UNIT - II

Objects and Classes

UNIT - II

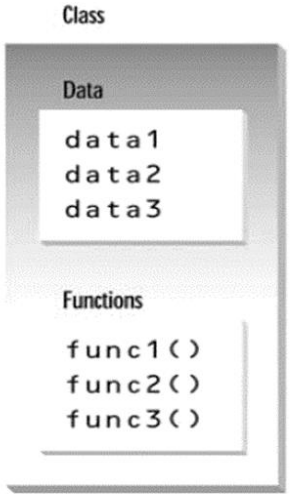
Objects and Classes: Implementation of classes in C++, C++ objects as physical objects, C++ objects as data types, constructors, objects as function arguments, returning object from function, default copy constructor, structures and classes, objects and memory, static class data, const data and classes, mutable. Arrays and String Fundamentals: Arrays as class member data, arrays of objects, strings, standard C++ string class. Templates: Function template, class template

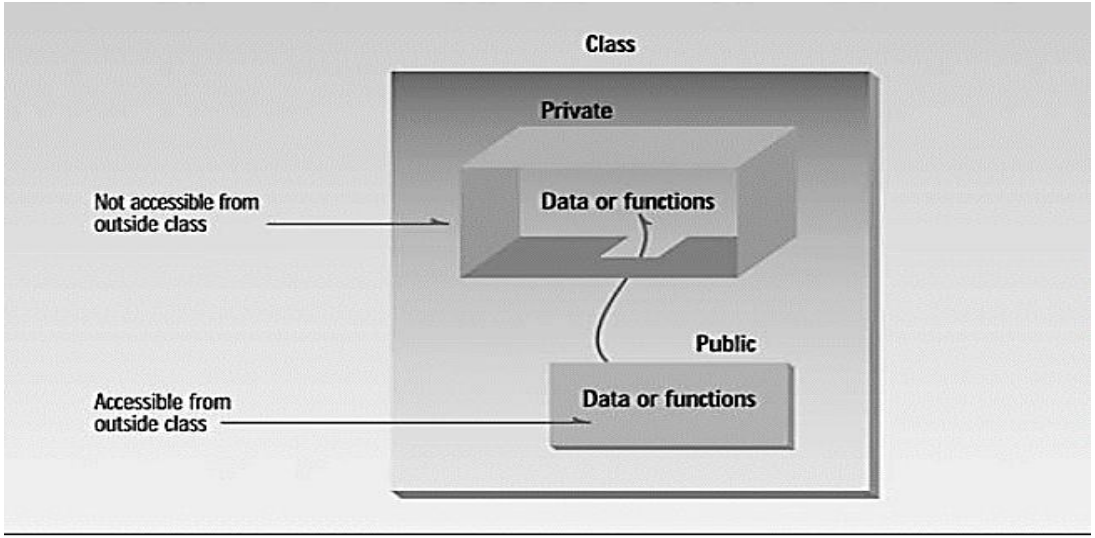
Class & objects

- A class 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 C++ class is like a blueprint for an object.
- An object is said to be an *instance of* a class.
- Classes contain data and functions.

Class & objects

- The data items within a class are called **data members** (or sometimes member data).
- **Member functions** are functions that are included within a class.





Access Specifiers or Modifiers

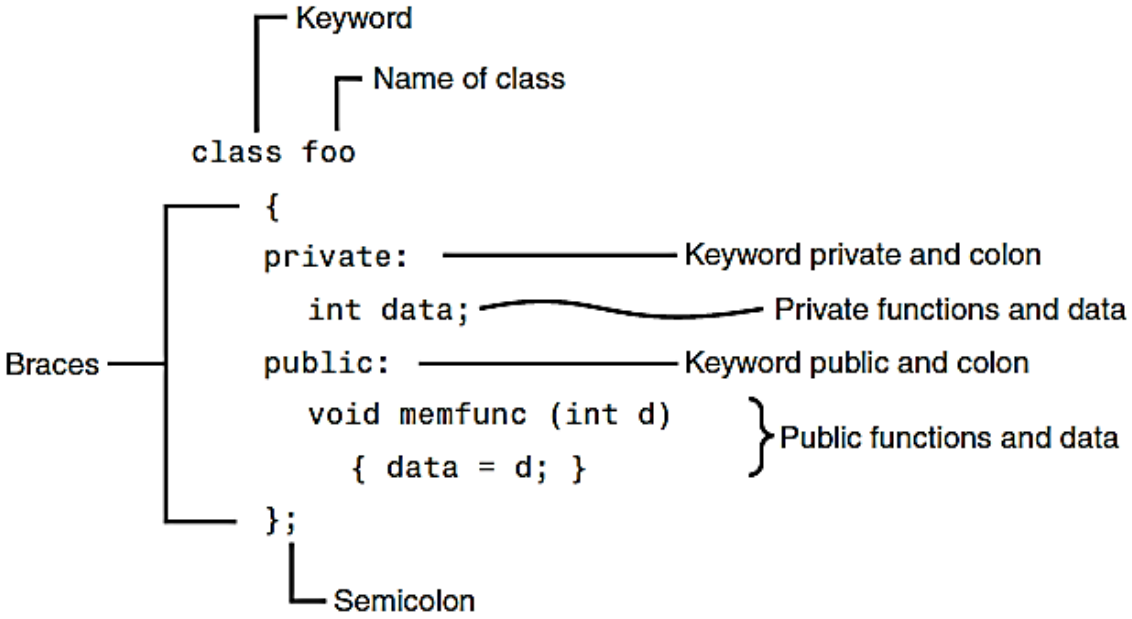
Public: Members are accessible from anywhere in the program.
Private: Members are accessible only within the same class.
Protected: Members are accessible within the same class and derived classes.

```
#include <iostream>
using namespace std;
class Example {
private:
    int x = 10;  // Private: Not accessible in main()
public:
    int y = 20;  // Public: Accessible in main() Anywhere in program
};

int main() {
    Example obj;
    cout << obj.y << endl;  // Accessible (public)
    cout << obj.x << endl;  //  Error: 'x' is private
    return 0;
}
```

Class & objects

Syntax of a class definition:



C++ Objects as Physical Objects &C++ Objects as Data Types

- Objects in programs represent physical objects
- C++ object representing a physical object in the real world
- C++ objects can represent: variables of a user-defined data type

Class & objects – Example 1

```
#include <iostream>
using namespace std;
class small     //define a class
{
private:
int s;        //class data
public:
void setdata(int d)     //member function to set data
{
s = d; }
void showdata()     //member function to display data
{
cout << “Data is “ << s << endl; }
};
```

```
int main()
{
small s1, s2;     //define two objects of class small
s1.setdata(1066); //call member function to set data
s2.setdata(1776);
s1.showdata(); //call member function to display data
s2.showdata();
return 0;
s1.s=2013; //Error Private
s2.s=4096;// Error Private
}
```

Class & objects – Example 2

```
#include <iostream>
using namespace std;
class part    //define class
{
private:
int modelnumber;
int partnumber;
float cost;
public:
void setpart(int mn, int pn, float c)
{
modelnumber = mn;
partnumber = pn;
cost = c;
}
```

```
void showpart() {
cout << “Model “ << modelnumber;
cout << “, part “ << partnumber;
cout << “, costs $” << cost << endl;
}
};

int main()
{
part part1;
part1.setpart(6244, 373, 217.55F);
part1.showpart();
return 0;
}
```

Example 3(Distance Units in English System)


```
#include <iostream>
using namespace std;
class Distance {
private:
int feet;
float inches;
public:
void setdist(int ft, float in) //set Distance to args
{ feet = ft; inches = in; }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches << '\"'; }
};
int main()
{
Distance dist1, dist2; //define two lengths
dist1.setdist(11, 6.25); //set dist1
dist2.getdist(); //get dist2 from user
//display lengths
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << endl;
return 0;
}
```

Enter feet: 10
Enter inches: 4.75
dist1 = 11'-6.25" ← provided by arguments
dist2 = 10'-4.75" ←input by the user

Constructors

- Constructor in C++ is a special method that is invoked automatically at the time an object of a class is created.
- It is used to initialize the data members of new objects generally.
- The constructor in C++ has the same name as the class or structure.
- It constructs the values i.e. provides data for the object which is why it is known as a constructor.

Types:

- Default Constructor - No parameters. They are used to create an object with default values.
- Parameterized Constructor -Takes parameters. Used to create an object with specific initial values.
- Copy Constructor - Takes a reference to another object of the same class. Used to create a copy of an object.

Constructors

How constructor differs from normal function?

- Constructor has same name as the class itself
- Default Constructors don’t have input argument however, Copy and Parameterized Constructors have input arguments
- Constructors don’t have return type
- A constructor is automatically called when an object is created.
- It must be placed in public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).

Default or Non Parameterized & Parameterized Constructor

```
#include <iostream>
#include <string>
using namespace std;
class Car {
private:
    string make;
    string model;
    int year;
public:
    // Default constructor (initializes with default values)
    Car() {
        make = "Unknown";
        model = "Unknown";
        year = 0;
        cout << "Default constructor called!" << endl;
    }
    // Parameterized constructor (initializes with given values)
    Car(string m, string mo, int y) {
        make = m;
        model = mo;
        year = y;
        cout << "Parameterized constructor called!" << endl;
    }

    void displayDetails() {
        cout << "Car Make: " << make << endl;
        cout << "Car Model: " << model << endl;
        cout << "Car Year: " << year << endl;
    }
};

int main() {
    Car car1;
    car1.displayDetails();
    cout << endl;

    Car car2("Toyota", "Camry", 2022);
    car2.displayDetails();

    return 0;
}
```

Default or Non Parameterized & Parameterized Constructor

```
#include <iostream>
#include <string>
using namespace std;
class Product {
private:
    int productID;
    string productName;
    float price;
public:
    // Non-parameterized constructor
    Product() {
        productID = 0;
        productName = "";
        price = 0.0;
        cout << "Product object created!" << endl;
    }
    // Destructor
    ~Product() {
        cout << "Object destroyed!" << endl;
    }
```

```
    // Member function to input product details
    void inputDetails() {
        cout << "Enter product ID: ";
        cin >> productID;
       cout << "Enter product name: ";
        in >> productName;
        cout << "Enter product price: ";
        cin >> price;
    }
    // Member function to calculate the total cost for a given quantity
    float calculateTotalCost(int quantity) {
        return price * quantity;
    }
    // Member function to display product details
    void displayDetails() {
        cout << "\nProduct Details:" << endl;
        cout << "Product ID: " << productID << endl;
        cout << "Product Name: " << productName << endl;
        cout << "Price: " << price << endl;
    } };
```

Default or Non Parameterized & Parameterized Constructor

```
int main() {

    Product p1;
   p1.inputDetails();
   p1.displayDetails();
    int quantity;
    cout << "\nEnter the quantity: ";
    cin >> quantity;
    float totalCost = p1.calculateTotalCost(quantity);
    cout << "Total cost for " << quantity << " units: " << totalCost << endl;
    return 0;
}
```

Initializer List

- One of the most common tasks a constructor carries out is initializing data members.
- The initialization takes place following the member function declarator but before the function body.
- It's preceded by a colon.
- The value is placed in parentheses following the member data.
- If multiple members must be initialized, they're separated by commas.
- The result is the initializer list (sometimes called by other names, such as the member-initialization list).

Initializer List

```
Product() : productID(0), productName(“”),price(0.0)
{
cout << "Product object created!" << endl;
}
```

someClass() : m1(7), m2(33), m2(4) ← **Initializer list**

```
{ }
```

```
circle(int x, int y, int r, color fc, fstyle fs) : xCo(x), yCo(y), radius(r), fillcolor(fc), fillstyle(fs)
{ }
```

```
circle c1(15, 7, 5, cBLUE, X_FILL);
circle c2(41, 12, 7, cRED, O_FILL);
circle c3(65, 18, 4, cGREEN, MEDIUM_FILL);
```

Initializer List

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

class Product {
public:
    int productID;
    string productName;
    double price;
    // Constructor
    Product() : productID(0), productName(""), price(0.0) {}
    void disp() {
        cout << "ID: " << productID << endl;
        cout << "Name: " << productName << endl;
        cout << "Price: " << price << endl;
    }
};
```

```
int main() {
    Product p;   // Object created
    p.disp();    // Call display
    return 0;
}
```

ID: 0
Name:
Price: 0

Initializer List

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

class Test {
    int x;

public:
    Test() : x(0) {}          // Default
    Test(int a) : x(a) { cout << a; }   // Parameterized
};

int main()
{
    Test t1;      // Calls default constructor
    Test t2(10);  // Calls parameterized constructor
    return 0;
}
```

Output:
10

Default or Non Parameterized & Parameterized Constructor

```
#include <iostream>
using namespace std;
class Counter
{
private:
int count; public:
Counter() : count(0)
{   }
void inc_count()
{ count++; }
int get_count()
{
return count;
}
};
```

```
int main()
{
Counter c1, c2;
 cout << “\nc1=” << c1.get_count();
 cout << “\nc2=” << c2.get_count();
c1.inc_count(); //increment c1
c2.inc_count(); //increment c2
c2.inc_count(); //increment c2
cout << “\nc1=” << c1.get_count();
cout << “\nc2=” << c2.get_count();
cout << endl;
return 0;
}
```

```
c1=0
c2=0
c1=1
c2=2
```

Constructor Overloading

Constructor overloading means having more than one constructor in the same class with different parameter lists, so that objects can be created in different ways.
It is a type of function overloading applied to constructors.

Constructor Overloading

```
#include <iostream>
using namespace std;
class Test {
    int x, y;
public:
    // Default Constructor
    Test() {
        x = 0;
        y = 0;
        cout << "Default Constructor\n";
    }
    // One-parameter Constructor
    Test(int a) {
        x = a;
        y = 0;
        cout << "One Parameter Constructor\n";
    }
    // Two-parameter Constructor
    Test(int a, int b) {
        x = a;
        y = b;
        cout << "Two Parameter Constructor\n";
    }
    void show() {
        cout << "x = " << x << ", y = " << y << endl;
    }
};
int main() {
    Test t1;        // Calls default
    Test t2(5);     // Calls 1-parameter
    Test t3(10,20); // Calls 2-parameter
    t1.show();
    t2.show();
    t3.show();
    return 0;}
```

Constructor Overloading

Default Constructor
One Parameter Constructor
Two Parameter Constructor
x = 0, y = 0
x = 5, y = 0
x = 10, y = 20

Function Overloading


```
#include <iostream>
using namespace std;
class Area {
public:
    // Rectangle
    float area(float l, float b) {
        return l * b;
    }
    // Square
    float area(float s) {
        return s * s;
    }
    // Circle
    double area(double r) {
        return 3.14 * r * r;
    }
};
```

```
int main() {
    Area a;
    cout << "Rectangle = " << a.area(5, 4) << endl;
    cout << "Square    = " << a.area(5) << endl;
    cout << "Circle    = " << a.area(3.0) << endl;

    return 0;
}
```

Constructor Overloading

```
#include <iostream>
using namespace std;
class Area {
    float area;
public:
    // Rectangle
    Area(float l, float b) {
        area = l * b;
    }
    // Square
    Area(float s) {
        area = s * s;
    }
    // Circle
    Area(double r) {
        area = 3.14 * r * r;
    }
    // Display only area
    void disp() {
        cout << area << endl;
    }
};
int main() {
    Area r(5, 4);     // Rectangle
    Area s(5);        // Square
    Area c(3.0);      // Circle
    cout << "Area of Rectangle = ";
    r.disp();
    cout << "Area of Square = ";
    s.disp();
    cout << "Area of Circle = ";
    c.disp();
    return 0;
}
```

Constructor Overloading

Create a class named BankAccount with data members accountNumber, accountHolderName, and balance.
Implement member functions to deposit an amount, withdraw an amount, and display the current balance.
Implement constructor overloading by defining:

- A parameterized constructor that initializes the account with account number, account holder name. This Constructor should initialize the account balance as 0.
- A parameterized constructor that initializes the account with account number, account holder name, and initial balance

Create objects of the class using different constructors and perform deposit and withdrawal operations.

Constructor Overloading

```
#include <iostream>
using namespace std;
class BankAccount {
private:
    int accountNumber;
    string accountHolderName;
    float balance;
public:
BankAccount(int accNo, string name) {
        accountNumber = accNo;
        accountHolderName = name;
        balance = 0.0;
    }
BankAccount(int accNo, string name, float bal) {
        accountNumber = accNo;
        accountHolderName = name;
        balance = bal;
    }
void deposit(float amount) {
          balance += amount;
            cout << "Deposited: " << amount << endl;
}
        void withdraw(float amount) {
        if ( amount <= balance) {
            balance -= amount;
            cout << "Withdrawn: " << amount << endl;
        } else {
            cout << "Invalid withdrawal or insufficient
balance!" << endl;
        }
    }
```

Constructor Overloading

```
void display() {
        cout << "\nAccount Number: " << accountNumber << endl;
        cout << "Account Holder: " << accountHolderName << endl;
        cout << "Current Balance: " << balance << endl;
    }
};
int main() {
    // Object using Constructor 1
    BankAccount acc1(1001, "Ravi");
    // Object using Constructor 2
    BankAccount acc2(1002, "Anita", 5000);
    // Operations on acc1
    acc1.deposit(2000);
    acc1.withdraw(500);
    acc1.display();
```

```
 // Operations on acc2
    acc2.deposit(1000);
    acc2.withdraw(3000);
    acc2.display();

    return 0;
}
```

Destructor

A destructor in C++ is a special member function of a class that is automatically invoked when an object of the class is destroyed (i.e., when the object goes out of scope or is explicitly deleted)

Its main purpose is to release resources that were acquired during the lifetime of the object, such as memory, file handles, or network connections.

It is called automatically when an object is destroyed. This is indeed the case. Such a function is called a destructor. A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde.

Destructor

Characteristics of a Destructor:

- A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence, destructor cannot be overloaded.
- It cannot be declared static or const.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

Destructor

```
class Foo
{
private:
int data;
public:
Foo() : data(0) //constructor (same name as class)
{ }
~Foo() //destructor (same name with tilde)
{ }
};
```

The most common use of destructors is to deallocate memory that was allocated for the object by the constructor

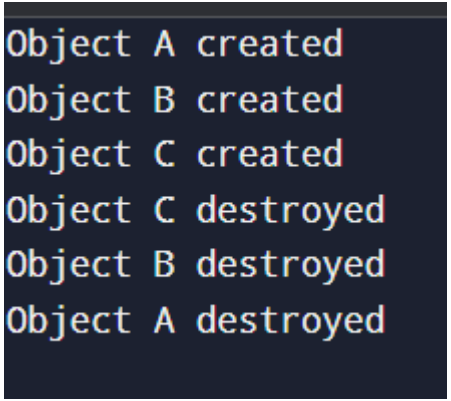
Destructor

```
#include <iostream>
using namespace std;
class Test {
    string name;

public:
    Test(string n) {
        name = n;
        cout << name << " created\n";
    }

    ~Test() {
        cout << name << " destroyed\n";
    }
};
```

```
int main() {
    Test a("Object A");
    Test b("Object B");
    Test c("Object C");
}
```



Destructor

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

class Test {
public:
    Test() {
        cout << "Created\n";
    }
    ~Test() {
        cout << "Destroyed\n";
    }
};

int main() {
    Test a, b, c;
    return 0;
}
```

Default Copy Constructor

We can initialize data members with another object of the same type. No need to create a special constructor for this; one is already built into all classes. It's called the default copy constructor.

It's a one argument constructor whose argument is an object of the same class as the constructor.

Default Copy Constructor

```
#include <iostream>
using namespace std;
 class Distance {
 private:
 int feet;
 float inches;
 public:
 //constructor (no args)
 Distance() : feet(0), inches(0.0)
 { }
 Distance(int ft, float in) : feet(ft), inches(in)
 { }
 void getdist() //get length from user
 {
 cout << "\nEnter feet: "; cin >> feet;
 cout << "Enter inches: "; cin >> inches;
 }
  void showdist() //display distance
  { cout << feet << "\'-" << inches << '\"'; }
  };
int main()
{
Distance dist1(11, 6.25);              //two-arg constructor
Distance dist2(dist1);                 //one-arg constructor
Distance dist3 = dist1;  //also one-arg constructor
//display all lengths
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
return 0;
}
```

dist1 = 11'-6.25"
dist2 = 11'-6.25"
dist3 = 11'-6.25"

Passing Object as arguments


```
#include <iostream>
using namespace std;
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //constructor (no args)
Distance() : feet(0), inches(0.0)
{ }
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches << '\"'; }
void add_dist( Distance, Distance ); //declaration
};
void Distance::add_dist(Distance d2, Distance d3)
{
inches = d2.inches + d3.inches;      //add the inches
feet = 0;
if(inches >= 12.0)                   //if total exceeds 12.0,
{                                    //then decrease inches
inches -= 12.0; //by 12.0 and
feet++;                              //increase feet
} //by 1
feet += d2.feet + d3.feet;           //add the feet
}
```

Passing Object as arguments

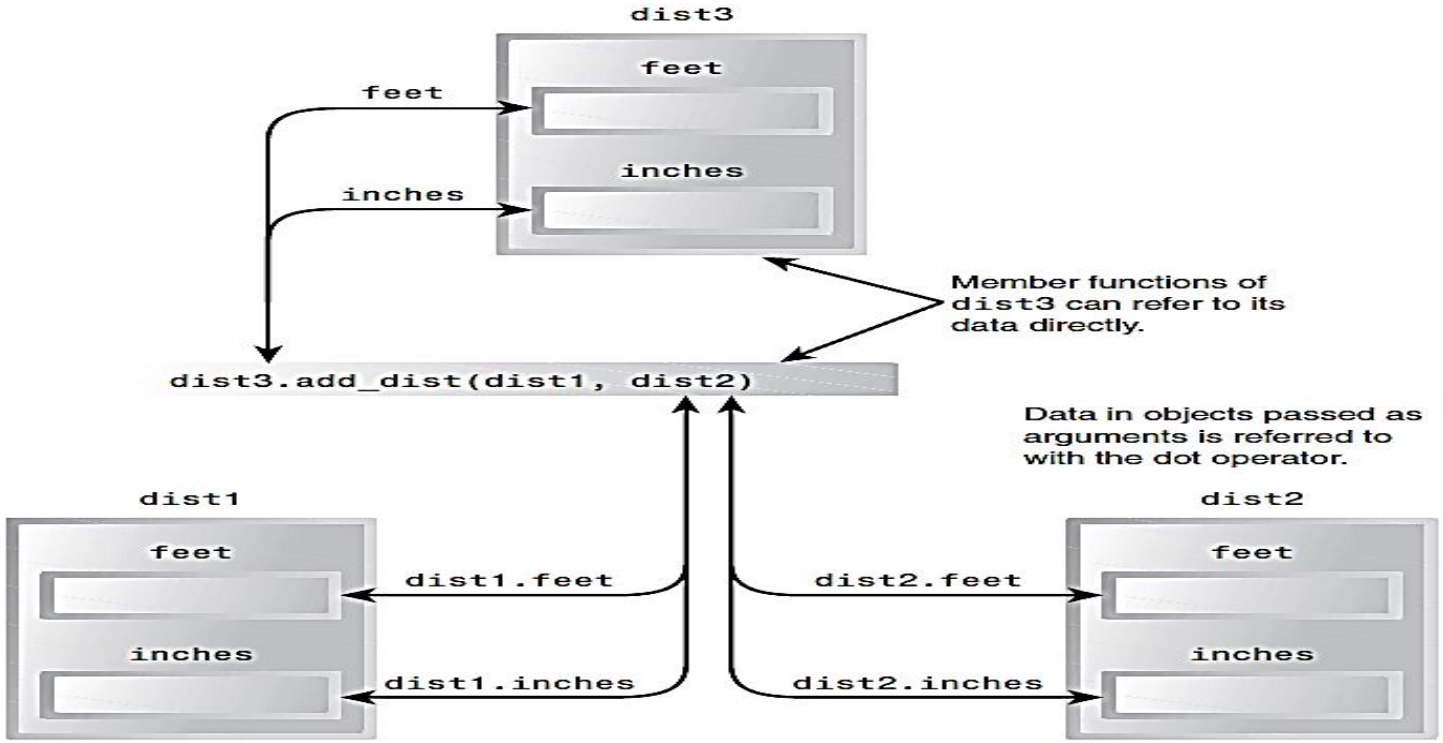

```
#include <iostream>
using namespace std;
class Distance //English Distance class
{
private:
int feet;
float inches;
public: //constructor (no args)
Distance() : feet(0), inches(0.0)
{ }
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist() //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist() //display distance
{ cout << feet << "\'-" << inches << '\"'; }
void add_dist( Distance, Distance ); //declaration
};
void Distance::add_dist(Distance d2, Distance d3)
{
inches = d2.inches + d3.inches;      //add the inches
feet = 0;
if(inches >= 12.0)                   //if total exceeds 12.0,
{                                    //then decrease inches
inches -= 12.0; //by 12.0 and
feet++;                              //increase feet
} //by 1
feet += d2.feet + d3.feet;           //add the feet
}
```

Passing Object as arguments


```
int main()
{
Distance dist1, dist3;                    //define two lengths
Distance dist2(11, 6.25);                 //define and initialize dist2
dist1.getdist();                          //get dist1 from user
dist3.add_dist(dist1, dist2);             //dist3 = dist1 + dist2
//display all lengths
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
return 0;
}
```

Enter feet: 17
Enter inches: 5.75
dist1 = 17'-5.75"
dist2 = 11'-6.25"
dist3 = 29'-0"

Passing Object as arguments



Returning Objects from Functions


```
#include <iostream>
using namespace std;
class Distance {
private:
int feet;
float inches;
public:                                  //constructor (no args)
Distance() : feet(0), inches(0.0)
{ }                                      //constructor (two args)
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist()                           //get length from user
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist()                          //display distance
{ cout << feet << "\'-" << inches << '\"'; }
```

```
Distance add_dist(Distance);             //add
};
Distance Distance::add_dist(Distance d2)
{
Distance temp;                  //temporary variable(or) object
temp.inches = inches + d2.inches;        //add the inches
if(temp.inches >= 12.0)                  //if total exceeds 12.0,
{                                        //then decrease inches
temp.inches -= 12.0; //by 12.0 and
temp.feet = 1;                           //increase feet
} //by 1
temp.feet += feet + d2.feet;             //add the feet
return temp;
}
```

Returning Objects from Functions


```
int main()
{
Distance dist1, dist3;                    //define two lengths
Distance dist2(11, 6.25);                 //define, initialize dist2
dist1.getdist();                          //get dist1 from user
dist3 = dist1.add_dist(dist2);             //dist3 = dist1 + dist2
//display all lengths
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
return 0;
}
```