Oriented programming is a Class. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

- ✓ For example consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

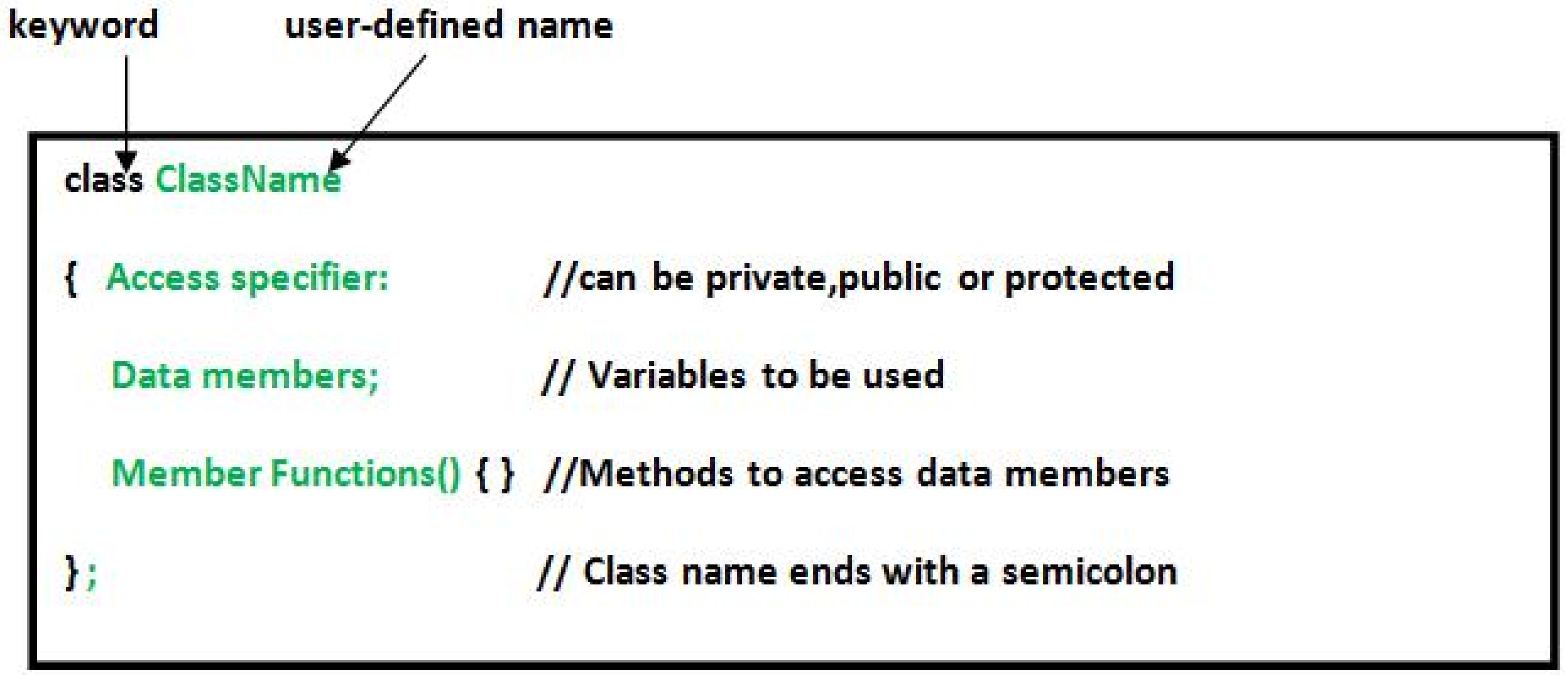
- ✓ A Class is a user defined data-type which have data members and member functions.
- ✓ Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

- ✓ In the above example of class Car, the data member will be speed limit, mileage etc and member functions can be apply brakes, increase speed etc.

- ✓ **An Object is an instance of a Class**. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



```
class ClassName
{  Access specifier:          //can be private,public or protected
   Data members;              // Variables to be used
   Member Functions() { }     //Methods to access data members
};                            // Class name ends with a semicolon
```

Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

```
ClassName ObjectName;
```

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object.

For example if the name of object is obj and you want to access the member function with the name printName() then you will have to write obj.printName() .

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

This access control is given by Access modifiers in C++. There are three access modifiers : public, private and protected.

```
// C++ program to demonstrate accessing of data members
 #include <iostream>
using namespace std;
class Test
{
    // Access specifier
    public:
     // Data Members
    string name;
     // Member Functions()
void printname()
    {
       cout << "my name is " <<name;
    }
};
```

```
int main() {

    // Declare an object of class Test
    Test obj1;

    // accessing data member
    obj1.name = "Abhi";

    // accessing member function
    obj1.printname();
    return 0;
}
```

Output:

```
My name is: Abhi
```

Member Functions in Classes

There are 2 ways to define a member function:

Inside class definition & Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

```
// C++ program to demonstrate function
declaration outside class
 #include <iostream>
using namespace std;
class Test
{
    public:
    string name;
    int id;
void printname();
      // printname is not defined inside class
definition
        // printid is defined inside class definition
    void printid()
    {
        cout << "Geek id is: " << id;
    }
};
```

```
// Definition of printname function using
scope resolution operator ::
void Geeks::printname()
{
    cout << "Geekname is: " << geekname;
}
int main() {
        Geeks obj1;
    obj1.geekname = "xyz";
    obj1.id=15;
        // call printname()
    obj1.printname();
    cout << endl;
        // call printid()
    obj1.printid();
    return 0;
}
```

Output:

```
Geekname is: xyz
Geek id is: 15
```

Note that all the member functions defined inside the class definition are by default inline, but you can also make any non-class function inline by using keyword inline with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.

Note: **Declaring a friend function is a way to give private access to a non-member function.**

Access Modifiers

Access modifiers are used to implement important feature of Object Oriented Programming known as Data Hiding.

Consider a real life example: What happens when a driver applies brakes? The car stops. The driver only knows that to stop the car, he needs to apply the brakes. He is unaware of how actually the car stops. That is how the engine stops working or the internal implementation on the engine side. This is what data hiding is.

Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

Public

Private

Protected

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be Private.

Public: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

```cpp
// C++ program to demonstrate public
// access modifier
 #include<iostream>
using namespace std;
 // class definition
class Circle
{
    public:
        double radius;
        double  compute_area()
        {
            return 3.14*radius*radius;
        }
 };
```

```cpp
 // main function
int main()
{
    Circle obj;

    // accessing public datamember outside class
    obj.radius = 5.5;

    cout << "Radius is:" << obj.radius << "\n";
    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

Output:

```
Radius is:5.5
Area is:94.985
```

Private: The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

```
// C++ program to demonstrate private
// access modifier
 #include<iostream>
using namespace std;
 class Circle
{
    // private data member
    private:
        double radius;
    // public member function
    public:
        double  compute_area()
        {   // member function can access
private
            // data member radius
            return 3.14*radius*radius;
        }
};
```

```
// main function
int main()
{
    // creating object of the class
    Circle obj;
         // trying to access private data member
    // directly outside the class
    obj.radius = 1.5;
         cout << "Area is:" <<
obj.compute_area();
    return 0;
}
```

Output:

```
 In function 'int main()':
11:16: error: 'double Circle::radius' is private
         double radius;                ^
31:9: error: within this context
     obj.radius = 1.5;
```

However we can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

```
// C++ program to demonstrate private
access modifier
 #include<iostream>
using namespace std;
 class Circle
{
    // private data member
    private:
        double radius;
    // public member function
    public:
        double  compute_area(double r)
        {   // member function can access
private
            // data member radius
            radius = r;
            double area = 3.14*radius*radius;
       cout << "Radius is:" << radius <<endl;
cout << "Area is: " << area;
  }
} ;

// main function
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.compute_area(1.5);
    return 0;
}
```

Output:

```
Radius is:1.5
Area is: 7.065
```

Protected: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

```
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;

// base class
class Parent
{
    // protected data members
    protected:
    int id_protected;

};
```

```
// sub class or derived class
class Child : public Parent
{
    public:
    void setId(int id)
    {
        // Child class is able to access the
inherited
        // protected data members of base
class
        id_protected = id;
    }
    void displayId()
    {
        cout << "id_protected is:" <<
id_protected << endl;
    }
};
```

```
// main function
int main() {
    Child obj1;
    // member function of derived class can access the protected data members of base
class
    obj1.setId(81);
    obj1.displayId();
    return 0;
}
```

Output:

```
id_protected is:81
```

Pointers to class members

Just like pointers to normal variables and functions, we can have pointers to class member functions and member variables.

Defining a pointer of class type

We can define pointer of class type, which can be used to point to class objects.

```
class Simple
{
 public:
 int a;
};
int main()
{
 Simple obj;
 Simple* ptr;   // Pointer of class type
 ptr = &obj;
  cout << obj.a;
 cout << ptr->a;  // Accessing member with pointer
}
```

Here you can see that we have declared a pointer of class type which points to class's object. We can access data members and member functions using pointer name with arrow -> symbol.

Lets take an example, to understand how to access member functions using pointers

```
#include <iostream>
using namespace std;
class Data
{
 public:
 int a;
 void print() { cout << "a is "<< a<<endl; }
};
int main()
{
 Data d, *dp;
 dp = &d;     // pointer to object
 dp->a=10;
 dp->print();
 Data *d2=new Data();
 d2->a=20;
 d2->print();
}
```

output:
```
a is 10
a is 20
```

New and delete operators in C++ for dynamic memory

- Dynamic memory allocation in C/C++ refers to performing memory allocation manually by 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 malloc() and calloc() function to allocate memory dynamically at run time and uses 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.

new operator

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

Syntax :

```
pointer-variable = new data-type;
```

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

Example:

```
// 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;
```

Initialize memory: We can also initialize the memory using new operator:

pointer-variable = new data-type(value);

Example:

int *p = new int(25);

float *q = new float(75.25);

Allocate block of memory:

new operator is also used to allocate a block(an array) of memory of type data-type.

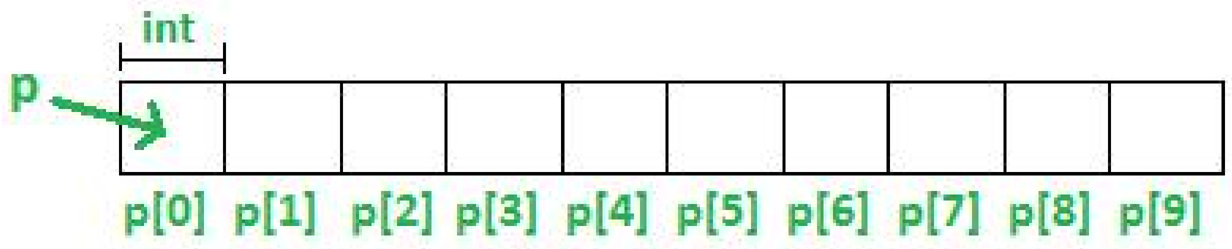
pointer-variable = new data-type[size];

where size specifies the number of elements in an array.

Example:

    int *p = new int[10]

Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc and new operator returns a pointer. Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

Example:

```
int *p = new int;
if (!p)
{
   cout << "Memory allocation failed\n";
}
```

delete operator

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

Syntax:

// Release memory pointed by pointer-variable

delete pointer-variable;

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

Examples:

```
  delete p;
  delete q;
```

To free the dynamically allocated array pointed by pointer-variable, use following form of delete:

// Release block of memory pointed by pointer-variable

delete[] pointer-variable;

Example:

```
   // It will free the entire array pointed by p.
   delete[] p;
```

C++ program to illustrate dynamic allocation and deallocation of memory using new and delete operators

```
#include <iostream>
using namespace std;
int main ()
{
    // Pointer initialization to null
    int* p = NULL;
    // Request memory for the variable
    // using new operator
    p = new int;
    if (!p)
        cout << "allocation of memory 
failed\n";
    else
    {
        // Store value at allocated address
        *p = 29;
        cout << "Value of p: " << *p << endl;
    }
```

```
 // Request block of memory using new 
operator
    float *r = new float(75.25);
    cout << "Value of r: " << *r << endl;
 // Request block of memory of size n
    int n = 5;
    int *q = new int[n];
    if (!p)
        cout << "allocation of memory failed\n";
    else
    {
        for (int i = 0; i < n; i++)
            q[i] = i+1;

        cout << "Value store in block of memory: 
";
        for (int i = 0; i < n; i++)
            cout << q[i] << " ";
    }
    // free the allocated memory
    delete p;
    delete r;

    // free the block of allocated memory
    delete[] q;

    return 0;
}
```
Output:
```
Value of p: 29
Value of r: 75.25
Value store in block of memory: 1 2 3 4 5
```

C++ Array of Objects

An object of class represents a single record in memory, if we want more than one record of class type, we have to create an array of class or object. As we know, an array is a collection of similar type, therefore an array can be a collection of class type.

Syntax for Array of objects

```
class class-name
{
      datatype var1;
      datatype var2;
      - - - - - - - - - - -
      datatype varN;

      method1();
      method2();
      - - - - - - - - - - -
      methodN();
};
class-name obj[ size ];
```

Example for Array of object

```
    #include<iostream.h>
    #include<conio.h>
    class Employee
    {
        int Id;
        char Name[25];
        int Age;
        long Salary;
public:
        void GetData()
        {
     cout<<"\n\tEnter Employee Id : ";
            cin>>Id;
     cout<<"\n\tEnter Employee Name : ";
            cin>>Name;
      cout<<"\n\tEnter Employee Age : ";
            cin>>Age;
      cout<<"\n\tEnter Employee Salary : ";
            cin>>Salary;
        }
void PutData()
{
cout<<"\n"<<Id<<"\t"<<Name<<"\t"
<<Age<<"\t"<<Salary;
        }
    };
    void main()
    {
        int i;
        Employee E[3];  //Creating Array of 3
Employees
        for(i=0;i<3;i++)
        {
cout<<"\nEnter details of "<<i+1
<<" Employee";
            E[i].GetData();
        }
        cout<<"\nDetails of Employees";
        for(i=0;i<3;i++)
        E[i].PutData();
    }
```

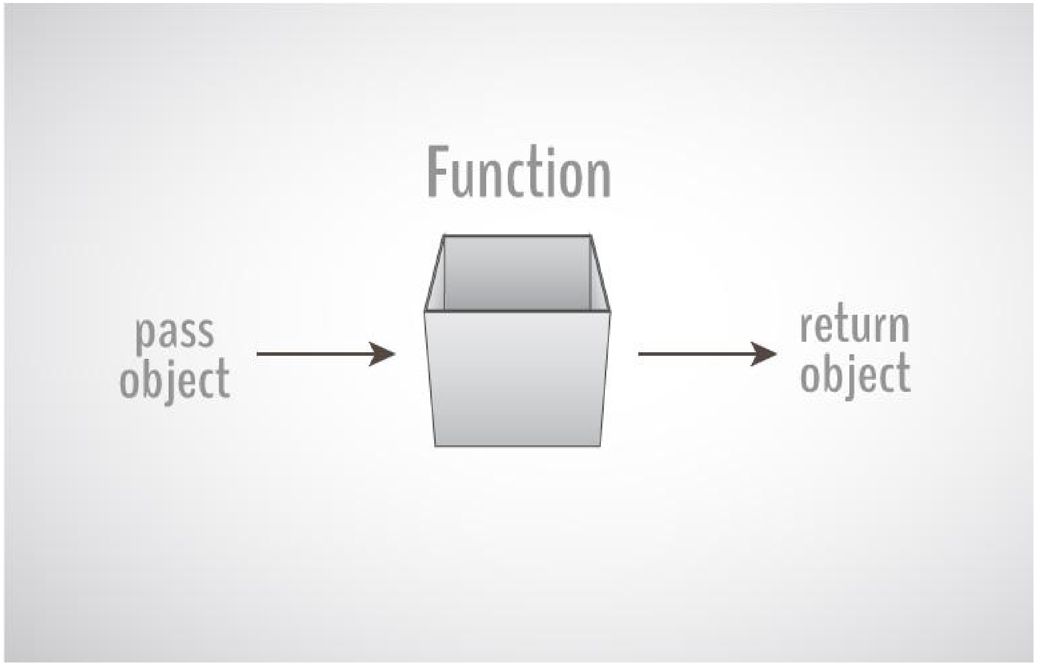
Output :

```
Enter details of 1 Employee
        Enter Employee Id : 101
        Enter Employee Name : Suresh
        Enter Employee Age : 29
        Enter Employee Salary : 45000

Enter details of 2 Employee
        Enter Employee Id : 102
        Enter Employee Name : Mukesh
        Enter Employee Age : 31
        Enter Employee Salary : 51000

Enter details of 3 Employee
        Enter Employee Id : 103
        Enter Employee Name : Ramesh
        Enter Employee Age : 28
        Enter Employee Salary : 47000
```

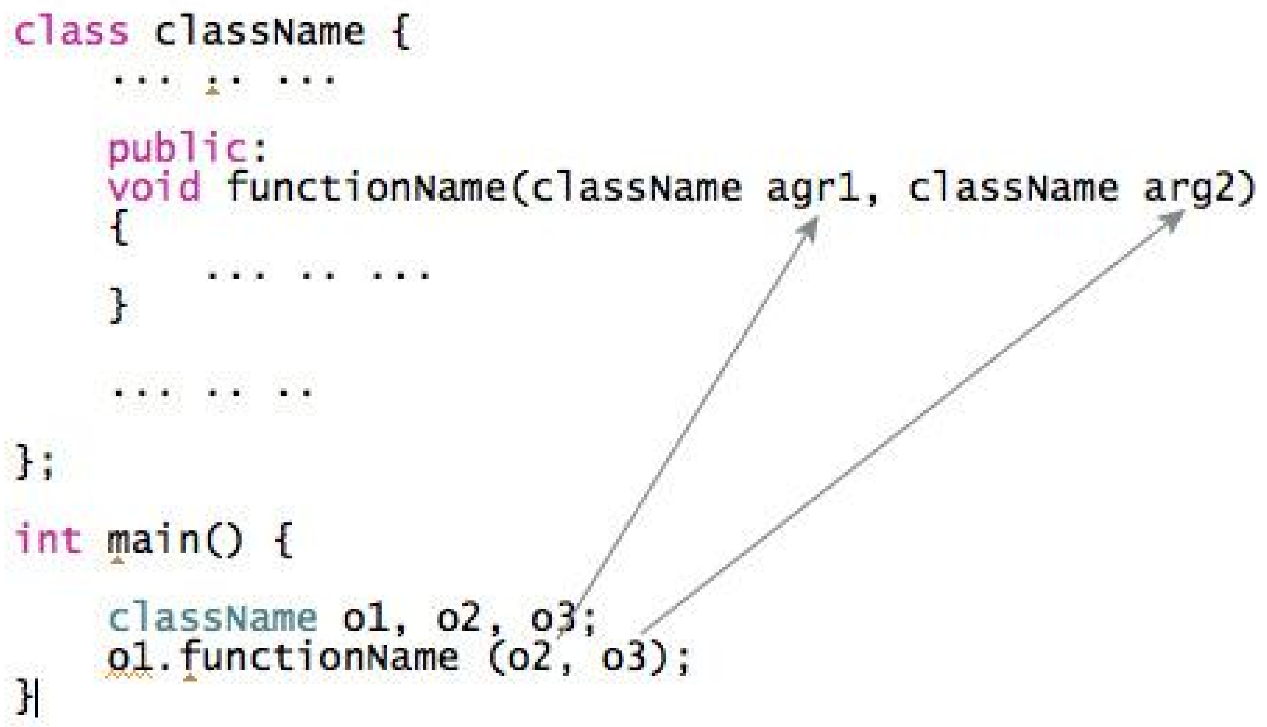
```
Details of Employees
                    101     Suresh     29     45000
                    102     Mukesh      31     51000
                    103     Ramesh      28      47000
```

In the above example, we are getting and displaying the data of 3 employee using array of object. Statement 1 is creating an array of Employee Emp to store the records of 3 employees.

How to pass and return object from a function in C++?



How to pass objects to a function?



```
class className {
    ... .. ...

    public:
    void functionName(className agr1, className arg2)
    {
        ... .. ...
    }

    ... .. ..

};

int main() {

    className o1, o2, o3;
    o1.functionName (o2, o3);
}
```

Example: Pass Objects to Function

C++ program to add two complex numbers by passing objects to a function.

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

class Complex
{
    private:
       int real;
       int imag;
    public:
       Complex(): real(0), imag(0) { }
       void readData()
        {
           cout << "Enter real and imaginary number respectively:"<<endl;
           cin >> real >> imag;
        }
```

```
 void addComplexNumbers(Complex comp1, Complex comp2)
        {
// real represents the real data of object c3 because this function is called using code
c3.add(c1,c2);
            real=comp1.real+comp2.real;


 // imag represents the imag data of object c3 because this function is called using code
c3.add(c1,c2);
            imag=comp1.imag+comp2.imag;
        }


        void displaySum()
        {
            cout << "Sum = " << real<< "+" << imag << "i";
        }
};
```

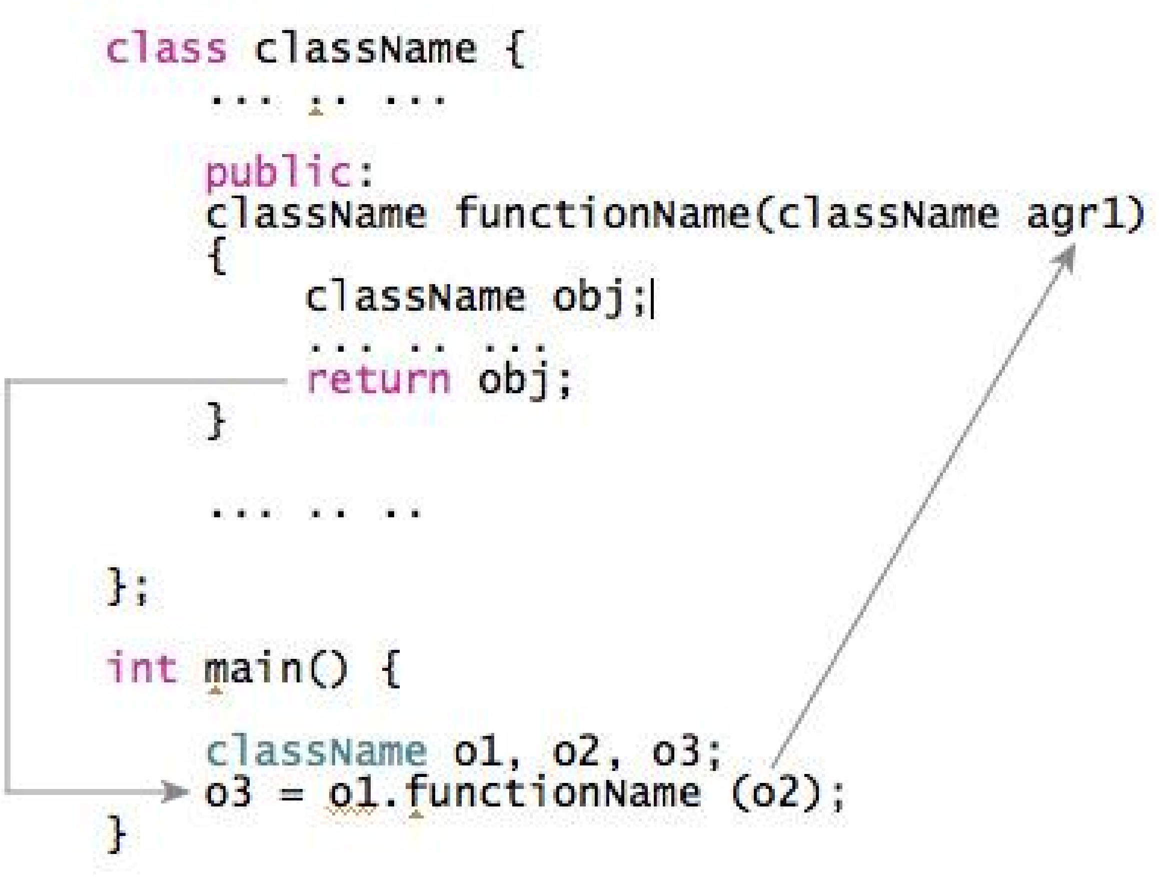
```
int main()
{
    Complex c1,c2,c3;

    c1.readData();
    c2.readData();

    c3.addComplexNumbers(c1, c2);
    c3.displaySum();

    return 0;
}
```

Output

```
Enter real and imaginary number respectively:
2
4
Enter real and imaginary number respectively:
-3
4
Sum = -1+8i
```

How to return an object from the function?

In C++ programming, object can be returned from a function in a similar way as structures.



```
class className {
    ... .. ...

    public:
    className functionName(className agr1)
    {
        className obj;
        ... .. ...
        return obj;
    }

    ... .. ..

};

int main() {

    className o1, o2, o3;
    o3 = o1.functionName (o2);
}
```

Example: Pass and Return Object from the Function

In this program, the sum of complex numbers (object) is returned to the main() function and displayed.

```
#include <iostream>
using namespace std;
class Complex
{
    private:
       int real;
       int imag;
    public:
       Complex(): real(0), imag(0) { }
       void readData()
        {
           cout << "Enter real and imaginary number respectively:"<<endl;
           cin >> real >> imag;
        }
Complex addComplexNumbers(Complex comp2)
        {
            Complex temp;

            // real represents the real data of object c3 because this function is called using
code c3.add(c1,c2);
            temp.real = real+comp2.real;

            // imag represents the imag data of object c3 because this function is called
using code c3.add(c1,c2);
            temp.imag = imag+comp2.imag;
            return temp;
        }
        void displayData()
        {
            cout << "Sum = " << real << "+" << imag << "i";
        }
};
```

```
int main()
{
    Complex c1, c2, c3;

    c1.readData();
    c2.readData();

    c3 = c1.addComplexNumbers(c2);

    c3.displayData();

    return 0;
}
```

Static Keyword

Static keyword has different meanings when used with different types. We can use static keyword with:

Static Variables : Variables inside a function, Variables inside a class

Static Members of Class : Class objects and Functions in a class

Static Variables

Static variables in a Function: When a variable is declared as static, space for it gets allocated for the lifetime of the program. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call. This is useful for implementing coroutines in C/C++ or any other application where previous state of function needs to be stored.

```
// C++ program to demonstrate  the use of static Static  variables in a Function
#include <iostream>
#include <string>
using namespace std;
 void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";
         // value is updated and will be carried to next function calls
    count++;
}
 int main()
{
    for (int i=0; i<5; i++)
        demo();
    return 0;
}
```

Output:

```
0 1 2 3 4
```

You can see in the above program that the variable count is declared as static. So, its value is carried through the function calls. The variable count is not getting initialized for every time the function is called.

Static variables in a class: As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables in a class are not shared by the objects. There cannot be multiple copies of same static variables for different objects. Also because of this reason static variables can not be initialized using constructors.

A static variable inside a class should be initialized explicitly by the user using the class name and scope resolution operator outside the class as shown below:

```
// C++ program to demonstrate static variables inside a class
 #include<iostream>
using namespace std;
class Demo
{
public:
    static int i;
    Demo()
    {
        // Do nothing
    };
};
int Demo::i = 1;
int main()
{
     // prints value of i
    cout << Demo::i; //Accessing static data members using Classname and :: operator
Demo d1;
cout<<d1.i;//you can also access static data member using objects
}
Output: 1
```

Static functions in a class: Just like the static data members or static variables inside the class, static member functions also does not depend on object of class. We are allowed to invoke a static member function using the object and the '.' operator but it is recommended to invoke the static members using the class name and the scope resolution operator.

Static member functions are allowed to access only the static data members or other static member functions, they can not access the non-static data members or member functions of the class.

```
// C++ program to demonstrate static member function in a class
#include<iostream>
using namespace std;
 class Demo
{
   public:
    // static member function
    static void printMsg()
    {
        cout<<"Welcome to Object Oriented Programming";
    }
};
 // main function
int main()
{
    // invoking a static member function
    Demo::printMsg(); // or printMsg();
}
Output:  Welcome to Object Oriented Programming
```

Friend class and function

Friend Class: A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class.

Friend Function: Like friend class, a friend function can be given special grant to access private and protected members.It is a non-member function of a class.

A friend function can be:

a) A method of another class

b) A global function

Following are some important points about friend functions and classes:

1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

2) Friendship is not mutual. If a class A is friend of B, then B doesn’t become friend of A automatically.

3) Friendship is not inherited

4)Friend function is not a member function of the class, therfore they should not be called with objects.

C++ program to demonstrate friend Class

```
#include <iostream>
using namespace std;
class A {
private:
   int a;
public:
   A() { a=0; }
   friend class B;     // Friend Class
};
class B {
private:
   int b;
public:
   void showA(A& x) {
      // Since B is friend of A, it can access
      // private members of A
      cout << "A::a=" << x.a;
   }
};
```

```
int main() {
  A a;
  B b;
  b.showA(a);
  return 0;
}
```

Output:

```
A::a=0
```

C++ program to demonstrate global friend

```
#include <iostream>
 using namespace std;
class A
{
    int a;
public:
    A() {a = 0;}
    friend void showA(A&); // global
friend function
};
void showA(A& x) {
    // Since showA() is a friend, it can
access
    // private members of A
    cout << "A::a=" << x.a;
}
```

```
int main()
{
    A a;
    showA(a);
    return 0;
}
```

Output:

```
A::a = 0
```

C++ program to demonstrate friend function of another class

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

class A
{
public:
    void showB(B& );
};

class B
{
private:
    int b;
public:
    B()  {  b = 0; }
    friend void A::showB(B& x); // Friend function
};
```

```
void A::showB(B &x)
{
    // Since show() is friend of B, it can
    // access private members of B
   cout << "B::b = " << x.b;
}
int main()
{
    A a;
    B x;
    a.showB(x);
    return 0;
}
```

Output:

```
B::b = 0
```

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Example:

```
#include <iostream>
using namespace std;
 class printData {
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }
      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }
      void print(char* c) {
        cout << "Printing character: " << c << 
endl;
      }
};
```

```
int main(void) {
   printData pd;
   // Call print to print integer
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
   // Call print to print character
   pd.print("Hello C++");
   return 0;
}
```

output:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

C++ Programming Default Arguments (Parameters)

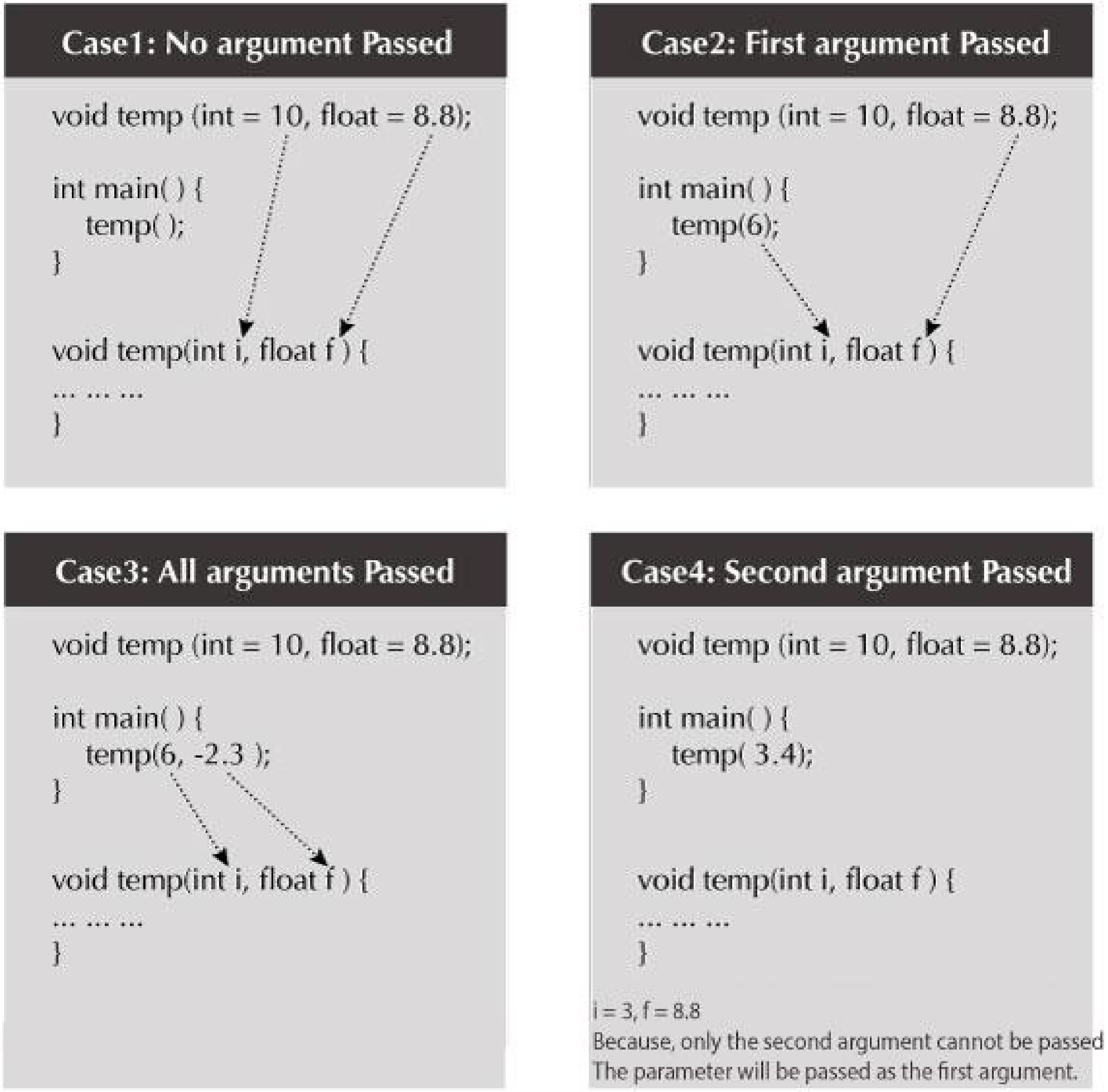
In C++ programming, you can provide default values for function parameters.

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>
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

- 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.

- 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);

- 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.