S J P N Trust's

HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI.

Inculcating Values, Promoting Prosperity

Approved by AICTE, Recognized by Govt. of Karnataka and Permanently Affiliated to VTU Belagavi.

Accredited at 'A' Grade by NAAC

Programmes Accredited by NBA: CSE, ECE, EEE & ME

Subject: Object Oriented Concepts (18CS45)

Module 1: Introduction to Object Oriented Concepts

Dr. Mahesh G. Huddar

Dept. of Computer Science and Engineering

Review of Structures

- There may be cases (when using groups of variables) where the value of one variable may influence the value of other variables logically.
- However, there is no language construct that actually places these variables in the same group. Thus, members of the wrong group may be accidentally sent to the function.
- Arrays could be used to solve this problem but this will not work if the variables are not of the same type.
- **Solution:** Create a data type itself using structures.

Review of Structures

- Structure is the collection of the data members that belong to different data types.
- Structure in C is a programming construct that keeps together the variables that need to be in one entity.
- The keyword **struct** is used to define the structure in C.

Syntax of C Structure

```
struct    name_of_structure
{
    data type member1;
    data type member2;
    data type member3;
    ……..
    data type membern;
};
```

Example

```
struct student
{
   int rollno;
   char name[10];
   int sem;
};
```

Declaration of Structure Variable

1.

```
struct student
{
    int rollno;
    char name[10];
    int sem;
}s1,s2;
```

2.

```
struct student
{
    int rollno;
    char name[10];
    int sem;
};
struct student s1, s2;
```

Accessing Structure Members

- Structure members can be accessed by using the dot operator.

Eg.

s1.rollno , s1.name, s1.sem

s2.rollno, s2.name, s2.sem

Structure Initialization

- Structure Initialization means assigning the value for all the members of the structure.
- It can be done in two ways
  - Initializing the members one by one
  - Initializing the members at once
  - Initializing the members by reading values from keyboard

Initializing members one by one

```
struct student
{
    int rollno;
   char name[10];
   int sem;
};
struct student S1;
S1.rollno = 1;
S1.name = “akash”;
S1.sem = 4;
```

Initializing members at Once

```
struct student
{
    int rollno;
    char name[10];
    int sem;
};
struct student s1 = {1, “akash”, 4};
```

Need for Structure

- The structure is required to bind together all the logically related data members together.
- Ex: when we maintain the record for each student, then each student has his own record of his roll number, name and sem.

Need for Structure

```
#include<stdio.h>
struct student
{
        int rno;
        char name[20];
        float per;
};

struct student s[3];
```

```
int main()
{
        printf("\n Enter the student record");
        for(int i=0; i<3;i++)
        {
                printf ("\n Enter the Roll Number: ");
                scanf ("%d", &s[i].rno);
                printf ("\n Enter Name: ");
                scanf ("%s", &s[i].name);
                printf ("\n Enter the Percentage: ");
                scanf ("%f", &s[i].per);
        }
        printf("\n Student Records are as follows:\n");
         for (int i=0; i<3; i++)
        {
                printf ("\n Roll No: %d", s[i].rno);
                printf ("\n Name: %s", s[i].name);
                printf("\n Percentage is: %f", s[i].per);
        }
}
```

Need for Structure

```
#include<stdio.h>
struct student
{
        int rno;
        char name[20];
        float per;
};

struct student s[3];
```

```
void display(struct student s[5])
{
        for (int i=0; i<3; i++)
        {
                printf ("\n Roll No: %d", s[i].rno);
                printf ("\n Name: %s", s[i].name);
                printf("\n Percentage is: %f", s[i].per);
        }
}
```

```
void input(struct student s[5])
{
        for(int i=0; i<3;i++)
        {
                printf ("\n Enter the Roll Number: ");
                scanf ("%d", &s[i].rno);
                printf ("\n Enter Name: ");
                scanf ("%s", &s[i].name);
                printf ("\n Enter the Percentage: ");
                scanf ("%f", &s[i].per);
        }
}

int main()
{
        printf("\n Enter the student record");
        input(s);
        printf("\n Student Records are as follows:\n");
        display(s);
}
```

Need for Structure

Enter the student record

Enter the Roll Number: 1

Enter Name: mahesh

Enter the Percentage: 80

Enter the Roll Number: 2

Enter Name: Ravi

Enter the Percentage: 70

Enter the Roll Number: 3

Enter Name: Prabhas

Enter the Percentage: 90

Student Records are as follows:

Roll No: 1

Name: mahesh

Percentage is: 80.000000

Roll No: 2

Name: Ravi

Percentage is: 70.000000

Roll No: 3

Name: Prabhas

Percentage is: 90.000000

Overview of C++

- C++ extension was first invented by “Bjarne Stroustrup” in 1979.
- He initially called the new language “ C with Classes”.
- However in 1983 the name was changed to C++.
- c++ is an extension of the C language, in that most C programs are also c++programs.
- C++, as an opposed to C, supports “Object-Oriented Programming”.

Object Oriented Programming System (OOPS)

- It is a collection of objects.
- In OOPS we try to model real-world objects.
- Most real world objects have internal parts (Data Members) and interfaces (Member Functions) that enables us to operate them.
- This is basically the bottom up problem solving approach.

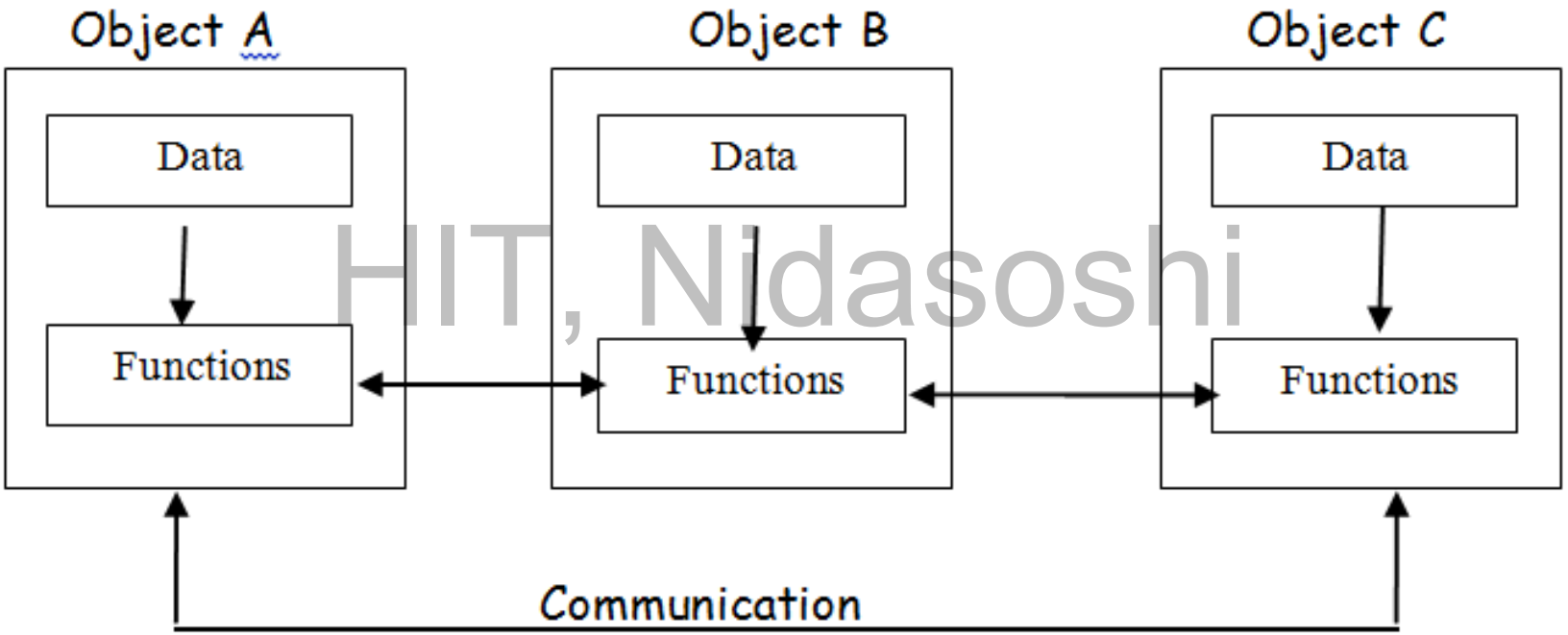
Object Oriented Programming System (OOPS)

- In OOPS we try to model real-world objects.
- Most real world objects have internal parts (Data Members) and interfaces (Member Functions) that enables us to operate them.

Object:

- Everything in the world is an object.
- An object is a collection of variables that hold the data and functions that operate on the data.
- The variables that hold data are called Data Members.
- The functions that operate on the data are called Member Functions.

Object Oriented Programming System (OOPS)



Basic concepts (features) of Object-Oriented Programming

- Objects
- Classes
- Data abstraction
- Data encapsulation
- Inheritance
- Polymorphism
- Binding
- Message passing

Basic concepts (features) of Object-Oriented Programming

- **Objects and Classes:**
  - Classes are user defined data types on which objects are created.
  - Objects with similar properties and methods are grouped together to form class.
  - So class is a collection of objects.
  - Object is an instance of a class.

Basic concepts (features) of Object-Oriented Programming

- **Data abstraction**
  - Abstraction refers to the act of representing essential features without including the background details or explanation.
  - **Ex:** Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Basic concepts (features) of Object-Oriented Programming

Example Data abstraction:

```
#include <iostream>

Using namespace std;

int main( )

{

    cout << "Hello C++" <<endl;

    return 0;

}
```

Basic concepts (features) of Object-Oriented Programming

– Here, you don't need to understand how cout displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.

Basic concepts (features) of Object-Oriented Programming

- **Data encapsulation**
  - Information hiding
  - Wrapping (combining) of data and functions into a single unit (class) is known as data encapsulation.
  - Data is not accessible to the outside world, only those functions which are wrapped in the class can access it.

Basic concepts (features) of Object-Oriented Programming

- **Inheritance**
  - Acquiring qualities.
  - Process of deriving a new class from an existing class.
  - Existing class is known as base, parent or super class.
  - The new class that is formed is called derived class, child or sub class.
  - Derived class has all the features of the base class plus it has some extra features also.
  - Writing reusable code.
  - Objects can inherit characteristics from other objects.

Basic concepts (features) of Object-Oriented Programming

- **Polymorphism**
  - The dictionary meaning of polymorphism is “having multiple forms”.
  - Ability to take more than one form.
  - A single name can have multiple meanings depending on its context.
  - It includes function overloading, operator overloading.

Basic concepts (features) of Object-Oriented Programming

- **Binding**
  - Binding means connecting the function call to the function code to be executed in response to the call.
  - Static binding means that the code associated with the function call is linked at compile time. Also known as early binding or compile time polymorphism.
  - Dynamic binding means that the code associated with the function call is linked at runtime. Also known as late binding or runtime polymorphism.

Basic concepts (features) of Object-Oriented Programming

- **Message passing**
  - Objects communicate with one another by sending and receiving information.

The process of programming in an OOP involves the following basic steps:

- Creating classes that define objects and behavior.
- Creating objects from class definitions.
- Establishing communications among objects.

Difference between POP and OOP

POP	OOP
Emphasis is on procedures (functions)	Emphasis is on data
Programming task is divided into a collection of data structures and functions.	Programming task is divided into classes and objects (consisting of data variables and associated member functions)
Procedures are being separated from data being manipulated	Procedures are not separated from data, instead, procedures and data are combined together.
A piece of code uses the data to perform the specific task	The data uses the piece of code to perform the specific task
Data is moved freely from one function to another function using parameters.	Data is hidden and can be accessed only by member functions not by external function.
Data is not secure	Data is secure
Top-Down approach is used in the program design	Bottom-Up approach is used in program design
Debugging is the difficult as the code size increases	Debugging is easier even if the code size is more

Structure of C++ Program

Include File Section

Class declaration section

Function definition section

Main function definition

Structure of C++ Program

Header File Declaration Section:

Header files used in the program are listed here.

- Header File provides Prototype declaration for different library functions.
- We can also include user define header file.
- Basically all preprocessor directives are written in this section.

Structure of C++ Program

Global declaration section:

- Global Variables are declared here.
- Global Declaration may include
  - Declaring Structure
  - Declaring Class
  - Declaring Variable

Structure of C++ Program

Class declaration section:

- Actually this section can be considered as sub section for the global declaration section.
- Class declaration and all methods of that class are defined here

Structure of C++ Program

- **Main function:**
  - Each and every C++ program always starts with main function.
  - This is entry point for all the function. Each and every method is called indirectly through main.
  - We can create class objects in the main.
  - Operating system calls this function automatically.

Structure of C++ Program

Method definition section

- This is optional section. Generally this method was used in C Programming.

Sample C++ Program

```
#include<iostream>

using namespace std;

int main( )

{

        cout<<"Hello World\n";

}
```

Console I/O - Input

- **Cin: used for keyboard input.**
- **Extraction operator (>>):**
  - To get input from the keyboard we use the extraction operator and the object Cin.
  - No need for "&" in front of the variable.
  - The compiler figures out the type of the variable and reads in the appropriate type.
- Syntax: cin>> variable;
- Example: cin>>n1>>n2;

Example

```
#include<iostream.h>

void main( )

{

        int   x;

        float  y;

        cin>> x>>y;

}
```

Console I/O - Output

- **Cout: used for screen output.**
- **Insertion operator (<<):**
  - To send output to the screen we use the insertion operator on the object Cout.
  - Compiler figures out the type of the object and prints it out appropriately.
  - n1 = 10, n2 = 20
  - printf(“The value of n1 is %d and n2 is %d”,n1,n2);
  - cout<<“The value of n1 is ”<<n1<<“ and n2 is ”<<n2;
  - Syntax: Cout<<variable;

Example:

```
#include<iostream.h>
void main( )
{
        cout<<5;
        cout<<4.1;
        cout<< “string”;
        cout<< “\n”;
}
```

Example using Cin and Cout

```
#include<iostream>
using namespace std;
int main( )
{
        int a,b;
        float k;
        char name[30];
        cout<< "Enter your name \n";
        cin>>name;
        cout<< "Enter two Integers and a Float \n";
        cin>>a>>b>>k;
        cout<< "Thank You," <<name<<",you entered\n";
        cout<<a<<", "<<b<<", and "<<k;
}
```

Example using Cin and Cout

Output:

Enter your name Mahesh

Enter two integers and a Float

10

20

30.5

Thank you Mahesh, you entered

10, 20 and 30.5

C++ program to find out the square of a number

```
#include<iostream>

using namespace std;

int main( )

{

        int i;

        cout<< "Enter a number: ";

        cin>>i;

        cout<<"Square of "<< i <<" is " << i*i<<"\n";

        return 0;

}
```

C++ program to find out the square of a number

Output:

Enter a number: 10

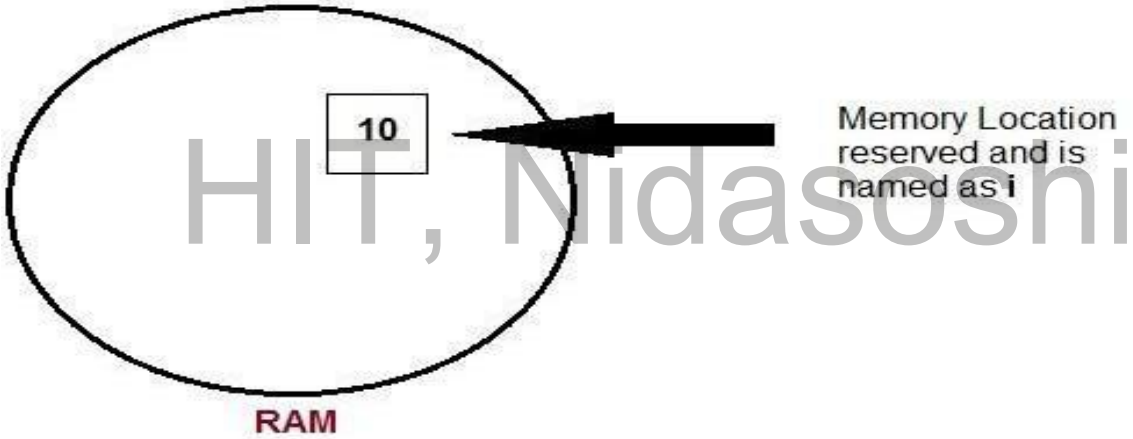
Square of 10 is 100

Variables

- Variable are used in C++, where we need storage for any value, which will change in program.
- Variable can be declared in multiple ways each with different memory requirements and functioning.
- Variable is the name of memory location allocated by the compiler depending upon the data type of the variable.

Variables

Example : `int i=10;` // declared and initialised



Declaration and Initialization

- Variable must be declared before they are used.
- Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

- **Example :**

```
int i;      // declared but not initialized
char c;
int i, j, k;     // Multiple declaration
```

Declaration and Initialization

- **Initialization means assigning valUE to an already declared variable**

```
int i;      // declaration

i = 10;    // initialization
```


- **Initialization and declaration can be done in one single step also,**

```
int i=10;

int i=10, j=11;
```

Scope of Variables

- All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable.
- For most of the cases its between the curly braces, in which variable is declared that a variable exists, not outside it. we can broadly divide variables into two main types,
- **Global Variables**
- **Local variables**

Global variables

- Global variables are those, which are once declared and can be used throughout the lifetime of the program by any class or any function.
- They must be declared outside the main() function.
- If only declared, they can be assigned different values at different time in program lifetime.
- But even if they are declared and initialized at the same time outside the main() function, then also they can be assigned any value at any point in the program.

Global variables

```
#include <iostream>
using namespace std;
int x;     // Global variable declared
int main()
{
        x=10;    // Initialized once
        cout <<"first value of x = "<< x;
        x=20;              // Initialized again
        cout <<"Initialized again with value = "<< x;
}
```

Local Variables

- Local variables are the variables which exist only between the curly braces, in which its declared.
- Outside that they are unavailable and leads to compile time error.

Local Variables

```
#include<iostream>
using namespace std;
int main()
{
        int i=10;
        if(i<20)                    // if condition scope starts
        {
                int n=100;          // Local variable declared and initialized
        }                           // if condition scope ends
        cout << n;                  // Compile time error, n not available here
}
```

Reference variable in C++

- When a variable is declared as reference, it becomes an alternative name for an existing variable.
- A variable can be declared as reference by putting “&” in the declaration.
- int x = 10;

// ref is a reference to x.

- int& ref = x;

Reference variable in C++

```
#include<iostream>
using namespace std;
int main()
{
        int x = 10;          // ref is a reference to x.
        int& ref = x;
        ref = 20;            // Value of x is now changed to 20
        cout << "x = " << x << endl ;   // Value of x is now changed to 30
        x = 30;
        cout << "ref = " << ref << endl ;
        return 0;
}
```

Output:
x = 20
ref = 30

Functions in C++:

- Definition: Dividing the program into modules, these modules are called as functions.
- General form of function:


```
return_type  function_name(parameter list)
{
// Body of the function
}
```

Functions in C++:

Where,

return_type:

- What is the value to be return.
- Function can written any value except array.

Parameter_list: List of variables separated by comma.

- The body of the function(code) is private to that particular function, it cannot be accessed outside the function.

Components of function:

- Function declaration (or) prototype.
- Function parameters (formal parameters)
- Function definition
- Return statement
- Function call

Example:

```
#include<iostream>
using namespace std;
int max(int x, int y);              //prototype(consists of formal arguments)
int main( )             //Function caller
{
    int a, b, c;
    cout<< "Enter 2 integers:"<<endl;
    cin>>a>>b;
    c=max(a,b);      //function call
    cout<<"Maximum Number is: "<<c<<endl;
}
int max(int x, int y)  // function definition
{
    if(x>y)
            return x;  // function return
    else
            return y;
}
```

Output:

```
Enter 2 integers:
20
10
Maximum Number is: 20
```

Function prototype:

- It provides the following information to the compiler.
- The name of the function
- The type of the value returned(default an integer)
- The number and types of the arguments that must be supplied in a call to the function.
- Function prototyping is one of the key improvements added to the C++ functions.
- When a function is encountered, the compiler checks the function call with its prototype so that correct argument types are used.

Function prototype:

- Consider the following statement:

int max(int x, int y);

- It informs the compiler that the function max has 2 arguments of the type integer.

- The function max( ) returns an integer value the compiler knows how many bytes to retrieve and how to interpret the value returned by the function.

Function definition:

- The function itself is returned to as function definition.
- The first line of the function definition is known as function declarator and is followed by function body.
- The declaratory and declaration must use the same function name, number of arguments, the argument type and return type.
- The body of the function is enclosed in braces.


```
int max(int x, int y)   // function definition
{
    if(x>y)
      return x;          // function return
    else
      return y;
}
```

Function call:

c= max (a, b) ;

- Invokes the function max( ) with two integer parameters, executing the call statement causes the control to be transferred to the first statement in the function body and after execution of the function body the control is resumed to the statement following the function call.
- The max( ) returns the maximum of the parameters a and b. the return value is assigned to the local variable c in main( ).

Function parameters:

- The parameters specified in the function call are known as actual parameters and specified in the declarator are known as formal parameters.
- c=max(a,b);
- Here a and b are actual parameters. The parameters x and y are formal parameters. When a function call is made, a one to one correspondence is established between the actual and the formal parameters.
- In this case the value of the variable aa is assigned to the variable x and that of b to y.
- The scope of formal parameters is limited to the function only.

Function return:

- Functions can be grouped into two categories. Functions that do not have a return value(void) and functions that have a return value.
- The statement:
- return x; // function return

and

- return y; //function return
- ex: c=max(a,b); //function call
- the value returned by the function max( ) is assigned to the local variable **c** in main().
- The return statement in a function need not be at the end of the function.
- It can occur anywhere in the function body and as soon as it is encountered , execution control will be returns to the caller.

Argument passing:

Two types

- Call by value
- Call by reference

Call by value:

- The default mechanism of parameter passing(argument passing) is called call by value.
- Here we pass the value of actual arguments to formal parameters.
- Changes made to the formal parameters will not be affected the actual parameters.

Call by value:

- The default mechanism of parameter passing(argument passing) is called call by value.
- Here we pass the value of actual arguments to formal parameters.
- Changes made to the formal parameters will not be affected the actual parameters.

Call by value:

```
//Example 1:
#include<iostream>
using namespace std;
void exchange(int x, int y);
int main( )
{
        int a, b;
        cout<< "enter values for a and b:"<<endl;  // 10 and 20
        cin>>a>>b;
        cout<<"Before Swapping "<<a<<" and "<<b<<endl;    //output: 10, 20
        exchange(a,b);
}
void exchange(int x, int y)
{
        int temp;
        temp=x;
        x=y;
        y=temp;
        cout<<“After Swapping "<<x<<" and "<<y<<endl;                    //output: 20, 10
}
```

Output:

```
Enter values for a and b:
10
20
Before Swapping 10 and 20
After Swapping 20 and 10
```

Call by value:

```
//Example 2:
#include<iostream>
using namespace std;
void add(int a);
int main( )
{
        int a=10, temp;
        add(a);
        cout<<a<<endl;
}
void add(int a)
{
        a=a+a;
        cout<<a<<endl;
}
```

Output:

20
10

Call by reference:

- We pass address of an argument to the formal parameters.
- Changes made to the formal parameters will affect actual arguments.

Call by reference:

```
//Example 1:
#include<iostream>
using namespace std;
void exchange(int &x, int &y);
int main( )
{
        int a, b;
        cout<< "enter values for a and b:"<<endl;  // 10 and 20
        cin>>a>>b;
        cout<<"Before Swapping "<<a<<" and "<<b<<endl;    //output: 10, 20
        exchange(a,b);
        cout<<"After Swapping "<<a<<" and "<<b<<endl;      //output: 20, 10
}
void exchange(int &x, int &y)
{
        int temp;
        temp=x;
        x=y;
        y=temp;
}
```

Output:

```
enter values for a and b:
10
20
Before Swapping 10 and 20
After Swapping 20 and 10
```

Call by reference:

```
//Example 2:
#include<iostream>
using namespace std;
void add (int &a);
int main( )
{
        int a=10;
        add(a);
        cout<<a;
}
void add(int &a)
{
        a=a+a;
}
```

Output:
20

Default arguments

- Default argument is an argument to the function to which the default value is provided.
- Default values are specified when the function is declared.
- The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values.

Default arguments

- A default argument is checked for type at the time of declaration and evaluated at the time of call.
- We must add defaults from right to left.
- We cannot provide a default value to a particular argument in the middle of an argument list.
- Default arguments are useful in situations where some arguments always have the same value.

Default arguments

- **Example:**
- int mul (int i, int j=5, int k=10); //legal.
- int mul (int i=5, int j); //illegal.
- int mul (int i=0,int j, int k=10); //illegal.
- int mul (int i=2, int j=5, int k=10); //legal.

Default arguments

```
//Example:
#include <iostream>
using namespace std;
void add(int a=10, int b=20,int c=30);
int main( )
{
        add(1,2,3);
        add(1,2);
        add(1);
        add( );
}
void add(int a, int b, int c)
{
        cout<< a+b+c <<endl;
}
```

Output:

6

33

51

60

Default arguments

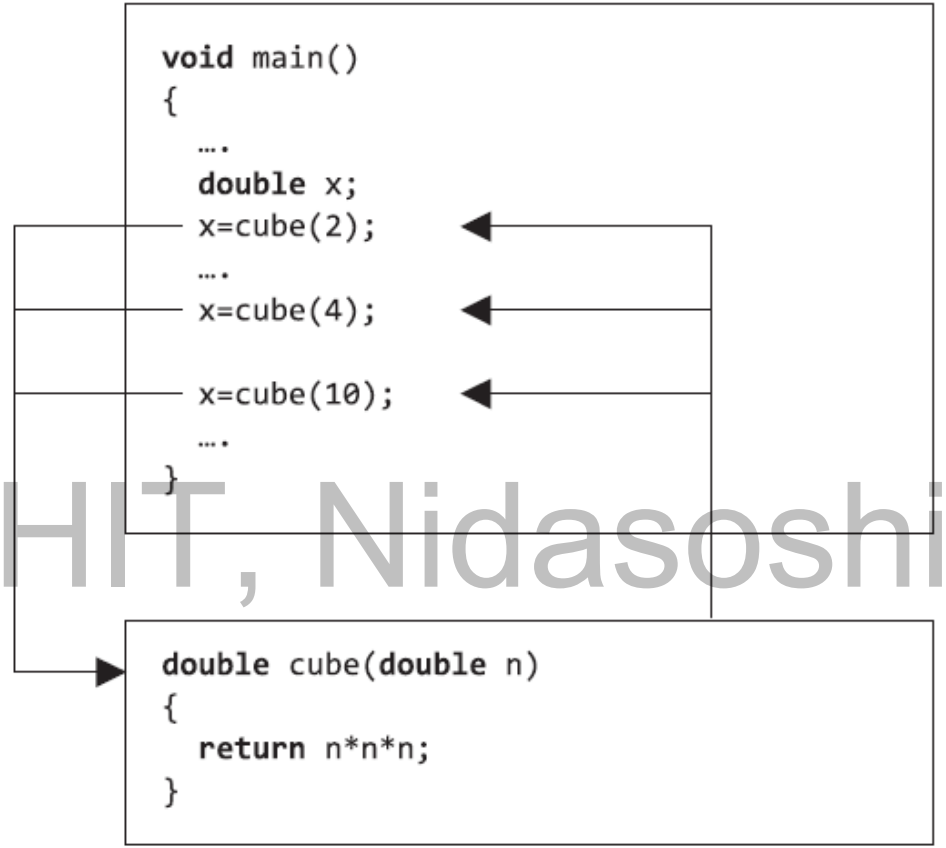
```
//Example:
#include <iostream>
using namespace std;
void add(int a=10, int b=20,int c=30);
int main( )
{
        add(1,2,3);
        add(1,2);
        add(1);
        add( );
}
void add(int a, int b, int c)
{
        cout<< a+b+c <<endl;
}
```

Output:

```
6
33
51
60
```

Inline functions

- In normal function, First control will move from calling to called function.
- Then arguments will be pushed on to the stack, then control will move back to the calling from called function.
- This process takes extra time in executing.
- To avoid this, we use inline function.



The control is transferred to the function definition in case of a non-inline function

Inline functions

- Definition : It is a function whose code is copied in place of each function call.
- When a function is declared as inline, compiler replaces function call with function code.

Note: inline functions are functions consisting of one or two lines of code.

```
void main()
{
  ….
  double x;
  {
    double n;
    n=2;
    x=n*n*n;
  }
  ….
  {
    double n;
    n=4;
    x=n*n*n;
  }
  ….
  {
    double n;
    n=10;
    x=n*n*n;
  }
  ….
}
```

Inline functions

- **Inline function cannot be used in the following situation:**
  - If the function definition is too long or too complicated.
  - If the function is a recursive function.
  - If the function is not returning any value.
  - If the function is having switch statement and goto statement.
  - If the function having looping constructs such as while, for, do-while.
  - If the function has static variables

Inline functions

//Example:

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
   return s*s*s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3) << endl;
   return 0;
}
```

Output:
The cube of 3 is: 27

Introduction to Classes and Objects

- Classes are to C++ what structures are to C.
- Both provide the library programmer a means to create new data types.
- C does not provide the library programmer with the facilities to encapsulate data, to hide data, and to abstract data.

Structures in C++

```
//C++ allows member functions in structures
#include<iostream>
using namespace std;
struct Distance
{
        int iFeet;
        float fInches;
        void setFeet(int x)
        {
                iFeet=x;
        }
        int getFeet()
        {
                return iFeet;
        }
        void setInches(float y)
        {
                fInches=y;
        }
        float getInches()
        {
                return fInches;
        }
};
```

```
int main()
{
        Distance d1,d2;
        d1.setFeet(2);
        d1.setInches(2.2);
        d2.setFeet(3);
        d2.setInches(3.3);
        cout<<d1.getFeet()<<endl<<d1.getInches()<<endl;
        cout<<d2.getFeet()<<endl<<d2.getInches()<<endl;
}
```


Output:

```
2
2.2
3
3.3
```

Structures in C++

- First, we must notice that functions have also been defined within the scope of the structure definition.
- This means that not only the member data of the structure can be accessed through the variables of the structures but also the member functions can be invoked.
- The struct keyword has actually been redefined in C++.

Structures in C++

- Each structure variable contains a separate copy of the member data within itself.
- However, only one copy of the member function exists.
- However, in the above example, note that the member data of structure variables can still be accessed directly.
- The following line of code illustrates this.

```
d1.iFeet=2; //legal!!
```

Private and Public Members

- What is the advantage of having member functions also in structures?
- We have put together the data and functions that work upon the data but we have not been able to give exclusive rights to these functions to work upon the data.
- Problems in code debugging can still arise as before.
- Specifying member functions as public but member data as private obtains the advantage.

Private and Public Members

```
//C++ allows member functions in structures
#include<iostream>
using namespace std;
struct Distance
{           private:
            int iFeet;
            float fInches;
            public:
            void setFeet(int x)
            {           iFeet=x;
            }
            int getFeet()
            {
                        return iFeet;
            }
            void setInches(float y)
            {
                        fInches=y;
            }
            float getInches()
            {
                        return fInches;
            }
};
```

```
int main()
{
            Distance d1,d2;
            d1.setFeet(2);
            d1.setInches(2.2);
            d2.setFeet(3);
            d2.setInches(3.3);
            d1.iFeet++;          //Error, private member accessed by
                                 //non-member function
            cout<<d1.getFeet()<<endl<<d1.getInches()<<endl;
            cout<<d2.getFeet()<<endl<<d2.getInches()<<endl;
}
```

Output:

```
2
2.2
3
3.3
```

Private and Public Members

- The keywords private and public are also known as access modifiers or access specifiers because they control the access to the members of structures.
- C++ introduces a new keyword class as a substitute for the keyword struct.
- *In a structure, members are public by default.*
- On the other hand, *class members are private by default*.
- This is the only difference between the class keyword and the struct keyword.

Class Specification

- A Class is way to bind(combine) the data and its associated functions together. it allows data and functions to be hidden.
- When we define a class, we are creating a new abstract data type that can be created like any other built-in data types.
- This new type is used to declare objects of that class.
- Object is an instance of class.

General form of class declaration is:

class class_name

{

access specifier : data

access specifier : functions;

};

- The keyword class specifies that what follows is an abstract data of type class_name.
- The body of the class is enclosed in braces and terminated by semicolon.

Access Specifiers

- **Private:**
  - Cannot be accessed outside the class.
  - But can be accessed by the member functions of the class.
- **Public:**
  - Allows functions or data to be accessible to other parts of the program.
- **Protected:**
  - Can be accessed when we use inheritance.

Access Specifiers

Note:

- By default data and member functions declared within a class are private.
- Variables declared inside the class are called as data members and functions are called as member functions.
- Only member functions can have access to data members.
- The binding of functions and data together into a single class type variable is referred as ***Encapsulation***.

Defining member functions

- The member functions have some special characteristics:
  - Several different classes can use same function name.
  - Member function can access private data of the class.
  - A member function can call another function directly, without using dot operator.
- Two Ways:
  - We can define the function inside the class.
  - We can define the function outside the class.

Accessing members of class

- Class members (variables (data) and functions) Can be accessed through an object and dot operator.
- Private members can be accessed by the functions which belong to the same class.
- The format for calling a member function is:
  - Object_name.function_name(actual arguments);

Class Example Program

```
//Example:
#include<iostream>
using namespace std;
class student
{
        private:
        char name[10]; // private variables
        int marks1,marks2;
        public:
        void getdata( ) // public function accessing private members
        {
                cout<<"Enter name, marks in two subjects:"<<endl;
                cin>>name>>marks1>>marks2;
        }
        void display( ) // public function
        {
                cout<<“Name:"<<name<<endl;
                cout<<“Marks are:"<<endl<<marks1<<endl<<marks2;
        }
}; // end of class
```

```
int main( )
{
        student obj1;
        obj1.getdata( );
        obj1.display( );
}
```


Output:

```
Enter name, marks in two subjects:
Mahesh 20 25
Name:Mahesh
Marks are:
20
25
```

Class Example Program

- In the above program, class name is student, with private data members name, marks1 and marks2, the public data members getdata( ) and display( ).
- Functions, the getdata( ) accepts name and marks in two subjects from user and display( ) displays same on the output screen.

Scope Resolution Operator (::)

- It is used to define the member functions outside the class.
- Scope resolution operator links a class name with a member name in order to tell the compiler what class the member belongs to.
- Used for accessing global data.

Syntax

```
return_type   class_name : : function_name (actual arguments)

{

//function body

}
```

Scope Resolution Example

```
#include<iostream>
using namespace std;
class student
{
    private:
        char name[10]; // private variables
        int marks1,marks2;
    public:
        void getdata( );
        void display( );
};

void student::getdata( )
{
    cout<<"Enter name,marks in two subjects:"<<endl;
    cin>>name>>marks1>>marks2;
}

void student::display( )
{
        cout<<"Name:"<<name<<endl;
        cout<<"Marks
are"<<endl<<marks1<<endl<<marks2;
}

int main( )
{
        student obj1;
        obj1.getdata( );
        obj1.display( );
}
```


Output:

```
Enter name,marks in two subjects:
Mahesh 25 26
Name:Mahesh
Marks are
25
26
```

Accessing global variables using scope resolution operator (: :)

```
//Example:
#include<iostream>
using namespace std;
int a=100; // declaring global variable
class xyz
{
    public:
        int a;
        void f( )
        {
            a=20; // local variable
            cout<<a<<endl; // prints value of a as 20
        }
};
int main( )
{
        xyz obj;
        obj.f( ); // this function prints value of a(local variable) as 20
        cout<<::a<<endl; // this statement prints value of a(global variable) as 100
}
```

Output:

```
20
100
```

Defining the functions with arguments(parameters):

```
#include<iostream>
using namespace std;
class item
{
    private:
    int number, cost;
    public:
    void getdata(int a,int b );
    void display( );
};

void item::getdata(int a,int b)
{
    number=a;
    cost=b;
}

void item::display( )
{
        cout<<"Cost:"<<number<<endl;
        cout<<"Number:"<<cost<<endl;
}
int main( )
{
        item i1;
        i1.getdata(10,20);
        i1.display( );
}
```


Output:

Cost:10
Number:20

Data Abstraction

- The class construct provides facilities to implement data abstraction.
- Let us take up the example of the LCD projector. It has member data (light and fan) as well as member functions (switches that operate the light and the fan).
- This real-world object hides its internal operations from the outside world.
- It, thus, obviates the need for the user to know the possible pitfalls that might be encountered during its operation. During its operation, the LCD projector never reaches an invalid state. Moreover, the LCD projector does not start in an invalid state.
- Data abstraction is a virtue by which an object hides its internal operations from the rest of the program. It makes it unnecessary for the client programs to know how the data is internally arranged in the object.
- Thus, it obviates the need for the client programs to write precautionary code upon creating and while using objects.

Arrow Operator

- Member functions can be called with respect to an object through a pointer pointing at the object.
- The arrow operator (->) does this.

```
//Example:
#include<iostream>
using namespace std;
class xyz
{
        public:
        int a;
        void f( )
        {
                a=20;
        }
};
int main( )
{
        xyz obj;
        xyz * objptr;
        objptr = & obj;
        objptr->f( );
        cout<<objptr->a<<endl;
}
```

Output:
20
100

Member Function Overloading in C++

- Two or more functions have the same names but different argument lists.
- The arguments may differ in type or number, or both.
- However, the return types of overloaded methods can be the same or different is called function overloading.

Member Function Overloading in C++

```
#include<iostream>
using namespace std;
class A
{
        public:
        void show();
        void show(int); //function show() overloaded!!
};
void A::show()
{
        cout<<"Hello, Mahesh\n";
}
void A::show(int x)
{
        for(int i=0;i<x;i++)
        cout<<"Hello, HIT\n";
}
```

```
int main()
{
        A A1;
        A1.show(); //first definition called
        A1.show(3); //second definition called
}
```


Output:

```
Hello, Mahesh
Hello, HIT
Hello, HIT
Hello, HIT
```

Default Values for Formal Arguments of Member Functions

- We already know that default values can be assigned to arguments of non-member functions.
- Default values can be specified for formal arguments of member functions also.

```
#include<iostream>
using namespace std;
class A
{
        public:
        void show(int=1);
};
void A::show(int p)
{
        for(int i=0;i<p;i++)
        cout<<"Hello, Mahesh\n";
}
```

```
int main()
{
        A A1;
        A1.show(); //default value taken
        A1.show(3); //default value overridden
}
```

Output:

```
Hello, Mahesh
Hello, Mahesh
Hello, Mahesh
Hello, Mahesh
```

Inline Member Functions

- Member functions are made inline by either of the following two methods.
  - By defining the function within the class itself
  - By only prototyping and not defining the function within the class. The function is defined outside the class by using the scope resolution operator. The definition is prefixed by the inline keyword. As in non-member functions, the definition of the inline function must appear before it is called. Hence, the function should be defined in the same header file in which its class is defined

Inline Member Functions

```
#include<iostream>
using namespace std;
class A
{
        public:
        void show();
};
inline void A::show() //definition in header file itself
{
        cout<<"Hello, Mahesh\n";    //definition of A::show() function
}
```

```
int main()
{
        A A1;
        A1.show(); //default value taken
}
```

Output:

```
Hello, Mahesh
```

Constant Member Functions

- Let us consider this situation. The library programmer desires that one of the member functions of his/her class should not be able to change the value of member data.
- This function should be able to merely read the values contained in the data members, but not change them.
- However, he/she fears that while defining the function he/she might accidentally write the code to do so.
- In order to prevent this, he/she seeks the compiler's help.
- If he/she declares the function as a *constant function*, and thereafter attempts to change the value of a data member through the function, the compiler throws an error.

Constant Member Functions

```
#include <iostream>
using namespace std;
 class Test {
     int value;
    public:
        Test(int v)
          {
                    value = v;
          }
         int getValue() const { return value; }
};
```

```
int main()
{
    Test t(20);
    cout << t.getValue();
    return 0;
}
```


Output:
20

Mutable Data Members

- A mutable data member is *never* constant.
- It can be modified inside constant functions also.
- Prefixing the declaration of a data member with the keyword mutable makes it mutable.

Mutable Data Members

```
class A
{
    int x;          //non-mutable data member
    mutable int y;      //mutable data member
    public:
    void abc() const //a constant member function
    {
        x++;     //ERROR: cannot modify a non-constant data
                 //member in a constant member function
        y++;     //OK: can modify a mutable data member in a
                 //constant member function
    }
    void def() //a non-constant member function
    {
        x++;     //OK: can modify a non-constant data member
                 //in a non-constant member function
        y++;     //OK: can modify a mutable data member in a
                 //non-constant member function
    }
};

int main()
{
        A obj;
        obj.abc();
        obj.def();
        retrun 0;
}
```

Friend Functions

- A class can have global non-member functions and member functions of other classes as friends.
- The friend function is a function that is not a member function of the class but it can access the private and protected members of the class.
- The friend function is given by a keyword friend.
- These are special functions which are declared anywhere in the class but have given special permission to access the private members of the class.

Friend Functions

- Properties of Friend Function
  - It is not defined within the scope of the class
  - It cannot be invoked by the object of particular class
  - It can be invoked like a normal function

Friend Functions

A few points about the friend functions that we must keep in mind are as follows:

- **friend** keyword should appear in the prototype only and not in the definition.
- Since it is a non-member function of the class of which it is a friend, it can be prototyped in either the private or the public section of the class.
- A friend function takes one extra parameter as compared to a member function that performs the same task. This is because it cannot be called with respect to any object.
- Instead, the object itself appears as an explicit parameter in the function call.
- We need not and should not use the scope resolution operator while defining a friend function.

Friend member functions

```
#include <iostream>

using namespace std;

class Box
{
  double width;

  public:
    friend void printWidth( Box box );
    void setWidth( double wid )
    {
          width = wid;
    }
};

void printWidth( Box box )
{
          cout << "Width of box : " << box.width <<endl;
}

// Main function for the program
int main()
{
   Box box;   // set box width without member function

  box.setWidth(10.0);    // Use friend function to print the wdith.
  printWidth( box );

   return 0;
}
```

Output:
Width of box : 10

Friend Class

- A class can be a friend of another class.
- *Member functions of a friend class can access private data members of objects of the class of which it is a friend.*
- If class B is to be made a friend of class A, then the statement

friend class B;

- should be written within the definition of class A.

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 << “Value of a is " << x.a;
        }
};

int main()
{
        A p;
        B q;
        q.showA(p);
        return 0;
}
```

Static data members

- Static data members hold global data that is common to all objects of the class. Examples of such global data are
  - count of objects currently present,
  - common data accessed by all objects, etc.
- **A static data member has certain special characteristics:**
  - It is initialized to zero when first object is created. No other initialization is permitted.
  - Only one copy of the data member is created for the entire class and is shared by all the objects of class, no matter how many objects are created.
  - Static variables are normally used to maintain values common to entire class objects.

Static data members

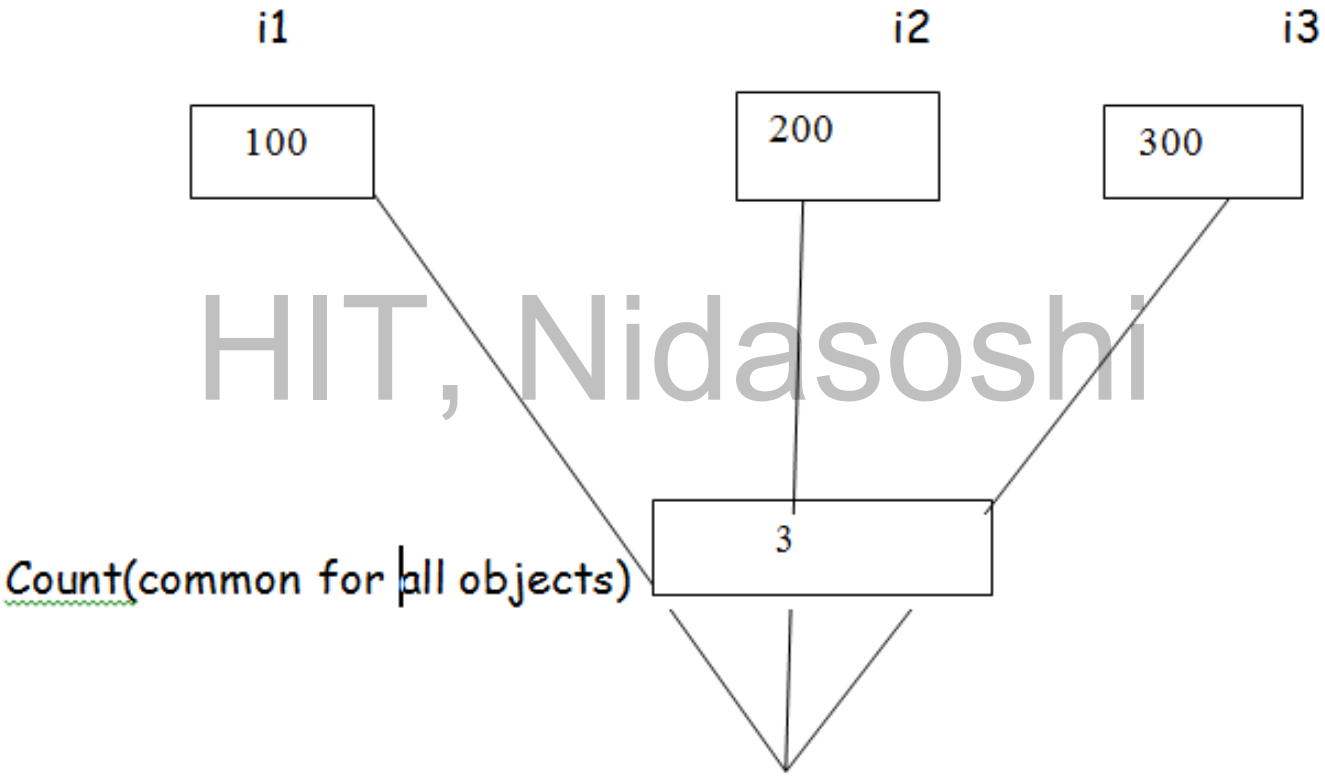
```
#include<iostream>
using namespace std;
class item
{
        static int count; // static data member
        int number;
        public:
        void getdata( )
        {
                number=0;
                count++;
        }
        void putdata( )
        {
                cout<<"count value"<<count<<endl;
        }
};
```

```
int main( )
{
        item i1,i2,i3; // count is initialized to zero
        i1.putdata( );
        i2.putdata( );
        i3.putdata( );
        i1.getdata( );
        i2.getdata( );
        i3.getdata( );
        i1.putdata( );    // display count after reading
data    i2.putdata( );
        i3.putdata( );
}
```

Output:

```
Count value 0
Count value 0
Count value 0
Count value 3
Count value 3
Count value 3
```

Static data members



Static Member Functions

- Like a static member variable, we can also have static member functions.
- ***A member function that is declared as static has the following properties:***
  - A static member function can have access to only other static members declared in the same class.
  - A static member function can be called using the class name, instead of objects.
  - A static function cannot refer to any non-static member data in its class.
- **Syntax:**

  class_name : : function_name ;

Static Member Functions

- In the above program, the static variable count is initialized to zero when objects are created.
- count is incremented whenever data is read into object.
- Since three times getdata( ) is called, so 3 times count value is created.
- All the 3 objects will have count value as 3 because count variable is shared by all the objects, so all the last 3 statements in
- main( ) prints values of count value as 3.

Static Member Functions

```
#include<iostream>
using namespace std;
class item
{
        int number;
        static int count;
        public:
        void getdata(int a )
        {
                number=a;
                count++;
        }
        static void putdata( )
        {
                cout<<"count value"<<count;
        }
};
int main( )
{
        item i1,i2;
        i1.getdata(10);
        i2.getdata(20);
        item::putdata( );
// call static member function using class name with scope resolution operator.
}
```

Static Member Functions

Output:

- Count value 2
- In the above program, we have one static data member count, it is initialized to zero, when first object is created, and one static member function putdata( ), it can access only static member.
- When getdata( ) is called twice, each time, count value is incremented, so the value of count is 2.when static member function putdata( ) is called, it prints value of count as 2.

Objects and Functions

- There are two cases that occur in relation to objects with functions.
  - Passing object as argument to function
  - Returning object from function

Passing object as argument to function

- The objects can be explicitly passed as an argument to the function.
- In this case, we have to write the constructor explicitly.

Returning object from function

- The object can be returned from the function.
- For that purpose the data type of that function must be of class type.