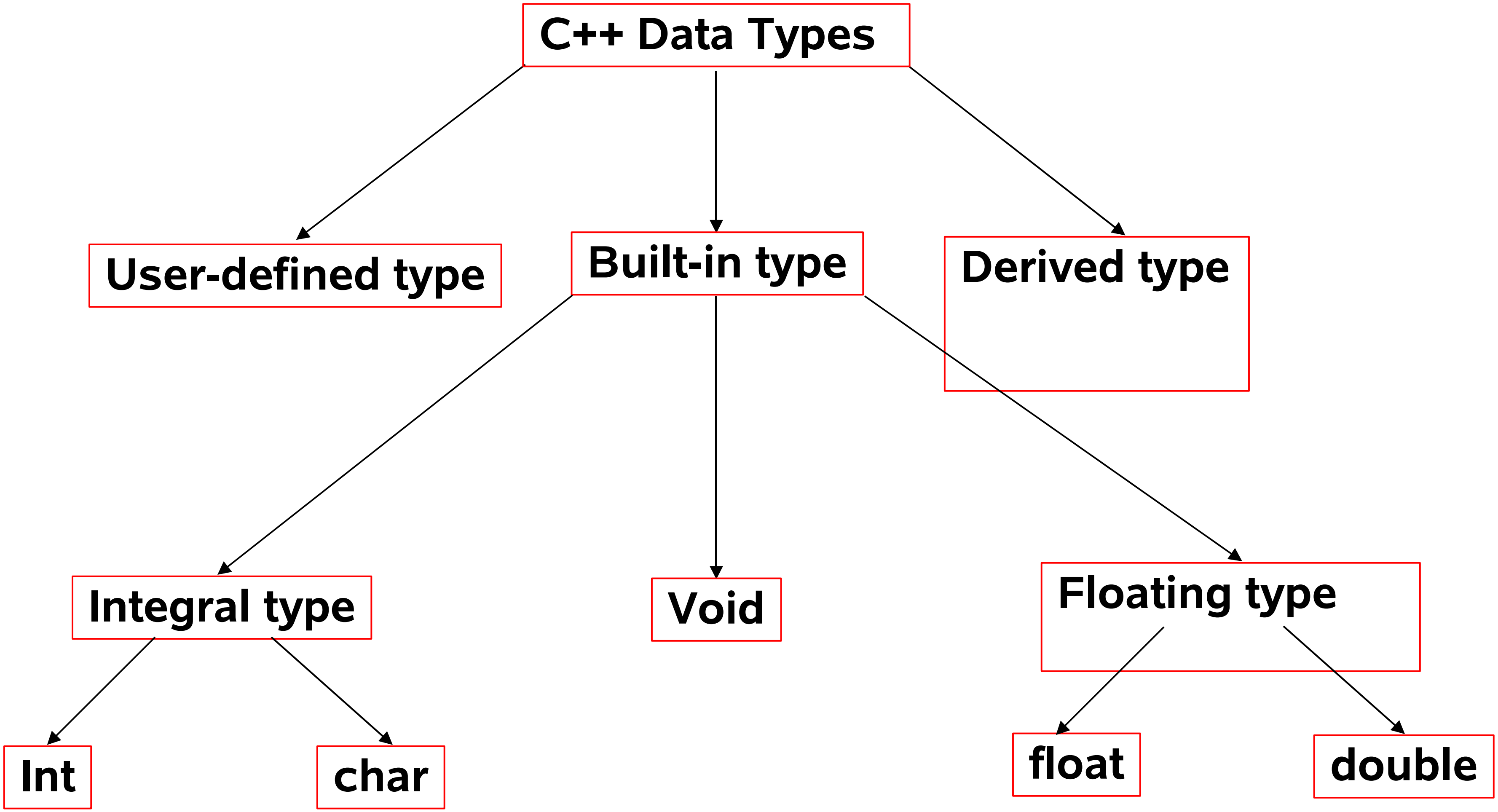
Object Oriented Programming in C++

Data Types


C++ Data Types
User-defined type
Built-in type
Derived type
Integral type
Void
Floating type
Int
char
float
double

Size and Range of C++ basic data types

Type	Bytes	Range
Char	1	-128 to 127
Unsigned char	1	0 to 255
Signed char	1	-128 to 127
Signed int	2	-32678 to 32767
Short int	2	- 32678 to 32767
Unsigned int	2	0 to 65535

Enumerated Data Type

A enumerate data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code.

enum shape{*circle*, *square*, *triangle*};

4 12 16

enum colour{*red*, *blue*, *green*, *yellow*};

shape *circle*;

colour *background*;

int c=*red*;

enum colour{*red*, *blue*=4, *green*=12, *yellow*=16};

Dynamic Initialization of Variables

In C, a variable must be initialized using a constant expression, and the C compiler would fix the initialization code at the time of compilation. C++, however, permits initializatiion of the variables at run time. This is referred as dynamic initialization.

```
int n = strlen (string);
float area= 3.14159 * rad * rad;

float average = sum /i;
```

Memory Management Operators

C uses malloc() and calloc() functions to allocate memory dynamically at run time.

It uses the function free() to free dynamically allocated memory, created by malloc() and calloc().

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

```
int *p = new int;
float *q = new float:
```

Memory Management Operators (Contd.)

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

int *p = new int (25);
```

new can be used to create a memory space for any data type including user-defined such as arrays, structures and classes.

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

Memory Management Operators (Contd.)

delete **pointer-variable;**

***delete* p;**

***delete* q;**

Memory Management Operators (Contd.)

The new opearator offers the following advantages over the function malloc();

- **It automatically computes the size of the data object. We need not use the operator sizeof.**
- **It automatically returns the correct pointer type, so that there is no need to use a type cast.**
- **It is possible to initialize the object while creating the memory space.**
- **Like any other pointer, new and delete can be overloaded.**

Memory Management Operators (Contd.)

The **new** opearator offers the following advantages over the function **malloc*();***

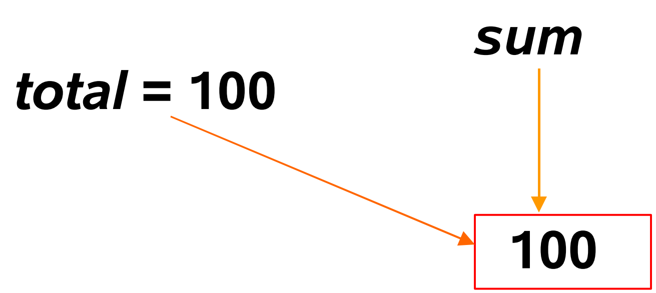
- It automatically computes the size of the data object. We need not use the operator **sizeof**.
- It automatically returns the correct pointer type, so that there is no need to use a type cast.
- It is possible to initialize the object while creating the memory space.
- It is possible to initialize the object while creating the memory space.

Reference variable

A reference variable provides an alias (alternative) for a previously defined variable,

data-type & reference-name = variable-name;

```
float total = 100;
float & sum  = total;
```



Call By Reference

Body of the Function:

```
int & max (int &x, int &y)
{
    if (x > y )
        return x;
    else
        return y;
}
```

The statement of function calling:

```
max (a, b) = -1;
```

Note: We can call a function **max(,)** of the left side of an equation.

Generic pointer

A generic pointer can be assigned a pointer value of any basic data type, but it may not be de-referenced.

For example,

```
void *gp;

int *ip;                // int pointer
gp = ip;
```

This is a valid statement but the statement,

*ip = *gp; is illegal

Classes and Objects

The most important feature of C++ is the "class". A class is an extension of the idea of a structure in C. It is a new way of creating and implementing a use-defined data-type.

The only difference between a structure and class in C++ is that, by default, the members of a class are *private*, while, by default, the member of a structure are *public*.

Classes and Objects (contd.)

```
Class Class_name
        {
                private:
                        variable declaration;
                        function declaration;

                public:
                        variable declaration;
                        function declaration;
        };
```

Classes and Objects (contd.)

```
Class Class_name
        {
                private:
                        variable declaration;
                        function declaration;

                public:

                        variable declaration;
                        function declaration;
        };
```

Classes and Objects (contd.)

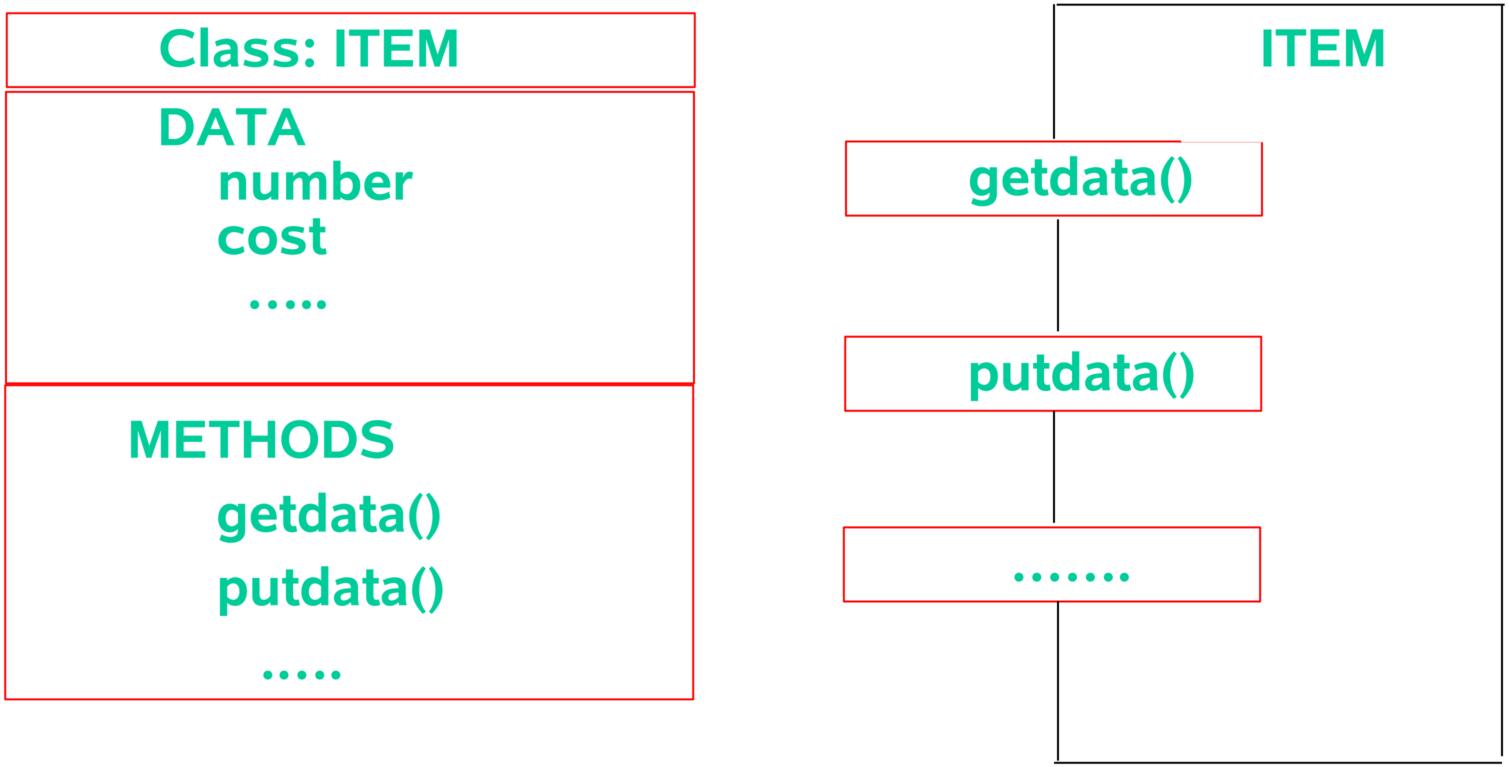
- The variables declared inside the class are know as *data member* and the functions are known as *member function*.

- The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.

- The keywords *private* and *public* are known as visibility labels.

- By default, the members are *private*.

Classes and Objects (contd.)



Classes and Objects (contd.)

Creating Objects:

item *x*; // memory for *x* is crated

item *y*, *z*; // more then one objects can be created

Access Class Members:

object-name.function-name(actual-arguments);

***x.getdata* (100,75.5);**

***x.putdata* ();**

Classes and Objects (contd.)

Outside the class definition:

```
return-type class-name :: function-name (argument declaration)
        {
            Function body
        }
```

Outside the class definition:

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

Outside the class definition:

```
void item :: putdata (void)
        {
cout<< "Number::"<<number<< endl;
cout << "cost"<<cost<<"\n";
        }
```

Classes and Objects (contd.)

Class definition:

```
class item
  {
int number;

float cost;
 public:
void getdata (int a, float b);
// inline function
 void putdata (void)
            {
cout << number <<“\n”;
cout <<cost <<endl;
            }
            };
```

Classes and Objects (contd.)

Class definition:

```
class item
{
 int number;

 float cost;

 public:
 void getdata (int a, float b)
// inline function
 void putdata (void)
{
cout << number <<"\n";
cout <<cost <<endl;
}
};
```

Classes and Objects (contd.)

```
void item :: getdata (int a, float b)
        {
         number = a; // private variable are directly used
         cost =  b;     // are directly used
         }
```

Inline Function

Class definition:

```
class item
  {
int Number;
float Cost;
public:
void GetData (int a, float b);
void PutData (void);
};
inline void item :: GetData (int a, int b)
 {
 Number = a;
 Cost = b;
 }
```

Classes and Objects (contd.)

```
main ()
 {
 item x;
 cout <<"\n Object x" << endl;
 x.GetData (100, 298.98);
 x.PutData ();
 item y;
 cout <<"\n Object y" << endl;
 y.GetData (300, 565.7678);
 y.PutData ();
  }
```

Private Member Functions

A private member function can only called by another function that is a member of class. <u>Even</u> an object cannot <u>invoke</u> a private function using dot operator.

```
class Sample
    {
        int m:
        void Read (void);
        public:
        void Update (void);
        void Write (void);
    }:
```

If *s1* is an object of <u>Sample</u>, then

```
s1.Read();      // won't work;
is illegal.     // objects
                //cannot
                //access
                // private
                //members.
```

Private Member Functions

```
void Sample :: Write (void)
{
    Read();
}
```

Arrays Within a Class

The arrays can be used within a class.

```
const int Size = 10;
        class array
        {
                int a[Size];
                public:
                void Set-Value (void);
                void Display (void);
        };
```

Arrays Within a Class (Contd.)

```
#include<iostream.h>
const int m = 50;
class ITEMS
{           int ItemCode [m];
            float ItemPrice [m];
            int Count;
    public:
            void Int (void) {Count = 0;}
            void GetItem (void);
            void DisplaySum (void);
            void Remove (void);
            void DisplayItem (void);
};
```

Arrays Within a Class (Contd.)

```
void ITEM :: GetItem (void)
      {
cout << "Enter Item Code:";
cin >> ItemCode [Count];
cout << "Enter Item Cost:  ";

cin >> ItemPrice [Count];
}
```

Arrays Within a Class (Contd.)

```
void ITEM :: DisplaySum (void)
{

float Sum = 0;
for (int i = 0; i < Count; i++)
Sum + = ItemPrice[i];
cout << “\n Total Value:” << Sum << “\n”;
}
```

Arrays Within a Class (Contd.)

```
void ITEM :: Remove (void)
{

        int a;
        cout << "\n Enter Item Code";
        cin >> a;
        for (int i = 0; i < Count; i++)
                if ( ItemCode[i] == a )
                    ItemPrice [i] = 0;
}
```

Arrays Within a Class (Contd.)

```
void ITEM :: DisplayItems (void)
{

        cout << “\n Code Price”;
        for (int i = 0; i < Count; i++)
        {
          cout << “   ” << ItemCode[i];
          cout << “    ” << ItemPrice[i];
        }
        cout << “\n”;
}
```

Arrays Within a Class (Contd.)

```
main ()
  {
          ITEMS Order;
          Order.Int ();
          int x;
do {
cout <<  "\n You can do following:"
          << "Enter Appropriate Number \n";
cout << "\n 1: Add Item";
cout << "\n 2: Display Total Value";
cout << "\n 3: Delete An Item";
cout << "\n 4: Display all Items";
cout << "\n 5: Quit";
cout << "\n \n What is your Option?";
cin >> x;
```

Arrays Within a Class (Contd.)

```
switch (x)
        {
case 1: Order.GetItem (); break;
case 2: Order.DisplaySum (); break;
case 3: Order.Remove (); break;
case 4: Order.DisplayItems (); break;
case 5: break;
default: cout << "\n Error in Input";
        }
} while (x != 5); \\ do-while ends here.

        return 0;

        } \\ main () ends here
```

Static Data Member

A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a C static variable.

- It initialized to zero when the first object of its class is created. No other initialization is permitted.

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

- It is visible only within the class, but its lifetime is the entire program.

Static Data Member (Contd.)

```
# include<iostream.h>
        class Item
        {
                static int Count;
                int Number;
         public:
                void GetData (int a)
                {
                Number = a;
                Count ++;
                }

                void GetCount (void)
                { cout << “Count: ”;
                  cout << Count  << “\n”;
                }

   };
```

Static Data Member (Contd.)

```
int Item :: Count;
int main()
        {
    Item a, b, c;
    a.GetCount ();          // count is initialized to zero
    b.GetCount ();          // display count
    c.GetCount ();

    a.GetData (100);        // getting data into object a
    b.GetData (200);        // getting data into object b
    c.GetData (100);        // getting data into object c
                 cout << “After reading data” << “\n”;
    a.GetCount ();
    b.GetCount ();
    c.GetCount ();
                          return 0;
}
```

Static Data Member (Contd.)

Output:

***Count*: 0**
***Count*: 0**
***Count*: 0**

After reading data

***Count*: 3**
***Count*: 3**
***Count*: 3**

```
int Item :: Count;
```

Note

The type and scope of each static variable must be defined outside the class definition.

Static Member Functions

- A static function can have access to only other static members (functions or variables) declared in the same class.

- A static member function can be called using the class name (instead of its objects) as follows:

class-name :: function-name;

Static Member Functions(Contd.)

```
#include <iostream.h>
class Test
{
int Code;
static int Count;
public:
void SetCode (void)
{     Code = ++ Count;          }

static void ShowCount (void)

{       cout << "Object Number:   " << Count << endl;       }

};
```

Static Member Functions(Contd.)

```
int Test :: Count;
int main()
{
        Test t1, t2;
        t1.SetCode();        t2.SetCode();

        test :: ShowCount ();

        test t3.SetCode();
        test :: ShowCount ();

                return 0;
}
```

Object As Function Arguments

- A copy of the entire object is passed to the function.
- Only the address of the object is transferred to the function.

Object As Function Arguments (contd.)

```
# include <iostream.h>
class Time
{        int Hours;
         int Minutes;
   public:
         void GetTime (int h, int m)
         { Hours = h;  Minutes = m;    }

         void PutTime (void)
         {     cout << Hours << " hours and   ";
               cout << Minutes << "minutes  ";   }

         void Sum (Time, Time);
};
```

Object As Function Arguments (contd.)

```
void Time :: Sum (Time t1, Time t2)
        {
        Minutes = t1.Minutes + t2.Minutes;
        Hours = Minutes / 60;
        Minutes = Minutes % 60;
        Hours = Hours + t1.Hours + t2.Hours;
        }
```

Object As Function Arguments (contd.)

```
int main ()
    {
        Time T1, T2, T3;
        T1. GetTime ( 2, 45 );
        T2. GetTime ( 4, 35 );

        T3. Sum ( T1, T2 );     // T3 = T1 + T2;

    cout << "T1 = ";        T1.PutTime ( );
    cout << "T2 = ";        T2.PutTime ( );
    cout << "T3 = ";        T3.PutTime ( );
                return 0;
    }
```

Object As Function Arguments (contd.)

The output of the program:

T1 **=** ***2*** **hours and** ***45*** **minutes**

T2 **= 4 hours and** ***35*** **minutes**

T3 **= 7 hours and** ***20*** **minutes**

Friend Function (Contd.)

```
class ABC
{
        …..
        …..
 public:
        …..
        …..
friend void xyz (void); // declaration
};
```

Friend Function

- The function declaration should be preceded by the key-word **friend**.
- The function canbe defined elsewhere in the program like a normal C++ function.
- The function definition does not use either the keyword **friend** or the **scope resolution operator ::**.

Friend Function (Contd.)

Definition:

The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any number of classes.

A friend function, although not a member function, has full access rights to the private members of the class.

Characteristics of Friend Function (Contd.)

- It is not in the scope of the class to which it has been declared as friend.

- Since it is not in the scope of the class, it cannot be called using the object of that class.

- It can be invoked like a normal function without the help of any object.

Characteristics of Friend Function (Contd.)

- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name (*A.x*).

- It can be declared either in the public or the private part of a class without affecting its meaning.

- Usually, it has the objects as arguments.

Friend Function (Contd.)

```
#include <iostream.h>
class Sample
{
            int a;
            int b;
public:
   void SetValue ()      { a = 25;   b = 40;   }

   friend float Mean (Sample s);
};

            float Mean (Sample s)
              {
  return float (s.a  + s.b) / 2.0;
              }
```

Friend Function (Contd.)

```
int main ()
{

Sample x;
x.SetValue ();
cout << “Mean Value = ” << Mean ( x );
return 0;
}
```

Output:

Mean Value = 32.5

Forward Declaration in Friend Function

```
        # include <iostream.h>
        class ABC;      // forward declaration
class XYZ
        {
                int x;
           public:
        void SetValue (int i) {           x = i;      }

        friend void Max (XYZ, ABC);
        };
```

Forward Declaration in Friend Function (contd.)

```
class ABC
{
        int a;
public:
    void SetValue (int i)  {        a = i;    }

    friend void Max (XYZ, ABC);
};
```

Forward Declaration in Friend Function (contd.)

```
void Max (XYZ m, ABC n)
{
        if (m.x > = n.a)
                cout << m.x;
        else
                cout << n.a
}
```

Forward Declaration in Friend Function (contd.)

```
int main ()
{
        ABC abc;
        abc.SetValue (10);
        XYZ xyz;
        xyz.SetValue (20);
        Max (xyz, abc);
        return 0;
}
```

Returning Object

```
#include <iostream.h>
class Complex
{
            float x;
            float y;
    public:
       void Input (float Real, float Imag)
        { x = Real;            y = Imag;         }

friend Complex Sum (Complex, Complex);

friend void Show (Complex):
};
```

Returning Object

```
Complex Sum (Complex c1, Complex c2)
{
        Complex c3;
        c3.x = c1.x + c2.x;
        c3.y = c1.y + c2.y;
        return (c3);
}

void Show (Complex c)

{    cout << c.x << "  +  j"  << c.y  << "\n";       }
```

Returning Object

```
int main
{
Complex A, B, C;
A.Input (3.1, 5.65);
B.Input (3.1, 5.65);

C = Sum (A, B);  // Our goal is to write C = A + B;
        cout << "A = "; Show (A);
        cout << "A = "; Show (B);
        cout << "A = "; Show (C);

return 0;
}
```

Returning Object

Output:

A = 3.1 + j 5.65

B = 2.75 + j 1.2

C= 5.85 + j 6.85

Pointer To Member (contd.)

We can define a pointer to the member *m* as follows:

int *A*::* ip = &*A* :: *m*;

The phrase &*A* :: *m* means the "address of the *m* member of *A* class"

int **ip* = *& m*; // won' work

Pointer To Member

```
Class A
{
private:
int m;
public:
        void Show ();
};
```

Function Prototyping

The prototype describes the function interface to the compiler by giving details such as the number of arguments, and the type of arguments and the type of return values.

```
type function-name (argument-list);
```

Call By Reference

The called function creates a new set of variables and copies the values of arguments into them.

The function does not have access to the actual variables in the *calling program* and can only work on the copies of values.

But, there may rise situations where we would like to change the values of the variables in the *calling program.* (For example, in Bubble sort ……)

Call By Reference (contd.)

```
Swap (int a, int b)
{
    int t = a;
    a = b;
    b = t;
}
```

```
Swap (int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```

Inline Function

One of the objectives of using functions in a program is to save memory space, which becomes appreciable when a function is likely to be called many times.

Inline Function

However, every time a function is called, it takes a lot of <u>extra time</u> in <u>executing a series of instructions</u> for tasks such as

- jumping to the function
- saving registers
- pushing arguments into the stack
- returning to the calling function

Conclusion: When the function is small, a substantial percentage of execution time may be spent in such overhead.

Inline Function (contd.)

Definition: An **inline** function is a function that is expanded in line when it is invoked.

That is, the compiler **replaces** the function call with the corresponding function code (something similar to **macros** expansion).

Inline Function (contd.)

```
inline function-header
        {
        Function-body
        }
```

```
inline double Cube (double a)
        {
        return a * a * a;
        }
```

Inline Function (contd.)

Some of the situations where inline expansion may not work are:

1. For function returning values, if a loop, a switch or a goto exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain static variables.
4. If inline functions are recursive.

Default Arguments

C++ allows us to call a function without specifying all its arguments.

In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call.

Example:

float Amount (float *Principal*, int *Period*, float *Rate* = 0.15);

Value = *Amount* (5000, 7);

Default Arguments (Contd.)

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 (Contd.)

```
# include <iostream.h>
int main ()
{
float Amout ;
float Value ( float p, int n, float r = 0.15 );
void PrintLine ( char Ch = '*', int Len = 40 );

PrintLine ();
Amount = Value ( 5000.00, 5 );

cout << "\n" <<      "Final Value =  "  << Amount  << "\n\n";
PrintLine();                 return 0;
}
```

Default Arguments (Contd.)

```
float Value ( float p, int n, float r )
{
int Year = 1;
float Sum = p;
while (Year <=  n)
    {
Sum = Sum * (1 + r);
Year = Year + 1;
    }
    return Sum;
} // function ends
```

Default Arguments (Contd.)

```
void PrintLine (char Ch= '*', int Len )
{
for ( int i = 1; i <= Len; i++ ) printf ("%c", Ch);
printf ("\n");
}
```

Function Overloading

Overloading refers to the use of the same thing for different purposes.

This means that C++ permits to use the same function name to create function that perform a variety of different tasks.

Function Overloading (contd.)

```
// Declarations

int Add ( int a, int b );   // Prototype 1

int Add ( int a, int b, int c );   // Prototype 2

double Add ( double  x, double  y );   // Prototype 3

double Add ( int p, double  q );   // Prototype 4

double Add ( double p, int  q );   // Prototype 5
```

Function Overloading (contd.)

```
// Function calls

cout << Add ( 25, 30 );                 // uses prototype 1
cout << Add ( 25, 10.0 );               // uses prototype 4
cout << Add ( 12.5, 17.5 );             // uses prototype 3
cout << Add ( 15, 100, 115 );           // uses prototype 2
cout << Add ( 15, 100, 115 );           // uses prototype 2
```

Function Overloading (contd.)

```
# inculude <iostream.h>
        int Volume ( int );
        double Volume ( double, int );
        long Volume ( long, int, int );

        int main ()
        {
cout << Volume ( 10 );     <<    "\n";
cout << Volume ( 4.5, 12 );     <<    "\n";
cout << Volume ( 65L, 89, 25 );     <<    "\n";
                return 0;

         }
```

Function Overloading (contd.)

```
int Volume (int s)  // Cube
                    {
              return s * s* s;
                    }
```

```
double Volume (double r, int h)  //Cylinder
          {
  return 3.14519 * r * r * h;
          }
```

Function Overloading (contd.)

```
long Volume (long l, int b, int h)
        {
  return (l * b * h);
        }
```

Constructors

Definition:

A constructor is a special type of <u>member function</u> whose task is to initialize the objects of its class.

- **It is special because its <u>name</u> is <u>same</u> as the class.**
- **The <u>constructor</u> is invoked whenever an object of its associated class is created.**
- **It is called <u>constructer</u> because it constructs the values of data members of the class.**

Constructors (Contd.)

```
// class with a constructor:
        class Integer
              {
        int m, n;
     public:
           Integer ( void ) ;     // constructor declared
           …………
           …………
              };

Integer :: Integer ( void ) {     m = 0;   n = 0;      }
```

Constructors (Contd.)

Special characterises:

- **They should be declared in the public section.**
- **They are invoked automatically when the objects are created.**
- **They do not have return types, even void and therefore, and they cannot return values.**
- **They cannot be inherited, though a derived class can call the base class constructor.**

Constructors (Contd.)

Special characterises:

- Like C++ functions, they can have default arguments.
- Constructors cannot be virtual.
- We cannot refer their address.
- An object with a constructor (destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators new and delete when memory allocation is required.

Constructors (Contd.)

```
Integer :: Integer ( void )  {      m = 0;   n = 0;       }

Integer :: Integer ( int x, int y )
           {       m = x;   n = y;              }

Integer Int;
Integer Int1 = Integer ( 12, 123 );      // explicit call
Integer Int2 ( 222, 345 );               // implicit call
```

Multiple Constructors in a Class

Integer *I1*;

Integer *I2* (20, 40);

Finally, the statement,

Integer *I3* (*I2*);

would invoke the <u>third constructor</u> which <u>initializes</u> the <u>data members</u> *m* and *n* of *I2* to 20 and 40 respectively.

Multiple Constructors in a Class

```
class Integer
{
int m, n;
public:
    Integer ( void )  {  m = 0;   n = 0;      }
    Integer ( int x, int y )
        {          m = x;   n = y;          }
    Integer ( Integer & i )
        {   m = i.m;
            n = i.n;
        }
};
```

Multiple Constructors in a Class

```
#include <iostream.h>
class Complex
{          float x;
           float y;
   public:
Complex ( )     {  cout << "Welcome to University of Kalyani";}
    Complex ( float a ) {    x = y = a; }
    Complex ( float Real, float Imag )
    {  x= Real;     y = Imag;     }

  friend Complex Sum ( Complex, Complex );
  void Show ( void );
  };
```

Multiple Constructors in a Class

```
Complex Sum ( Complex c1, Complex c2 )
{
        Complex c3;
        c3.x = c1.x + c2.x;
        c3.y = c1.y + c2.y;
        return (c3);
}

void Complex :: Show ( void )

{    cout << x << "  + j"  << y  << "\n"; }
```

Multiple Constructors in a Class

```
int main
{
        Complex A;
        Complex B ( 3.1);
        Complex C ( 4.1, 9.99 );

Complex D = Sum ( B, C );      //D = B + C;
        cout << “B = ”; B.Show ( );
        cout << “C = ”; C.Show ( );
        cout << “D = ”; D.Show ( );

return 0;
}
```

Multiple Constructors in a Class

Output:

Welcome to University of Kalyani

B **= 3.1 + j 3.1**

C **= *4*.1 + j 9.99**

D **= 7.2 + j 13.09**

Constructors with Default Arguments

```
Complex ( float Real, float Imag = 0 );

  // function declaration with defaults argument
```

```
Complex C( 5.0 );

        // function call with default argument
```

Dynamic Constructors

```
#include <iostream.h>
#include <string.h>
class String
  {       char *Name;
          int Length;
    public:
          String ( )
          {
               Length = 0;
               Name = new  char [ Length + 1];
          }
```

Dynamic Constructors

```
String (char * s)
        {
          Length = strlen ( s );
          Name = new  char [ Length + 1];
          strcpy ( Name, s )
              }

   void Display ( void )
        {
            cout << Name << "\n";
        }
```

Dynamic Constructors

```
Void Join ( String &a, String &b  );

};   // class definition ends
```

```
void String :: Join ( String &a, String &b )
              {
         Length = a.Length + b.Length;
         delete Name;
         Name = new char [ Length + 1 ];
         strcpy ( Name, a.Name );
         strcat ( Name, b.Name );
               }
```

Dynamic Constructors

```
Int main ()
{
char *first = "Anjan ";
String Name1 ( first ), Name2 ( "Kumar " ),
        Name3 ( "Sharma" ), s1, s2;
            s1. Join ( Name1, Name2 );
            s2. Join ( s1, Name3 );
    Name1.Display ( );
    Name2.Display ( );
    Name3.Display ( );
    s1.Display ( );
    s2.Display ( );
return 0;
}
```

Dynamic Constructors

Output:

Anjan

Kumar

Sharma

Anjan Kumar

Anjan Kumar Sharma

Constructing Two-Dimensional Arrays

```
#include <iostream.h>
class Matrix
{          int ** p;   // pointer to matrix
           int d1, d2;
   public:
Matrix ( int, int );
void GetElement (int i, int j, int Value )
     {          p [  i ] [  j  ] = Value;          }
int & PutElement ( int i, int j )
     {     return p [  i ] [  j  ] ;          }
 };
```

Constructing Two-Dimensional Arrays (Contd.)

```
Matrix :: Matrix ( int x, int y)
        {

    d1 = x;          d2= y;
p = new int * [ d1 ];  // creates an array pointer

 for ( int i = 0; i < d1; i++ )
     p [ i ] = new int [ d2 ];  // creates space
                                   for each row
        }
```

Constructing Two-Dimensional Arrays (Contd.)

```
int main ()
        {
          int m, n;
          cout << "Enter Size of Matrix:  ";
          cin >> m  >> n;
          Matrix ( m, n );  // matrix object is created
          int i, j, Value;
    for ( i = 0; i  < m; i++ )
         for ( j = 0; j < n; j++ ) {
cin  >> Value;        cout << A.GetElement ( i, j, Value);
                                }
```

Constructing Two-Dimensional Arrays (Contd.)

```
cout << "\n"
cout << A.PutElement ( 1, 2 );
                return 0;
} // main () end here.
```

Output: Enter the Size of Matrix: 3 4

11 13 17 19

67 68 87 98

69 43 23 18

87

Destructors

A **Destructor**, as the name implies, is used to destroy <u>the objects</u> that have been created by a **constructor**.

```
~ Integer ();
            Matrix :: ~ Matrix ( )  {
                  for ( int i = 0; i < d1; I ++) {
                        delete p [ i ];
                        delete p;                 }
```

Destructors (Contd.)

```
# include <iostream.h>
int Count = 0;
class Alpha{

Alpha()

        {
        Count ++;
        cout << "\n No. of Object Created " << Count;

        }
    ~ Alpha ()        {
        cout << "\n No. of Object Destroyed " << Count;
        Count --;        }
            };
```

Destructors (Contd.)

```
int main ()
    { cout << "\n \n Enter Main \n";
     Alpha A1, A2, A3, A4;
        {   cout << "\n \n Enter Block1 \n";
             Alpha A5;
         }
            {   cout << "\n \n Enter Block2 \n";
             Alpha A6;
            }
               cout << "\n \n RE-ENTER MAIN \n";
                return 0;
         } // main () ends here.
```

Destructors (Contd.)

Output:

```
ENTER MAIN
No. of Object Created 1
No. of Object Created 2
No. of Object Created 3
No. of Object Created 4

ENTER BLOCK 1
No. of Object Created 5
No. of Object Destroyed 5
```

Destructors (Contd.)

Output:

ENTER BLOCK 2

No. of Object Created 5

No. of Object Destroyed 5

RE-ENTER MAIN

No. of Object Destroyed 4

No. of Object Destroyed 3

No. of Object Destroyed 2

No. of Object Destroyed 1

Operator Overloading

Operator overloading is one of many exciting features of C++ language.

It is an important technique that has enhanced the power of extensibility of C++.

We have stated more than once that C++ tries to make the user-defined data types behave in much the same way as the built-in types.

Operator Overloading (Contd.)

This means that C++ has the ability to provide the operators with a special meaning for a data type.

The mechanism of giving such special meaning to an operator is known as operator overloading.

Operator Overloading (Contd.)

We can overload to all C++ operators except the following:

- Class member access operators (., .*)
- Scope resolution operator (::)
- Size operator (sizeof)
- Conditional operator (?)

Defining Operator Overloading

```
Return-type class-name :: operator op ( operand-list )
          {
          Function Body   // task defined
          }        // operator op is the function
```

Defining Operator Overloading

```
Vector operator + ( Vector );    // vector addition
Vector operator - ( );           // unary minus
friend Vector operator + ( Vector, Vector ); // vector addition
friend Vector operator - ( Vector );    // unary minus
Vector operator - ( Vector & );    // substraction
int operator == ( Vector );   // comparison
friend int operator == ( Vector, Vector ); // comparison
```

Overloading Unary Operators

```
# include <iostream.h>

class Space  {
            int x, y, z;
    public:
            void Getdata (int a, int b, int c);
            void Display ( void );
            void operator- ();
                        };
```

Overloading Unary Operators (contd.)

```
void space :: GetData ( int a, int b, int c )
    {   x = a;   y = b;   z = c;      }

void Space :: Display ( void )
  {   cout << x   << "   ";
      cout << y   << "   ";
      cout << z   << "   "
  }

void Space :: operator - ( ) {
this->x = -this->x; this-> y = - this-> y;
this-> z = - this-> z;     }
```

Overloading Unary Operators (contd.)

```
int main ( )
        {
        Space S;     S. GetData (10, 23, -98);
        cout << "S  : ";
        S.Display ();

- S;      // activates operator – ( ) function
        cout << "S  : ";
        S.Display ();
                return 0;
        }
```

Overloading Unary Operators (contd.)

By friend function:

```
friend void operator – (Space &s);        // declaration

    void operator – (Space &s)     // definition
    {
                    s.x = - ( s.x );
                    s.y = - s.y;
                    s.z = - s.z;
        }
```

Overloading Binary Operators

```
#include <iostream.h>
class Complex
{           float x;
            float y;
   public:
Complex ()      {x = y = 0;  }
Complex ( float a ) {     x = y = a; }
Complex ( float Real, float Imag )
{  x= Real;      y = Imag;      }
Complex operator + ( Complex );
void Display ( void );
};
```

Overloading Binary Operators

```
Complex Complex :: Operator +  ( Complex c )
        {
                Complex Tmp;
                Tmp.x = this->x + c.x;
                Tmp.y = y + c.y;
                return (Tmp);
        }

        void Complex :: Display ( void )

{    cout << x << “  +  j”  << y  << “\n”; }
```

Overloading Binary Operators

```
int main
{
        Complex A;
        Complex B ( 3.1, 5,65 );
        Complex C ( 4.1, 9,99 );

        A = B + C;   // B.operator+ ( C )
        cout << “B = ”; B.Display ( );
        cout << “C = ”; C.Display ( );
        cout << “A = ”; A.Display ( );

return 0;
}
```

Our Goal

```
int main
{
        Complex A; Complex P, Q, S;
        Complex B ( 3.1, 5,65 );
        Complex C ( 4.1, 9,99 );
        A = B+2; P = 3 + Q;
        S = 2 + 3;
        A = B + C;
        cout << “B = ”; B.Display ( );
        cout << “C = ”; C.Display ( );
        cout << “A = ”; A.Display ( );

return 0;
}
```

Overloading Binary Operators

Output:

B = 3.1 + j 5.65

C = 4.1 + j 9.99

A = 7.2 + j 15.64

Overloading Binary Operators

The statement *A* = *B* + *C*; is equivalent to

A = *B*.operator + (*C*);

Overloading Binary Operators

The statement *A* = *B* + *C*; is equivalent to

A = *B*.operator + (*C*);

Overloading Binary Operators

Why friend function ?

$A = B + 2;$

$A = 2 + B;$ (??)

Overloading Binary Operators

```
#include <iostream.h>
const Size = 3;
class Vector
 {                    int V [ size ];
            public:
            Vector ( );
            Vector ( int *x );
        friend Vector operator * ( int a, Vector b );
        friend Vector operator * ( Vector b, int a );
friend istream & operator >> ( istream &, Vector & );
friend ostream & operator << ( ostream &, Vector & );


   }
```

Overloading Binary Operators

```
Vector :: Vector ( ) {
        for ( int i = 0; i < Size; i ++ )    V [ i ] = 0;
  }

Vector :: Vector ( int *x )   {
        for ( int i = 0; i < Size; i ++ )    V [ i ] = x [ i ];
    }
 Vector operator * ( int a, Vector b ) {
              Vector c;
   for ( int i = 0; i < Size; i ++ )
        c.V [ i ] = a * b.V [ i ];
              return 0;          }
```

Overloading Binary Operators

```
Vector operator * ( int a, Vector b ) {
        Vector c;
for ( int i = 0; i < Size; i ++ )
        c.V [ i ] = a * b.V [ i ];
                return c;          }

Vector operator * (Vector b, int a ) {
                Vector c;
        for ( int i = 0; i < Size; i ++ )
                c.V [ i ] = b.V [ i ] * a;
                return c;          }
```

Overloading Binary Operators

```
Vector Vector :: operator * (int a ) {
        Vector c;
    for ( int i = 0; i < Size; i ++ )
        c.V [ i ] = ( this->V [ i ]  ) * a;
        return c;            }
```

Overloading Binary Operators

```
istream & operator >> ( istream & Din, Vector & b )
        {
        for ( int i = 0; i < Size; i ++ )
                Din  >> b.V [ i ];
                        return ( Din );
        }
ostream & operator << ( ostream & Dout, Vector & b )
                {
        Dout << " ( " << b.V [ i ];
            for ( int i = 0; i < Size; i ++ )
                    Dout <<" , "  << b.V [ i ];
                    Dout  << " ) ";
                                return Dout;
                    }
```

Overloading By Friend Functions

```
int main ()
{          int x [ 3 ] = {2, 3,4 };
Vector m;
Vector n = x;     // Vector n ( x )
cout << "Enter Elements of Vector m   " << "\n";
cin >> m;    //  invoke operator >> ()
cout << "\n";
cout << " m = "  << m << "\n";    //  invoke operator << ()
Vector p, q;
p =2 * m;
q = n * 2;
cout << "\n"   ;
cout << "p =  "  << p  << "\n";  //  invoke operator << ()

return 0;               } // main () end here
```

Overloading By Friend Functions

Output:

Enter elements of vector m

5 10 15

m = (5, 10, 15)

p = (10, 20, 30)

q = (4, 8, 12)

Manipulating of Strings Using Operators

```
# include <string.h>
#include <iostream.h>
class String
  {   char *p;  int Len;
  public:
String ( ) {      Len = 0;  p = new char [ Len + 1 ];      }
          String ( const char * s );
          String ( const String & s );
          ~ String {        delete p ;        }
                // + operator
friend String operator + ( const String & s, const String & t );
friend   int operator <=  ( const String & s, const String & t );
```

Manipulating of Strings Using Operators

```
// <= oprator

friend   int operator <=  ( const String & s, const String & t );

friend void Show ( const String s );

        }; // class definition of string ends here.
```

Manipulating of Strings Using Operators

```
String :: String ( const char *s )   {
        Len = strlen ( s );
        p = new char [ Len + 1 ];
        strcpy ( p, s );                }

String :: String ( const String & s )   {
        Len = s.Len;
        p = new char [ Len + 1 ];
        strcpy ( p, s.p );                }
```

Manipulating of Strings Using Operators

```
// overloading + operator
String operator + ( const String & s, const String & t )
{       String Tmp;
        Tmp.Len = s.Len  + t.Len;
        Tmp.p = new char [ Tmp.Len + 1 ];
        strcpy ( Tmp.p, s.p );
        strcat (Tmp.p, t.p );
        return ( Tmp );
}
```

Manipulating of Strings Using Operators

```
// overloading <= operator
int operator <= ( cost String & s, const String & t )
{       int m = strlen ( s.p );
        int n = strlen ( t.p );

        if ( m <= n )    return 1;
        else return 0;
}
```

Manipulating of Strings Using Operators

```
void Show ( const String s )
        {    cout << s.p;    }
```

Manipulating of Strings Using Operators

```
int main ( )
{
String S1 = "New "; String S2 = "York"; String S3 =
"Delhi";
String T1, T2, T3; T1 ( S1 ); T2 ( S2 ); T3 ( S3 );
cout << "\nT1 = "; Show ( T1 );
cout << "\nT2 = "; Show ( T2 ); cout << "\n";
cout << "\nT3 = "; Show ( T3 ); cout << "\n\n";
String T4 = T1 + T2;
```

Manipulating of Strings Using Operators

```
if ( t1 < = t3 )        {
      Show ( t1 );
      cout << "Smaller than  ";
      Show ( t3 );   cout << "\n";    }


  else              {
       Show ( t3 );
      cout << "Smaller than  ";
      Show ( t1 );
      cout << "\n";     }

            return 0;   } // main () ends here.
```

Manipulating of Strings Using Operators

Output:

t1 = New

t2 = York

t3 = New Delhi

New smaller than New Delhi

Inheritance

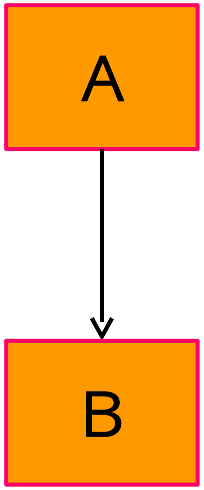
Reusability is yet another important feature of OPP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again.

Fortunately, C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways.

Inheritance (Contd.)

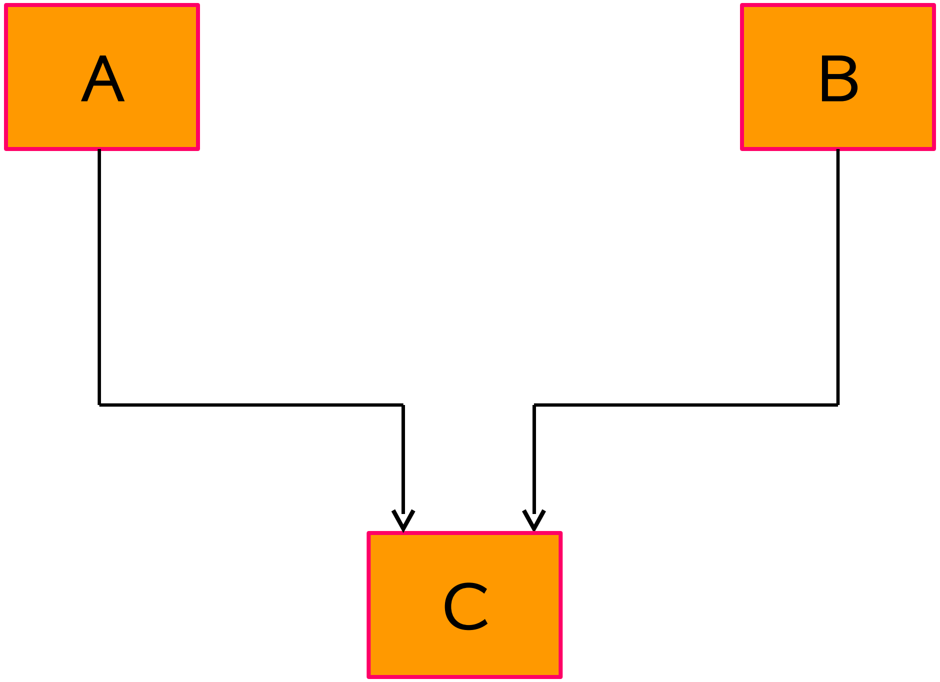
This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called Inheritance (or derivation)

Inheritance (Contd.)



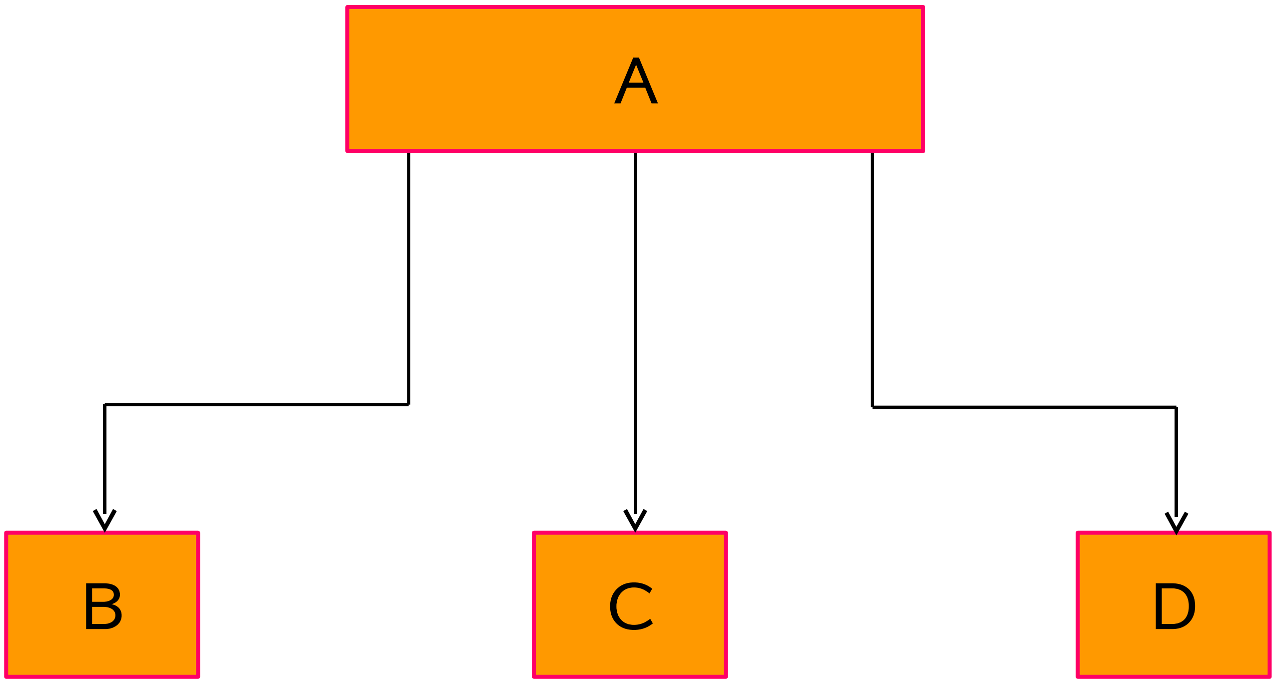
(a) Single Inheritance

Inheritance (Contd.)



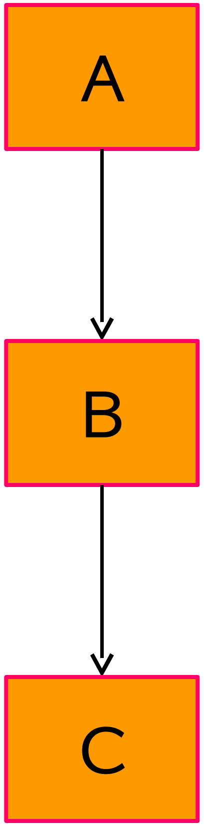
(c) Multiple Inheritance

Inheritance (Contd.)



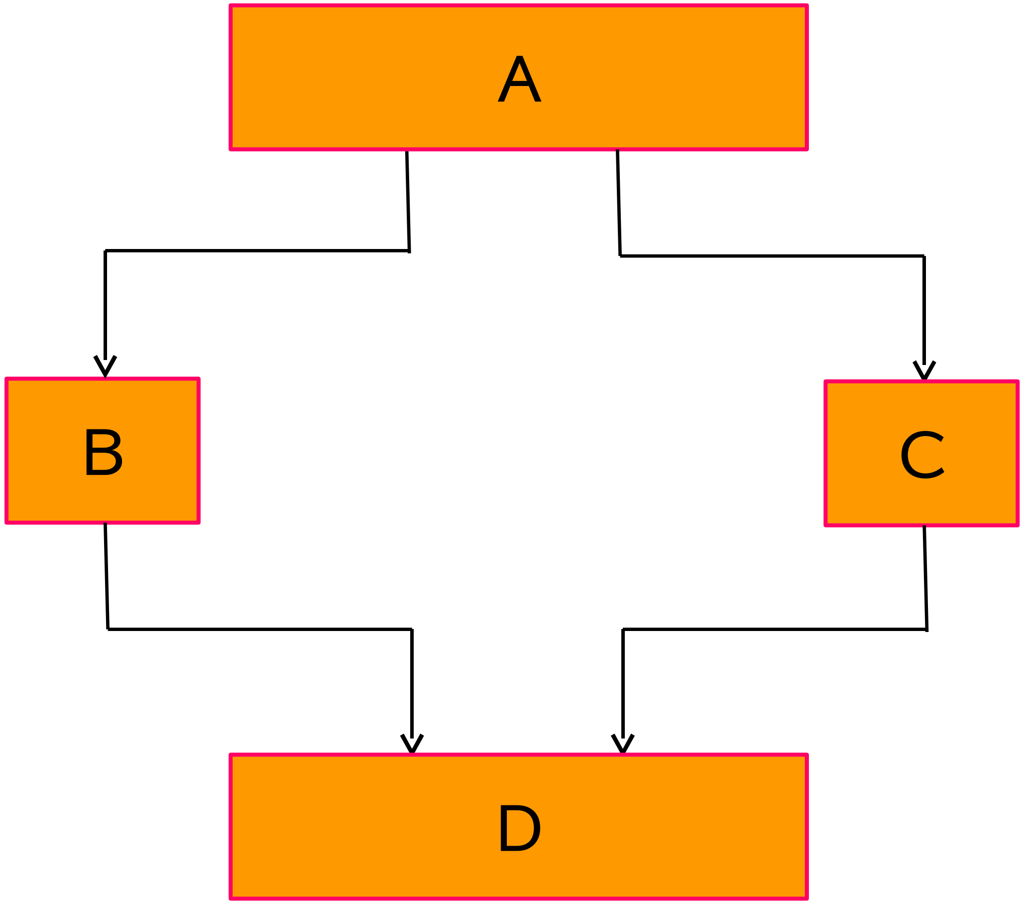
(B) Hierarchical Inheritance

Inheritance (Contd.)



(a) Multilevel Inheritance

Inheritance (Contd.)



(B) Hybrid Inheritance

Inheritance (Contd.)

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

        class ABC: private XYZ  // private derivation
        {        members of ABC        };

        class ABC: public XYZ   // public derivation
        {        members of ABC        };

class ABC: XYZ   // private derivation by default
        {        members of ABC        };
```

Single Inheritance

```
#include<iostream.h>
class B{
int a;        // private;  not inheritable
public:       // public; ready for inheritance
int b;
    void Getab ( );
    int Geta ( void );
    void Showa ( void );
          };
```

Single Inheritance (Contd.)

```
class D : public B   // public derivation
        {
                int c;
                public:
            void Mul ( void );
            void Display ( void );
          };
```

Single Inheritance (Contd.)

Single Inheritance (Contd.)

```
void B :: Getab ( void )  { a = 5;    b = 10;  }

int B :: Geta ( )  {  retrun a;    }

void B :: Showa ( )    {   cout << “a =   ” << a << “\n”       }

void D :: Mul ( void )        {        c = b * Geta ( );
}
```

Single Inheritance (Contd.)

```
void D :: Display ( )
{  cout << "a =  " << Geta ( )  << "\n";
   cout << "b =  " << b << "\n";
   cout << "c =  " << c << "\n\n";
}
```

Single Inheritance (Contd.)

```
int main ( )                                   {
        D d;
        d.Getab ( );
        d.Mul ( );
        d.Showa ( );
        d.Display ( );

        d.b = 20;
        d.Mul ( );
        d.Dispaly ( );
                return 0;                      }
```

Single Inheritance (Contd.)

Output

a **= 5**

a **= 5**

b **= 10**

c **= 50**

a **= 5**

b **= 20**

c **= 100**

Single Inheritance

```
#include<iostream.h>
class B {
    int a;      // private;  not inheritable
public:     // public; ready for inheritance
    int b;
    void Getab ( );
    int Geta ( void );
    void Showa ( void );
};
```

Single Inheritance (Contd.)

```
class D : private B    // private derivation
        {
              int c;
              public:
          void Mul ( void );
          void Display ( void );
         };
```

Single Inheritance (Contd.)

```
void B :: Getab ( void )  {
 cout << "Enter values for a and b:";
 cin >> a >> b;                    }

int B :: Geta ( )  {  retrun a;    }

void B :: Showa ( )    {   cout << "a =   " << a << "\n"       }

void D :: Mul ( )                  {
          Getab ( );
          c = b * Geta ( );     }
```

Single Inheritance (Contd.)

```
void D :: Display ( )
 {  cout << "a =  " << Geta ( )  << "\n";
    cout << "b =  " << b << "\n";
    cout << "c =  " << c << "\n\n";
 }
```

Single Inheritance (Contd.)

```
int main ( )                                    {
D d;
        // d.Getab ( );  won't work
d.Mul ( );
        // d.Showa ( );   won't work
          d.Display ( );

      // d.b = 20;     won't work
          d.Mul ( );
          d.Display ( );
                    return 0;                     }
```

Single Inheritance (Contd.)

Output

Enter value for a and b : 5 10

a = 5

b = 10

c = 50

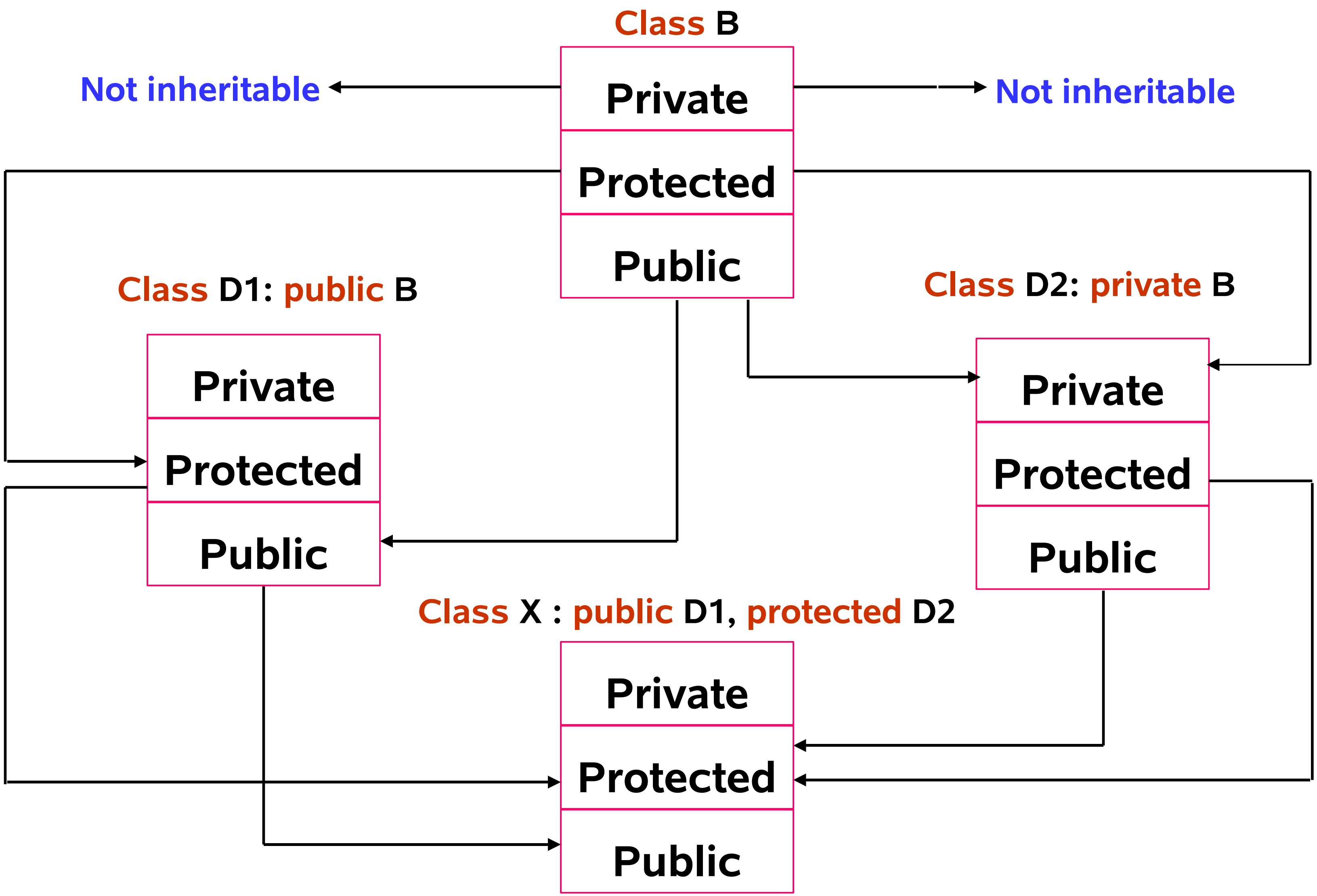
Enter value for a and b : 12 20

a = 12

b = 20

c = 240

Effect of Inheritance on the Visibility



Multilevel Inheritance

```
#include <iostream.h>
class Student
{
protected :
    int RollNumber;
public :
    void GetNumber ( int );
    void PutNumber ( void );
};
```

Multilevel Inheritance

```
void Student :: GetNumber ( int a ) {     RollNumber = a;     }
```

```
void Student :: PutNumber ( void )
    {     cout << “Roll Number:  ” << RollNumber << “\m”;    }
```

Multilevel Inheritance

```
Class Test : public Student
{
    protected :
        float Sub1;
        float Sub2;
    public :
        void GetMarks ( float, float );
        void PutMarks ( void );

};
```

Multilevel Inheritance

```
void Test :: GetMarks ( float x, float y )
        {     Sub1 = x;     Sub2 =  y;     }


void Test :: PutMarks ( void )
  {     cout << "Marks in Sub 1 = " << Sub1 << "\n";
        cout << "Marks in Sub 2 = " << Sub2 << "\n";
  }
```

Multilevel Inheritance

```
Class Result : public Test
{
                float Total;
public :
                void Display ( void );
};    // class definition of Result ends here

        void Result :: Display ( void )            {
                        Total = Sub1 + Sub2;
                        PutNumbers ( );
                        PutMarks( );
                cout << "Total =  " << Total << endl;  }
```

Multilevel Inheritance

```
int main ( )
{
Result Student1;        // Student1 created
Student1.GetNumber ( 222 );
Student1.GetMarks ( 85.0, 59.5 );
Student1.PutMarks();
Student1.Display ( );
return 0;
}
```

Multilevel Inheritance

Output

Roll number : 222

Marks in Sub1 = 85.0

Marks in Sub2 = 59.5

Total = 144.5

Multiple Inheritance

```
# include <iostream.h>

class M {
protected :
int m;
public :
void Get-m ( int );      }; // class definition of M ends here

class N {
protected :
int n;
public :
void Get-n ( int );      }; // class definition of N ends here
```

Multiple Inheritance

```
class P : public M, public N
        {
public :
        void Display ( void );
        };

void M :: Get-m ( int x )   {     m  = x;       }

void N :: Get-n ( int y )   {     n  = y;       }
```

Multiple Inheritance

```
void P :: Display ( void )                {
cout << “m =  ” << m << “\n”;
cout << “n =  ” << n << “\n”;
cout << “m * n  =  ” << m * n << “\n”;          }

    int main ()     {
            P p;
      p.Get-m ( 10 );          p.Get-n ( 20 );
            p.Display ( );
                          return 0;
                }
```

Multiple Inheritance

Output

m = 10

n = 20

m * n = 200

Ambiguity Resolution in Inheritance

```
class M {
public:
void Display ( ) {       cout << "Class M\n";      }
};

class N {
public:
void Display ( ) {       cout << "Class N\n";      }
};
```

Ambiguity Resolution in Inheritance

```
class P : public M, public N
{
public:
    void Display ( void )
    {
        M :: Display ( );      // overrides Display ( ) of M and N
// This statement is used to remove ambiguity
// if M :: is not placed before Display ( ) within the definition
of Display ( ) for class P then an ambiguity will be created.
    }
};
```

Ambiguity Resolution in Inheritance

```
int main ( )
{
        P p;
        p.Display ( );
}
```

Ambiguity Resolution in Inheritance

```
class A {
public:
void Display ( ) {        cout << “Class A\n”;        }
};

class B : public A
{
public:
void Display ( ) {        cout << “Class B\n”;        }
};
```

Ambiguity Resolution in Inheritance

```
int main ( )
        {
B b;                        // derived class object
b.Display ( );              // invokes Display ( ) in B
b.A :: Display ( );         // invokes Display ( ) in A
b.B :: Display ( );         // invokes Display ( ) in B
        return 0;
        }
```

Multiple Inheritance

Output

B

A

B

Hybrid Inheritance

```
#include <iostream.h>
class Student    {
        protected :
                int RollNumber;
        public :
    void GetMumber ( int )  {    RollNumber = a;    }

    void PutMumber ( void )
    {    cout << “Roll Number:  ” << RollNumber << “\m”;   }

        };  // class definition of Student ends here
```

Multilevel Inheritance

```
Class Test : public Student    {
                 protected :
                 float Sub1;    float Sub2;
                 public :
void GetMarks ( float x, float y )
                 {     Sub1 = x;     Sub2 =  y;     }


void PutMarks ( void )   {
                cout << "Marks in Sub 1 = " << Sub1 << "\n";
                cout << "Marks in Sub 2 = " << Sub2 << "\n";
                                                    }
                                             };
```

Multilevel Inheritance

```
Class Sports   {
protected :
float Score;
public :

void GetScore ( float s )
{     Score  = s;          }


void PutScore ( void )   {
      cout << “Sports wt :    ” << Score << “\n”;
              }
 };       // Definition of Sports class ends here
```

Multilevel Inheritance

```
Class Result : public Test, public Sports
{                     float Total;
public :
                      void Display ( void );
};    // class definition of Result ends here

              void Result :: Display ( void )  {
                  Total = Sub1 + Sub2 + Score;
                  PutNumbers ( );
                          PutMarks( );
                         PutScore ( );
         cout << "Total Score =  " << Total << endl;      }
```

Multilevel Inheritance

```
int main ( )
{
Result Student1;        // Student1 created
Student1.GetNumber ( 222 );
Student.GetMarks ( 27.0, 33.5 );
Student.GetScore ( 6.0 );

Student1.Display ( );
return 0;
}
```

Multilevel Inheritance

Output

Roll number : 222

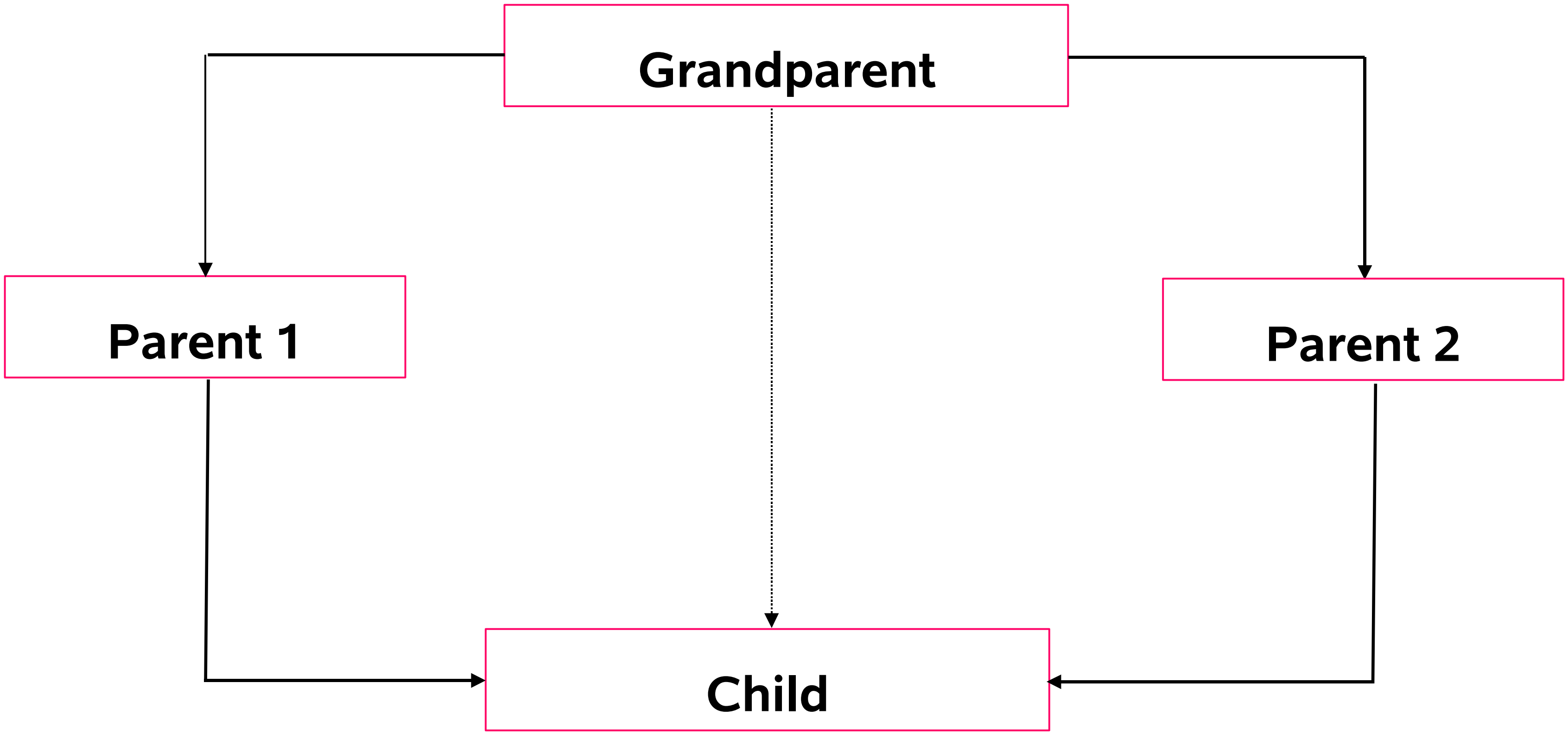
Marks in Sub1 = 27.0

Marks in Sub1 = 33.5

Sports wt. = 6.0

Total = 66.5

Virtual Base Class



Virtual Base Class

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as <u>virtual</u> base class while declaring the direct or intermediate base classes.

Inheritance with Virtual Class

```
#include <iostream.h>
class Student   {
        protected :
                int RollNumber;
        public :
    void GetNumber ( int a )  {    RollNumber = a;    }

    void PutNumber ( void )
    {     cout << "Roll Number:  " << RollNumber << "\m";    }

        };  // class definition of Student ends here
```

Inheritance with Virtual Class

```
Class Test : virtual public Student    {
        protected :
        float Sub1;    float Sub2;
        public :
void GetMarks ( float x, float y )
        {     Sub1 = x;     Sub2 =  y;     }


void PutMarks ( void )   {
        cout << “Marks in Sub 1 = ” << Sub1 << “\n”;
        cout << “Marks in Sub 2 = ” << Sub2 << “\n”;
                                    }
                        };
```

Inheritance with Virtual Class

```
Class Sports :   public virtual Student {
protected :
float Score;
public :

void GetScore ( float s )
{     Score  = s;           }


void PutScore ( void )   {
      cout << “Sports wt :    ” << Score << “\n”;
             }
 };       // Definition of Sports class ends here
```

Inheritance with Virtual Class

```
Class Result : public Test, public Sports
{                    float Total;
public :
                     void Display ( void );
};    // class definition of Result ends here

        void Result :: Display ( void )  {
              Total = Sub1 + Sub2 + Score;
              PutNumbers ( );
                         PutMarks( );
                         PutScore ( );
        cout << "Total Score =  " << Total << endl;      }
```

Inheritance with Virtual Class

```
int main ( )
{
Result Student1;        // Student1 created
Student1.GetNumber ( 678 );
Student.GetMarks ( 30.5, 25.5 );
Student.GetScore ( 7.0 );

Student1.Display ( );
return 0;
}
```

Inheritance with Virtual Class

Output

Roll number : 678

Marks in Sub1 = 30.5

Marks in Sub1 = 25.5

Sports wt. = 7.0

Total = 63.0

Abstract Classes

An abstract class is one that is used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes).

It is design concept in program development and provides a base upon which other classes may be built.

Constructors in Derived Classes

- We do not use constructor earlier in the derived classes for the shake of simplicity.
- One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function.
- However, if any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.

Constructors in Derived Classes

Derived-Constructor (Arglist1, Arglist2, ... ArglistN, Arglist (D):

Base1 (Arglist1),

Base2 (Arglist2),

.....

.....

BaseN (ArglistN),

{

Body of derived constructor

}

Constructors in Derived Classes

```
D ( int a1, int a2, float b1, float b2, int d1):
A ( a1, a2 ),
B ( b1, b2 ),
 {
Body of derived constructor
d = d1;        // executes its own body
}
```

Constructors in Derived Classes

D objD(5, 22, 33.6, 23.78, 43);

5 $\longrightarrow$ a1

22 $\longrightarrow$ a2

33.6 $\longrightarrow$ b1

23.78 $\longrightarrow$ b2

30 $\longrightarrow$ d1

Constructors in Derived Classes

Method of inheritance	Order of Execution
class B : public A { };	A ( ); base B ( );
class A : public B, public C { };	B ( ); base C( ); base A ( ); derived
class A : public B, virtual public C { };	C ( ); virtual base B ( ); base A ( ); derived

Constructors in Derived Classes

```
# include <iostream.h>
class Alpha   {
   int x;
   public:
    Alpha ( int i )
{       x = i;       cout << "Alpha Initialized \n";    }

void Show-x ( void )
{               cout << "x =   " <<  x  << "\n";            }


 } ;            // Definition of class Alpha ends here
```

Constructors in Derived Classes

```
class Beta {
    float y;
    public:
    Beta ( float j )
    {       y = j;       cout << "Beta Initialized \n";     }
    void Show-y ( void )
    {              cout << "y =   " << y << "\n";           }

};              // Definition of class Beta ends here
```

Constructors in Derived Classes

```
class Gamma : public Beta, public Alpha      {
                    int m, n;
                    public:
                     Gamma ( int a, float b, int c, int d ):
                            Alpha ( a ), Beta ( b)
                   {        m = c; n = d;
                     cout << “Gamma initialized \n”;    }
 void Show-mm ( void )
     {  cout << “m =   ” <<  m  << “\n”;
         cout << “n =   ” <<  n  << “\n”;            }
  } ;             // Definition of class Gamma ends here
```

Constructors in Derived Classes

```
int main ( )
        {
Gamma g ( 5, 10.75, 20, 30 );
cout << “\n”;
g.Show-x ( );
g.Show-y ( );
g.Show-mn ( );

        return 0;
          }  //  main ( ) ends here.
```

Constructors in Derived Classes

Output

Beta initialized

Alpha initialized

Gamma initialized

x = 5

y = 10.75

m = 20

n = 30

Note:

Beta is initialized first, although it appears second in the derived constructor.

Initialization in Constructors

```
# include <iostream.h>
class Alpha   {
   int x;
   public:
    Alpha ( int i )
{       x = i;       cout << “Alpha Initialized \n”;    }

void Show-x ( void )
{               cout << “x =   ” <<  x  << “\n”;            }


  } ;            // Definition of class Alpha ends here
```

Constructors in Derived Classes

```
class Beta {
   float y;
   public:
float p, q;
Beta ( float a, float b ) : p ( a ), q ( b  + p )
{       cout << "Beta Initialized \n";     }

void ShowBeta ( void )
{               cout << "p=   " << p  << "\n";
                cout << "p=   " << p  << "\n";          }
} ;             // Definition of class Beta ends here
```

Constructors in Derived Classes

```
class Gamma : public Beta, public Alpha      {
                    int u, v;
                    public:
                     Gamma ( int a, float b, int c ):
                            Alpha ( a * 2 ), Beta ( c, c ) , u ( a )
    {  v = c;         cout << “Gamma initialized \n”;    }
 void Show-mn ( void )
    {  cout << “u =   ” <<  u  << “\n”;
        cout << “v =   ” <<  v  << “\n”;             }
  } ;              // Definition of class Gamma ends here
```

Constructors in Derived Classes

```
int main ( )
        {
Gamma g ( 5, 10.75, 30 );
cout << "\n";
g.Show-x ( );
g.Show-y ( );
g.Show-mn ( );

        return 0;
          }  //  main ( ) ends here.
```

Constructors in Derived Classes

Output

Beta initialized

Alpha initialized

Gamma initialized

x = 10

p = 30

q = 60

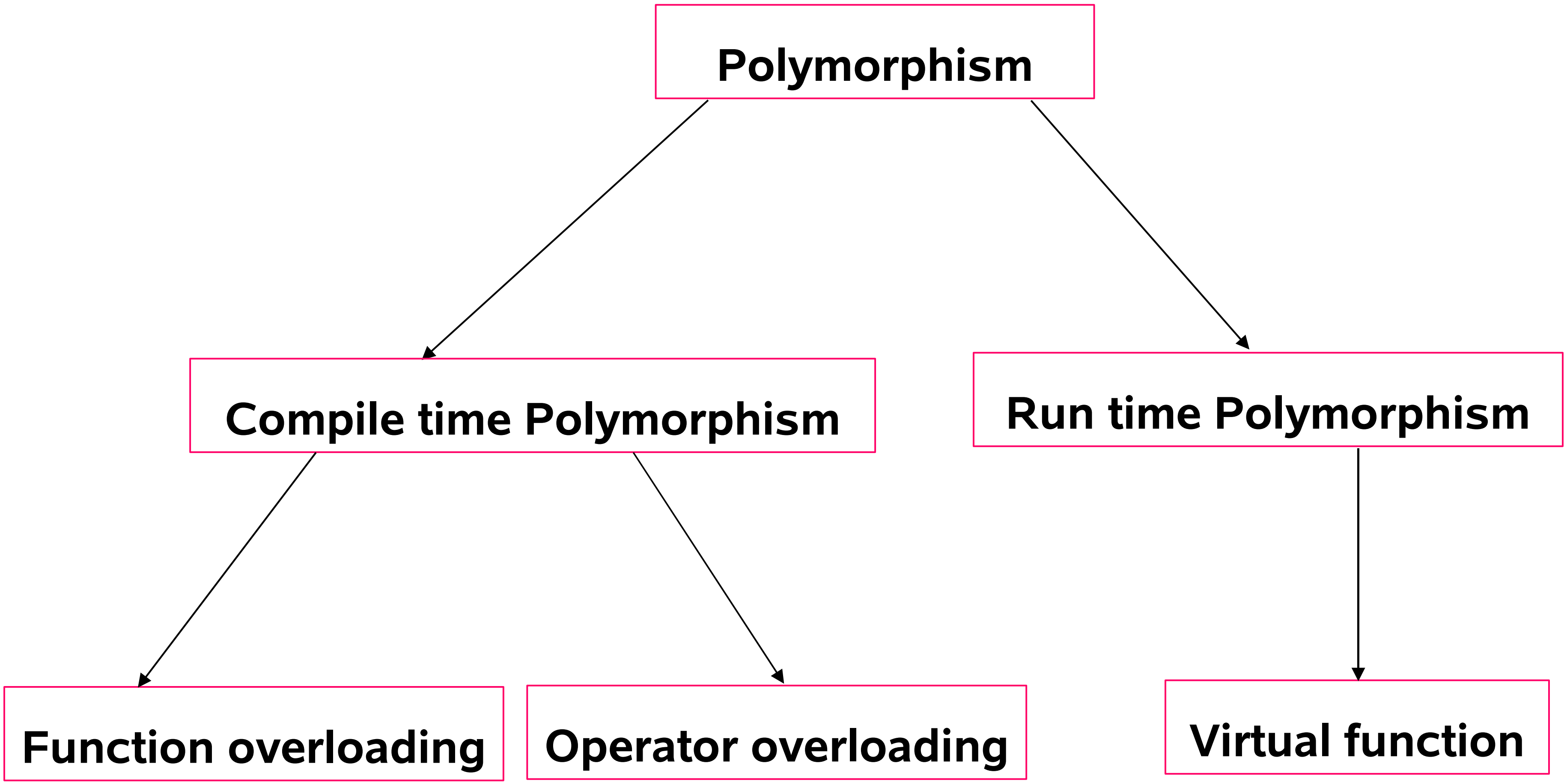
u = 5

v = 30

Virtual Function

- It is nice if the appropriate member function could be selected while the program is running. This is known as runtime polymorphism.
- C++ supports a mechanism known as virtual function to achieve run time polymorphism.
- Since the function is linked with a particular class much later after the compilation, this process is termed as late binding.
- It is also known as dynamic binding because the selection of the appropriate function is done dynamically at run time.

Virtual Function



Virtual Function

```
# include <iostream.h>
class BC    {
public :
int b;
void Show ( )
                    {    cout << "b =   "  <<  b   << " \n";     }
          };  // BC ends here
                    class DC : public BC    {
                              public :
                              int d;
          void Show ( )      {    cout << "b =   "  <<  b   << " \n";
 cout << "d =   "  <<  d   << " \n"; }       }; // DC ends here
```

Virtual Function

```
int main ( )
{         BC * bptr;                    // Base pointer
          BC Base;
          bptr = & Base;                // Base address


          bptr-> b = 100;     // access BC via base pointer
cout << “bptr points to base object \n”;
          bptr -> Show ( );

          // derived class
          DC Derived;
          bptr = & Derived;     // address of derived object
bptr -> b = 200;    // address of DC via base pointer
```

Virtual Function

```
// bptr -> d = 300; won't work

cout << "bptr now points to derived object \n";
bptr -> Show ( ); // bptr now points to derived object

// accessing d using a pointer of type derived class DC
DC * dptr;       // derived type pointer
dptr = & Derived;
dptr -> d = 300;


cout << "dptr is derived type pointer \n";
dptr -> Show ( ) ;
```

Virtual Function

```
cout << " using ( ( DC * ) bptr ) \n";
 ( ( DC * ) bptr )-> d = 400;
( ( DC * ) bptr )-> Show ( );
                        return 0;
            } // main ( ) end here.
```

Virtual Function

Output:

bptr points base object

***b* = 100**

bptr now points to derived object

***b* = 200**

dptr is derived type pointer

***b* = 200**

***d* = 300**

using ((DC *) bptr)

***b* = 200**

***d* = 400**

Virtual Function

```
# include <iostream.h>
class Base    {
public :
void Display ( );      {    cout << "Display Base \n";     }
virtual void Show ( )      {cout << "Show Base \n"; }

                }; // definition of Base class ends here.

                    class Derived  : public Base  {
                    public :
          void Display ( );      {    cout << "Display Base \n";     }
          void Show ( )      {cout << "Show Derived \n";   }
                    };  // definition of Derived class ends here.
```

Virtual Function

```
Int main ( )    {
        Base  B;
        Derived D;
        Base * bptr;            // Base address
            cout << “bptr points to Base \n”;
            bptr = &B;
            bptr -> Display ( );   // call Base version
            bptr -> Show ( );      // call Base version
                cout << “bptr points to Derived \n”;
                bptr = &D;
                bptr -> Display ( );   // call Base version
            bptr -> Show ( );      // call Derived version
                    return 0;
                } //  main ( ) ends here.
```

Virtual Function

Output:

bptr points Base

Display base

Show base

bptr points Derived

Display base

Show Derived`

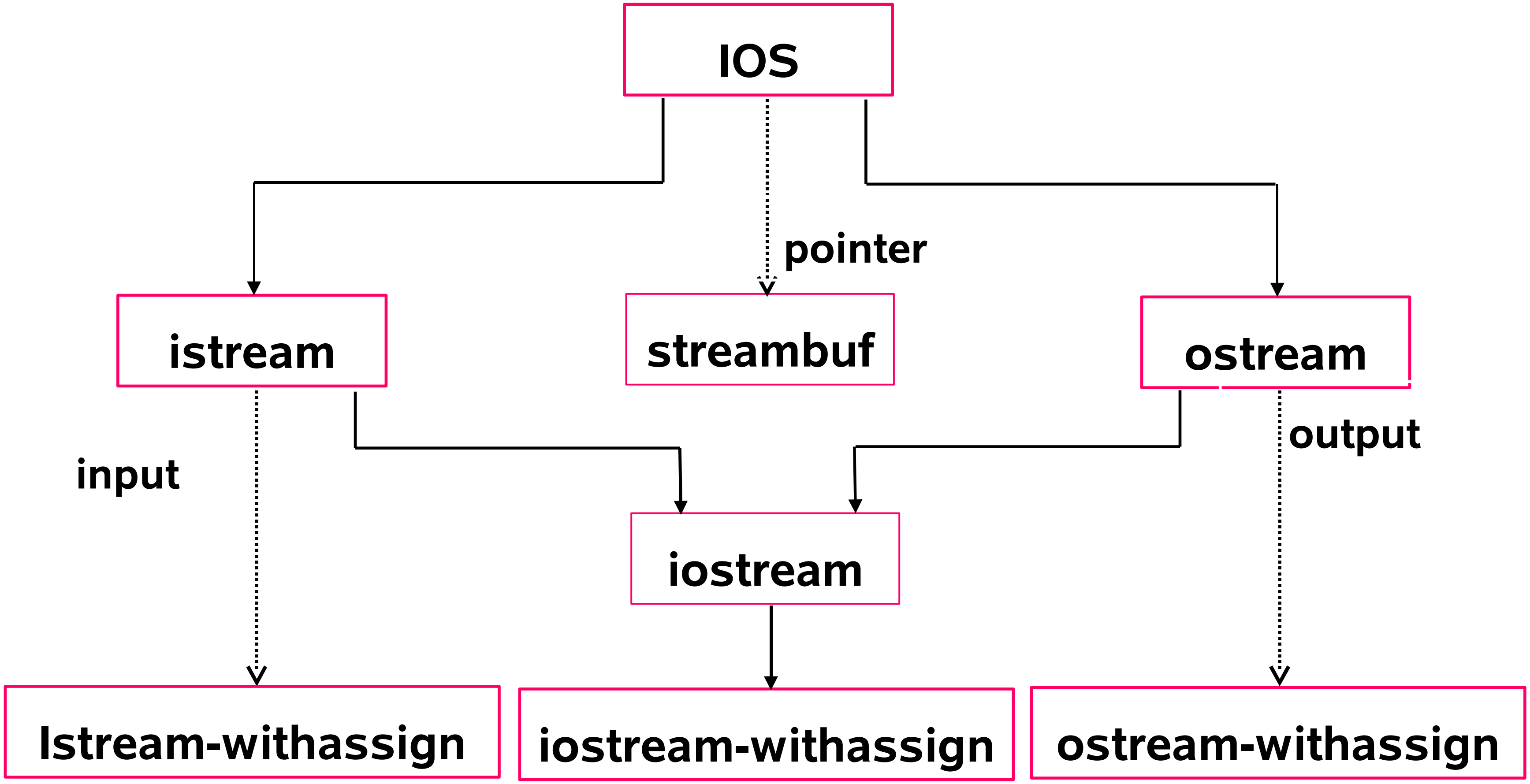
Pure Virtual Function

- A pure virtual function is a function declared in a base class that has no definition relative to the base class.
  In such cases, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function
- A class containing a pure virtual function is called abstract class.
  Remember that, a class containing pure virtual functions cannot be used to declare any objects of its own.

C++ Streams

- A **stream** is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent.
- The source stream that provides data to the program is called the **input stream** and the destination stream that receives output from the program is called the **output stream**.

C++ Streams



Stream classes for console I/O operations

C++ Streams

Class name	Contents
IOS (General input/ output class)	• contains basic facilities that are used by all other input and output classes • Also contains a pointer to a buffer object (streambuf object) • Declares constants and functions that are necessary for handling formatted input and output operations

C++ Streams

Class name	Contents
istream (input stream)	• inherits the properties of ios class • Declares input functions such as get ( ), getline ( ) • Contains overloaded extraction operator >>
ostream (output stream)	• inherits the properties of ios class • Declares output functions such as put ( ), write ( ) •Contains overloaded insertion operator <<

C++ Streams

Class name	Contents
iostream (input/output stream)	• inherits the properties of ios, istream and ostream through multiple stream inheritance and thus contains all the input and output functions.
streambuf	• Provides an interface to physical buffer. • Acts as a base for filebuf class used in ios

Unformatted I/O operations

```
# include <iostream.h>
int main ( )   {
int Count = 0;      char c;
cout << “Input Text\n”;      cin.get ( c );
while ( c != ‘\n’)
    {      cout.put ( c );
           Count ++;
           cin.get ( c );
     }
cout << “Number of Characters =   ” << Count”;
                    return 0;
                    }
```

Unformatted I/O operations

Input:

Object Oriented Programming

Output:

Object Oriented Programming

Number of characters = 27

Unformatted I/O operations

cin.getline (*line*, *size*);

- **This functions call invokes the function getline () which reads character input into the variable *line*.**
- **The reading is terminated as soon as either the newline character '\n' is encountered or *size* – 1 characters are read (whichever occurs first).**

Unformatted I/O operations

cout.write (*line*, *size*);

- **The first argument *line* represents the name of the string to be displayed and the second argument indicate *size* indicates the number of characters to display.**

Unformatted I/O operations

```
#include <iostream.h>
#include <string.h>
int main ( )    {
char * String1 = "C++  ";          char * String2 = "Programming ";
int m = strlen ( String1 );         int n = strlen ( String2 );
    for ( int i = 1; i < n; i++ )
    {    cout.write ( String2, i );
         cout << "\n";        }
```

Unformatted I/O operations

```
for ( int i = n; i > 0; i-- )
    {    cout.write ( String2, i );
         cout << "\n";    }
// concatenating strings
cout.write ( String1, m ).write ( String2, n );

cout.write ( String1, 11 );
    return 0;
} // main () ends here.
```

Unformatted I/O operations

Output:

```
p
p r
p r o
p r o g
p r o g r
p r o g r a
p r o g r a m
p r o g r a m m
p r o g r a m m i
p r o g r a m m i n
p r o g r a m m i n g
p r o g r a m m i n
p r o g r a m m i
p r o g r a m m
p r o g r a m
p r o g r a
p r o g r
p r o g
p r o
p r
p
```

Unformatted I/O operations

Output:

C++ Programming
C++Progra

Note:

cout.write (*string1*, *m*).write (*string2, n*);

is equivalent to the following two statements:

cout.write (*string1*, *m*);
cout.write (*string2*, *n*);

Formatted Console I/O operations

Function	Task
width ( )	To specify the required field size for displaying an output value
precesion ( )	To specify number of digits to be displayed after the decimal point of a float value.
fill ( )	To specify a character that is used to fill the unused portion of a field.
setf ()	To specify format flags that can control the form of output display (such as left-justification and right-justification).
unsetf ()	To clear the flags specified.

ios format functions

Formatted Console I/O operations

Manipulators	Equivalent ios function
setw ( )	width ( )
setprecesion ( )	precesion ( )
setfill ( )	fill ( )
setiosflagsf ()	setf ()
setiosflagsf ()	unsetf ( )

Manipulators

Uses of Width ()

```
#include <iostream.h>
int main ( )
{
int Items [ 4 ] = { 10, 8, 12, 15 };
int Cost [ 4 ] = { 75, 100, 60, 99 };
cout.width ( 7 );
cout << "ITEMS"

cout.width ( 15 );
cout << "Total Values  " << endl;
```

Uses of Width ()

```
int Sum = 0;
            for ( int i = 0; i < 4; i ++ )   {
            cout.width ( 5 );
            cout.Items [ i ];

            cout.width ( 8 );
            cout.Cost  [ i ];
      int Value = Item [ i ] * Cost [ i ];
            cout.width ( 15 );
            cout << Value  <<endl;
            Sum = Sum + Value;
                                  }  // for loop ends here.
```

Uses of Width ()

```
cout << "\n Grand Total";
cout.width ( 2 );
cout << Sum << "\n";
                    return 0;
                              } // main () ends here.
```

Uses of Width ()

```
# include <iostream.h>
# include <cmath.h>
int main ( )
        {
cout << “precession set to 3 digits \n\n”;
cout << precession ( 3 );


cout.width ( 10 );
cout << “Value”;
cout.width ( 15 );
cout << “Sqrt of value  ”<< endl;
```

Uses of Width ()

```
for ( int n = 1; n < = 5; n ++ )   {
cout.width ( 8 );
cout << n;
cout.width ( 13 );
cout << sqrt ( n ) << "\n";          } // for loop end here
cout << "\n Precession set to 5 digits \n \n";
cout << precision ( 5 ) ; // precision parameter changed
cout << "sqrt ( 10 ) =   " << sqrt ( 10 ) << "\n\n";

cout << precision ( 0 ) ; // precision set to default
cout << "sqrt ( 10 ) =   " << sqrt ( 10 ) << "default setting\n";
return 0;
}  // main ( ) ends here
```

Uses of Width ()

Output:

```
Precision set to 3 digits
VALUE          SQRT-OF-Value
   1                    1
   2                 1.41
   3                 1.73
   4                    2
   5                 2.24
Precision set to 5 digits

sqrt ( 10 ) = 3.1623
sqrt ( 10 ) = 3.162278 ( default setting )
```

Uses of Width ()

```
# include < iostream.h>
int main ()
{          cout.fill ( '<');
           cout.precision ( 3 );
for ( int n = 1; n <= 6; n ++ )                    {
cout.width ( 5 );          cout << n;
           cout.width ( 10 );
           cout << 1.0 / float ( n ) << "\n";
if ( n == 3 )  cout.fill ('>');         // fill ( ) with '>'      }

cout << "\m padding Changed" << "\n\n";
cout.fill ( '#');        // fill ( ) reset
cout.width ( 15 );            cout << 12.345678;
           return 0;          } / / main ( ) ends here
```

Uses of Width ()

Output:

```
<<<<1<<<<<<<<<1
<<<<2<<<<<<<0.5
<<<<3<<<<<0.333
<<<<3<<<<<0.333
>>>>4>>>>>>0.25
>>>>5>>>>>>>0.2
 >>>>5>>>>>0.167

Padding Changed

###########12.3
```

Working with Files

A program typically involves either or both of the following kinds of data communication:

- Data transfer between the console unit and the program.

- Data transfer between the program and a disk file.

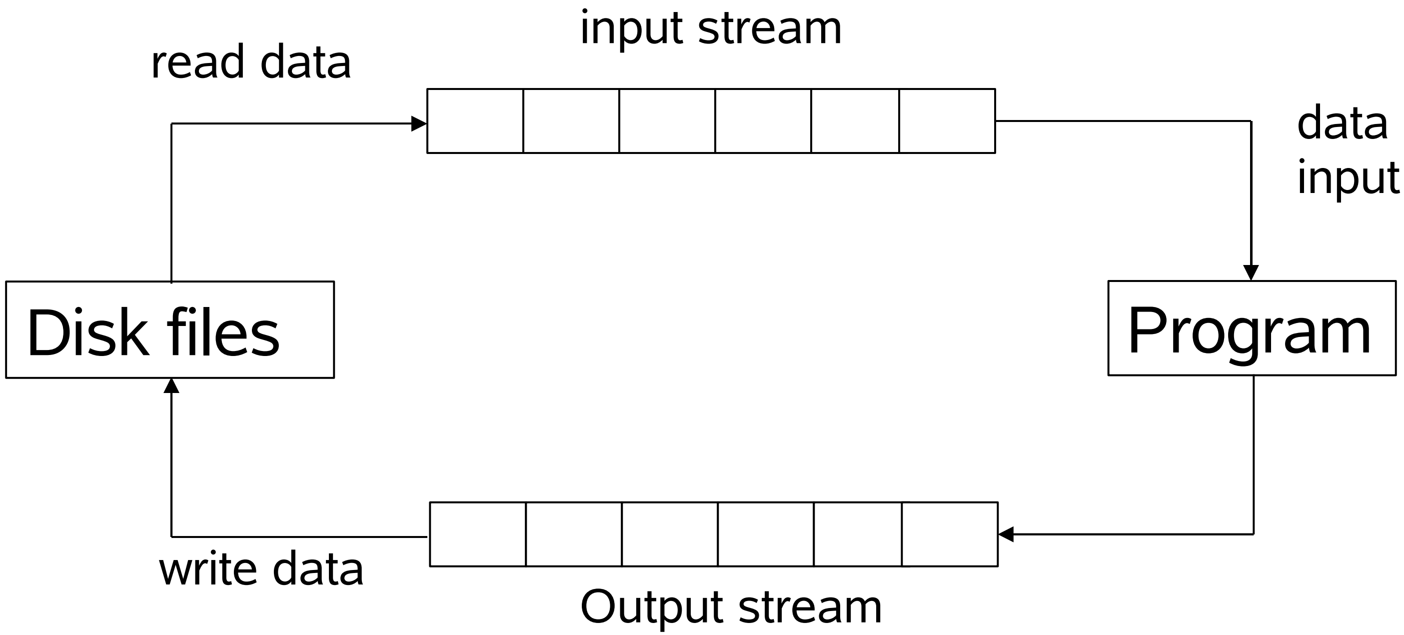
Working with Files

Stream is a sequence of bytes.

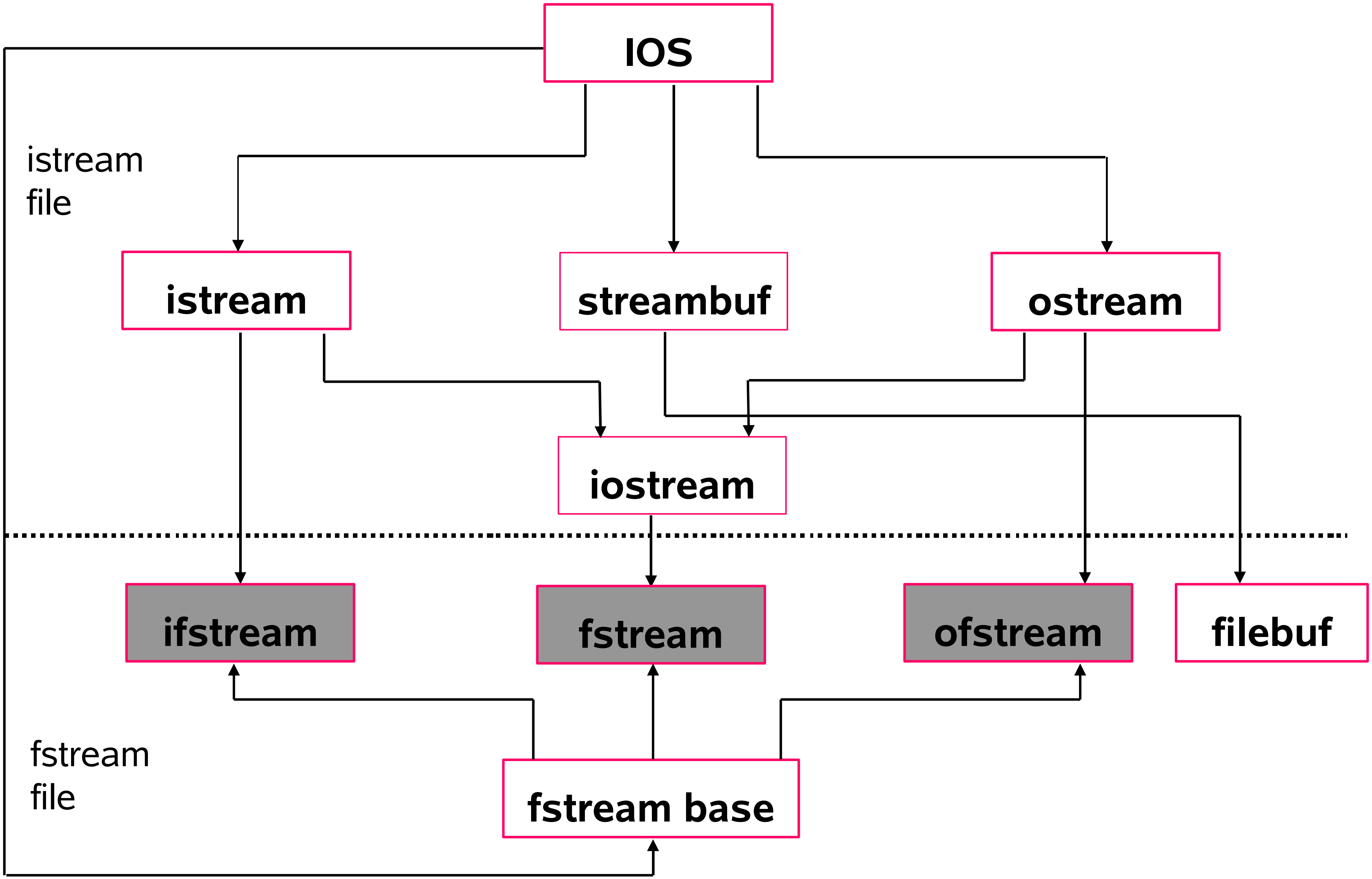
The stream that supplies data to the program is known as *input stream* and the one that receives data from the program is known as *output stream*.

In other words, the input stream extracts (or reads) data from the file and the output stream inserts (or writes) data to the file.

Working with Files



File input and output streams



Stream classes for console I/O operations

Details of File Stream Classes

Class name	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains Openprot constant used in the open() of file stream classes. Also contain close ( ) and open ( ) as members.
fstream base	Provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream class. Contain close ( ) and open ( ) as members.

Details of File Stream Classes

Class name	Contents
ifstream	Provides input operations. Contains open () with default input mode. Inherits the functions get (), getline (), read (), Seekg() and tellg () functions from istream.
ofstream	Provides input operations. Contains open () with default output mode. Inherits the functions put (), write (), Seekp() and tellp () functions from ostream.

Details of File Stream Classes

Class name	Contents
fstream	Provides support for simultaneous input and output operations. Contains open () with default input mode. Inherits all the functions from istream and ostream classes through iostream.

Opening Files

A file stream can be defined using the classes ifstream, ofstream, and fstream that are contained in the header file fstream.

The class to be used depends upon the purpose, that is, whether we want to read data from file or write data to it. A file can be opened in two ways:

- Using the constructor function of the class.
- Using the member function open ( ) of the class.

Opening Files Using Constructor

ofsteam *Outfile* ("Result"); // Output only

This creates *Outfile* as an ofstream object that manages the output stream. This object can be any valid C++ name such as *O-File*, *My-File* or *Fout*. This statement also opens the file results and attaches it to the output stream *Outfile*.

The program may contain statements like:

Outfile << "TOTAL\n";

Outfile << *Sum*<<"\n";

Opening Files Using Constructor

Similarly, the following statement declares *Infile* as an ifstream object and attaches it to the file data for reading (input)

ifsteam *Infile* (“Data”);   // Input only

The program may contain statements like:

Outfile << *Number*”;

Outfile << *Sum*;

Infile >> *Number*;

Infile >> *Sum*;

Opening Files Using Constructor

```
#include<iostream.h>
#include<fstream.h>
int main ( )
{
ofstream Outf ( “ITEM” );  // connect ITEM file to Outf
cout << “Enter Item Name”;
char Name[ 30 ]; cin >> Name;
Outf  << Name << “\n”;  // write to file ITEM
float Cost; cin >> Cost;
Outf  << Cost << “\n”;
Outf.close ( );     // Disconnect ITEM file to Outf
```

Opening Files Using Constructor

```
ifstream Inf ("ITEM"); // connect ITEM file to Inf
Inf >> Name;    // read Name from file ITEM
Inf >> Cost;
cout << "\n";
cout << "Item Name: " << Name << "\n";
cout << "Item Cost: " << Cost << "\n";
Inf.close ( );    // Disconnect ITEM file to Inf
        return 0;
                    } // main ( ) ends here.
```

Opening Files Using Constructor

Output:

Enter Item Name: CD-ROM

Enter Item Cost: 250

Item Name: CD-ROM

Item Cost: 250

Caution!

When a file is opened for writing only, a new file is created if there is no file of that name. If a file by that name exists already, then its contents are deleted and the file is presented as a clean file.

Opening Files Using Open ( )

```
#include<iostream.h>
#include<fstream.h>
int main ( )
{
ofstream Fout;
Fout.open ("Country");
Fout << "United States of America\n";
Fout << "United Kingdom\n";
Fout << "South Korea\n";
Fout.close ( );
```

Opening Files Using Open ( )

```
Fout.open ("Capital");
Fout << "Washington\n";
Fout << "London\n";
Fout << "Seoul\n";
Fout.close ( );
const int N = 80;
char Line [ N ];
ifstream Fin;
Fin.open ( "Country");
cout << "Contents of Country File";
```

Opening Files Using Open ( )

```
                While ( Fin )      // check end-of-file
                {   Fin.getline ( Line, N );
                    cout << Line;      }
                Fin.close ( );

Fin.open ( "Capital");    cout << "Contents of Capital File";

                While ( Fin )      // check end-of-file
                {   Fin.getline ( Line, N );
                    cout << Line;    }
                Fin.close ( );
                              return 0;
                      }    // main ( ) ends here.
```

Opening Files Using Constructor

Output:

Contents of Country File

United States of America

United Kingdom

South Korea

Contents of Capital File

Washington

London

Seoul

Opening Files Using Open ( )

```
#include<iostream.h>
#include<fstream.h>
#include <stdlib.h>
int main ( )
{   const int SIZE = 80;
char Line [ SIZE ];
ifstream Fin-1, Fin-2;
Fin-1.open ("Country");
Fin-2.open ("Capital");
```

Opening Files Using Open ( )

```
for ( int i = 1; i <= 10; i++ )
{
   if ( Fin-1.eof ( ) != 0 )
  {  cout << “Exit from Country\n”;
         exit ( 1 );
   }
Fin-1.getline ( Line, SIZE );
cout << “Capital of ”  << Line<< endl;
```

Opening Files Using Open ( )

```
if ( Fin-2.eof ( ) != 0 )
   {  cout << "Exit from Capital\n";
            exit ( 1 );
    }
Fin-2.getline ( Line, SIZE );
cout << Line << "\n";
}   // for-loop ends here
                   return 0;
}  // main ( ) ends here
```

Opening Files Using Open ( )

Output:

Capital of United States of America

Washington

Capital of United Kingdom

London

Capital of South Korea

Seoul

More About Open ( ): File Modes

Stream-Object.open (“filename”, mode);

More About Open ( ): File Modes

Parameter	Meaning
ios:: app	Append to end-of-file
ios :: ate	Go to end-of-file on opening
ios :: binary	Binary file
ios :: in	Open file for reading only
ios :: nocreate	Open fails if the file does not exist
ios :: noreplace	Open fails if the file already exists
ios :: out	Open file for writing only
ios :: trunc	Delete the contents of the file if it exists

More About Open ( ): File Modes

1. Opening a file in ios :: out mode also opens it in the ios :: trunc mode by default.

2. Both ios :: app and ios :: ate take us to the end of the file when it is opened. The difference between the two parameters is that the ios :: app allows us to add data to the end of the file only, while ios :: ate mode permits us to add data or to modify the existing data anywhere in the file. In both the cases, a file is created by the specified name, if it does not exist.

More About Open ( ): File Modes

1. The parameter **ios :: app** can be used only with the files capable of output.

2. Creating a stream using **ifstream** implies input and creating a stream using **ofstream** implies input and creating a stream **ofstream** implies output. So in these cases it is not necessary to provide the mode parameters.

3. The **fstream** class does not provide a mode default and therefore, we must provide the mode explicitly when using an object of **ofstream** class.

4. The mode can combine two or more parameters using the bitwise OR operator (symbol |) shown below:

More About Open ( ): File Modes

```
Fout.open ( "Data", ios :: app, ios :: nocreate );
```

This opens the file in the append mode but fails to open the file if it does not exist.

File Pointers and their Manipulations

Each file has two associated pointers known as the file pointers. One of them is called the input pointer (or get pointer) and the other is called the output pointer (or put pointer).

Each time an input or output operation takes place, the appropriate pointer is automatically advanced.

File Pointers and their Manipulations

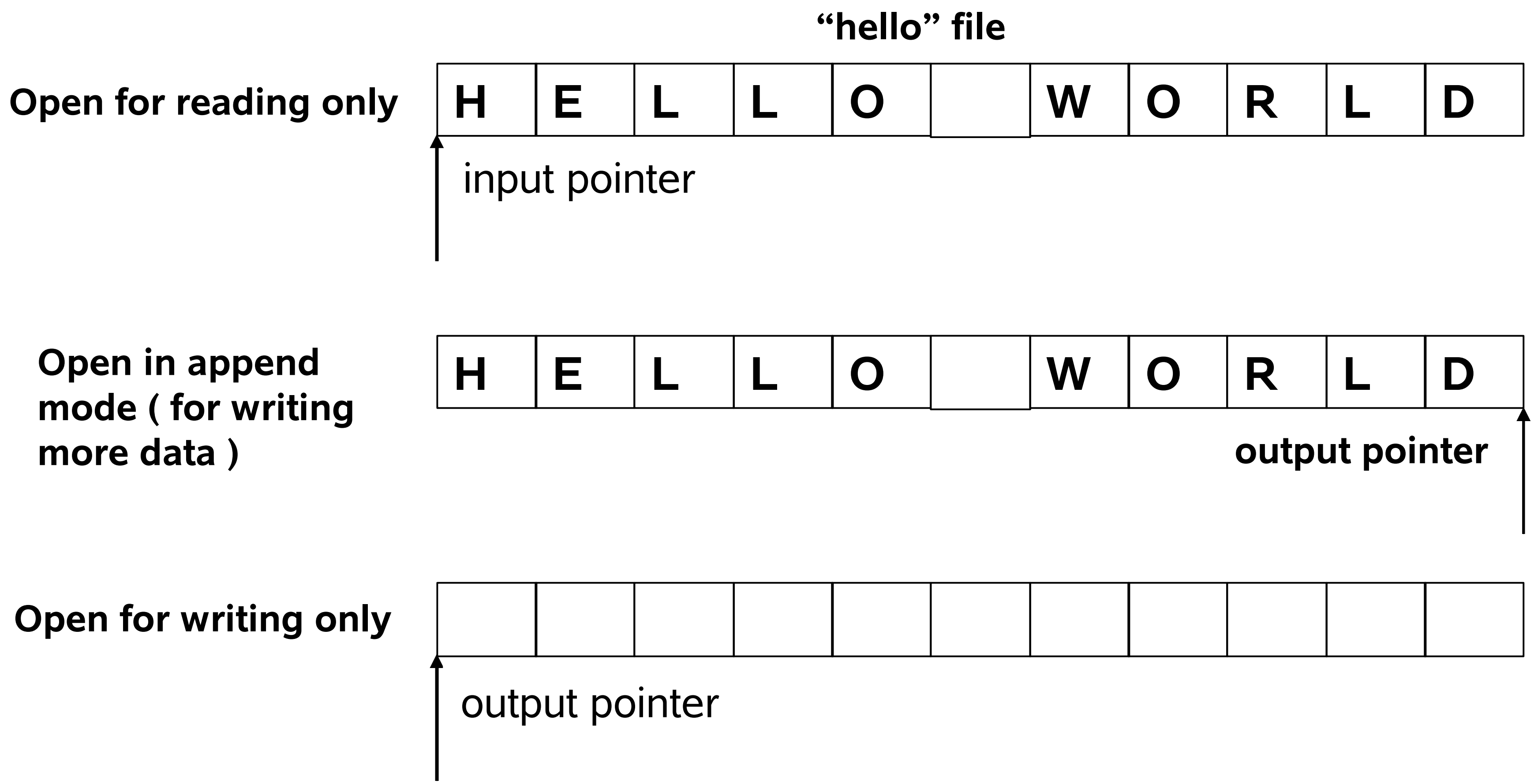


Fig : Action on file pointers while opening a file

File Pointers and their Manipulations

- **seekg ()** Moves **get pointer** (input) to a specified location.
- **seekp ()** Moves **put pointer** (output) to a specified location.
- **tellg ()** Gives the current position on the **get pointer**.
- **tellp ()** Gives the current position on the **put pointer**.

File Pointers and their Manipulations

For example, the statement

Outfile.seekp (10);

moves the file pointer to the byte number 10. Remember, the bytes in a file are numbered beginning from zero. Therefore, the pointer will be pointing to the 11th byte in the file.

Consider the following statements:

```
ofstream Fileout;
Fileout.open ("hello", ios :: app);
int p = Fileout.tellp ( );
```

File Pointers and their Manipulations

Consider the following statements:

```
ofstream Fileout;
Fileout.open ("hello", ios :: app);
int p = Fileout.tellp ( );
```

On execution of these statement, the output pointer is moved to the end of the file "Hello" and the value of *p* will represent the number of bytes in the file.

Specifying the Offset

seekg (offset, *refposition*);

seekp (*offset*, *refposition*);

The parameter *offset* represents the number of bytes the file pointer is to be moved from the location specified by the parameter *refposition*.

The refposition takes one of the following three constants defined in the ios class:

• ios :: beg	start of the file
• ios :: cur	current position of the pointer
• ios :: end	end of the file

Specifying the Offset

Seek call	Action
Fout.seekg (0, ios :: beg)	Go to start
Fout.seekg (0, ios :: cur)	Go to the current position
Fout.seekg (0, ios :: end)	Go to the end of file
Fout.seekg (*m*, ios :: beg)	Move to (*m* + 1) th byte in the file

Specifying the Offset

Seek call	Action
Fout.seekg (*m*, ios :: cur)	Go forward by *m* bytes from the current position
Fout.seekg (-*m*, ios :: cur)	Go backward by *m* bytes from the current position
Fout.seekg (-*m*, ios :: end)	Go backward by *m* bytes from the end

Sequential I/O Operations

```
#include <iostream.h>
#include <fsttream.h>
#include < string.h>
int main ( )
{
  char String [ 80 ];
  cout << “Enter a String \n”;
  cin >> String;
  int Len = strlen ( String );
  fstream File;
File.open ( “TEXT”, ios :: in | ios :: out ); // input and output
                                                  stream
```

Sequential I/O Operations

```
        for ( int i = 0; i < Len; i++ )
        File.put (String [ i ] );    // put a character to file
File.seekg ( 0 );      // go to the start
        char Ch;
        while ( File )
        {
        File.get ( Ch );      // get a character from file
        cout << Ch;        // display it on screen
        }
                return 0;
        }   // main ( ) ends here
```

Sequential I/O Operations

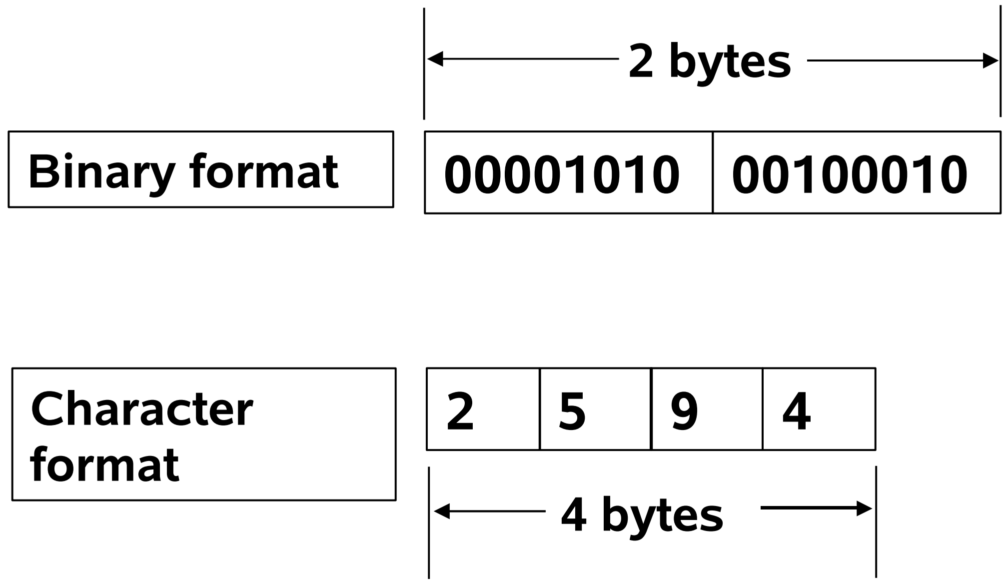
Output:

Enter a string	
Object-Oriented-Programming-Through-C++	**input**
Object-Oriented-Programming-Through-C++	**output**

write () and read () Functions

The functions write ( ) and read ( ), unlike the functions put ( ) and get ( ), handle the data in binary form. This means that the values are stored in the disk file in the same format in which they are stored in the internal memory. An int takes two bytes two store its value in the binary form, irrespective of its size. But a 4 digit int will take four bytes to store it in the character form. Fig 2 shows how an int value 2594 is stored in the binary and character formats.

Binary and character formats of an integer value



Binary and character formats of an integer value

write () and read () Functions

***Infile*.read ((char *) & *V*, sizeof (*V*));**

***Onfile*.write ((char *) & *V*, sizeof (*V*));**

These functions take two arguments. The first is the address of the variable V, and the second is the length of that variable in bytes. The address of the variable must be cast to type char * (i.e. pointer to character type).

Write () and read () Functions

Infile.read ((char *) & *V*, sizeof (*V*));

Onfile.write ((char *) & *V*, sizeof (*V*));

These functions take two arguments. The first is the address of the variable *V*, and the second is the length of that variable in bytes. The address of the variable must be cast to type char * (i.e. pointer to character type).

Write () and read () Functions

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
const char * Filename = “Binary”;
int main ( )
{
float Height [ 4 ] = { 175.5, 153.0, 167.25, 160.70};
ofstream Outfile;
Outfile.open ( “Filename”, ios :: binary | ios :: out );
Outfile.write ( ( char * ) Height, sizeof ( Height ) );
Outfilre.close ( );  // close the file for reading
```

Write () and read () Functions

```
for ( int i = 0; i <= 4; i++ )
Height [ i ] = 0;  // clear array from memory
ifstream Infile;  Infile.open ( "FileName", ios :: binary | ios :: in );
Infile.read ( ( char * ) Height, sizeof ( Height ) );
for ( int i = 0; i <= 4; i++ )
cout.setf ( ios :: showpoint );
cout << setw ( 10 ) << setprecision ( 2 )
                  << Height [ i ];
}
Infile.close ( );                     return 0;
                        }  // main () ends here
```

Write () and read () Functions

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
class Inventory
{
char Name [ 10 ];
int Code;    float Cost;
public:
void Read-Data ( void );
 void Write-Data ( void );
 };      // class definition ends here
```

Write () and read () Functions

```
void Inventory :: Read-Data ( Void )
{
cout << "Enter Name ";        cin >> Name;
cout << "Enter Code ";        cin >> Code;
cout << "Enter Cost ";        cin >>
Cost;
}
```

Write () and read () Functions

```
void Inventory :: Write-Data ( Void )
{
cout << setioflags ( ios :: left )
        << setw ( 10 ) << Name
        <<  setioflags ( ios :: right )
        << setw ( 10 ) << Code
        << setprecision ( 2 )
         << setw ( 10 ) << Cost
         << endl;
 }
```

Write () and read () Functions

```
int main ( )
{
Inventory Item [ 3 ];
fstream File;
File.open ( "Stock.data", ios :: in | ios :: out  | ios :: binary )
cout << "Enter Details For Three Items \n";
for ( int i = 0; i < 3; i++ )
{
Item [ i ].Read-Data ( );
File.write ( ( char * ) & Item [ i ], sizeof (Item [ i ] ) );
}
```

Write () and read () Functions

```
        File.seekg ( 0 );
        for ( int i = 0; i < 3; i++ )
        {
File.read ( ( char * ) & Item [ i ], sizeof ( Item [ i ] ) );
Item [ i ].Write-Data ( );
}
File.close ( );
                return 0;
        }  // main ( ) ends here
```

Write () and read () Functions

Enter Details For three Items

Enter Name: C++

Enter Code: 101

Enter Cost: 175

Enter Name: FORTAN

Enter Code: 102

Enter Cost: 160

Enter Name: JAVA

Enter Code: 103

Enter Cost: 275

Write () and read () Functions

Output

C++	101	175
FORTAN	102	160
JAVA	103	275

Write () and read () Functions

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
class Inventory
{
char Name [ 10 ];
int Code;   float Cost;
public:
void Read-Data ( void );
void Write-Data ( void );
};      // class definition ends here
```

Write () and read () Functions

```
void Inventory :: Read-Data ( Void )
{
cout << "Enter Name ";          cin >> Name;
cout << "Enter Code ";          cin >> Code;
cout << "Enter Cost ";          cin >>
Cost;
}
```

Write () and read () Functions

```
void Inventory :: Write-Data ( Void )
{
cout << setioflags ( ios :: left )
        << setw ( 10 ) << Name
        <<  setioflags ( ios :: right )
        << setw ( 10 ) << Code
        << setprecision ( 2 )
        << setw ( 10 ) << Cost
        << endl;
 }
```

Write () and read () Functions

```
int main ( )
{
Inventory Item;
fstream File;
File.open ( "Stock.data", ios :: ate | ios :: in | ios :: out | ios :: binary );
File.seekg ( 0, ios :: beg );
cout << "Current Contents of Stock \n";
while ( File.read ( ( char * ) & Item , sizeof ( Item  ) )
Item.Write-Data ( );
```

Write () and read () Functions

```
// Add More Items
cout << “\nAdd an Item\n”;
Item.Read-Data ( );
char Ch;
cin.get ( Ch );

File.write ( ( char * ) & Item, sizeof ( Item ));
```

Write () and read () Functions

```
// Display the append file
File.seekg ( 0 );
cout << "Contents of Appended File";
while ( File.read ( ( char * ) & Item , sizeof ( Item ) ) )
                    Item.Write-Data ( );
// Find Number of objects in the File
int Last = File.tellg ( );
int N = Last / sizeof ( item );
```

Write () and read () Functions

```
cout << “Number of Objects = ” << N << “\n”;
cout << “Total Bytes in the File ” << Last << “\n”;
    // Modify The Details of an Item
cout << “Enter Object to be Modified” ;
int Object;
cin >> Object;
cin.get ( Ch );
int Location = ( Object - 1 ) * sizeof ( Item );
if ( File.eof ( ) )
File.clear ( );
```

Write () and read () Functions

```
File.seekp ( Location );
cout << "Enter New Values of the Object \n";
Item.Read-Data ( );
cin.get ( Ch );
File.write ( ( char * ) & Item, sizeof ( Item ));
```

Write () and read () Functions

```
            // Show Updated file
File.seekg ( 0 );                 // go to the start
cout << "Contents of the updated File \n";
while ( File.read ( ( char * ) & Item , sizeof ( Item ) ) )
            Item.Write-Data ( );
File.close ( );
            return 0;
            } // main ( ) ends here.
```

Write () and read () Functions

Current Contents Of Stock

C++	101	175
FORTAN	102	160
JAVA	103	275
PASCAL	104	375
COBOL	105	278

Add An Item

Enter Name: PERL

Enter Code: 106

Enter Cost: 398

Write () and read () Functions

Contents of Appended File

C++	101	175
FORTAN	102	160
JAVA	103	275
PASCAL	104	375
COBOL	105	278
PERL	106	398

Number of Objects = 6

Total Bytes in the Files = 96

Enter Object Number to be Updated

6

Write () and read () Functions

Enter New Values of the Object

Enter Name: C#

Enter Code: 107

Enter Cost: 178

Contents of Updated File

C++	101	175
FORTAN	102	160
JAVA	103	275
PASCAL	104	375
COBOL	105	278
PERL	106	398
C#	107	178

Error Handling During file operations

Function	Return value and meaning
eof ( )	Returns true (non-zero value) if end-of-file is encountered while reading; Otherwise returns false.
fail ( )	Returns true when an input or output operation has failed.
bad ( )	Returns true if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is false, it may be possible to recover from any other error reported, and continue operation.

Error Handling During File Operations

Function	Return value and meaning
good ( )	Returns true if no error has occurred. This means, all the above functions are false. For instance, if file.good ( ) is true, all is well with the stream *File* and we can proceed to perform I/O operations. When it returns false, no further operations can be carried out.

Recall: The function clear ( ) resets the error state so that further operations can be attempted.

Command-Line Arguments

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main ( int Arg, char * Arv [ ] )
{
int Number [ 9 ] = { 11, 22, 33, 44, 55, 66, 77, 88, 99 };
	if ( Argc ! = 3 )	{
		cout << “Argc =  ”<< Argc << “\n”;
		cout << “Errors in arguments \n”;
		exit ( 1 );	}
```

Command-Line Arguments

```
ofstream Fout-1, Fout-2;
    Fout-1.open ( Argv [ 1 ] );
        if ( Fout-1.fail ( ) )
    {
        cout << "Could Open The File";
                << Argc [ 1 ] << "\n";
        exit ( 1 );
    }
```

Command-Line Arguments

```
Fout-2.open ( Argv [ 2 ] );
if ( Fout-2.fail ( ) )
    {
        cout << “Could Open The File”;
        << Argc [ 2 ] << “\n”;
    exit ( 1 );
        }
```

Command-Line Arguments

```
for ( int i = 0; i < 9; i ++ )
{
 if ( Number [ i ] % 2 == 0 )
Fout-2 << Number [ i ] << " ";        // Write to ODD File
else
Fout-1 << Number [ i ] << " ";        // Write to EVEN File
}    // for-loop ends here
File-1.close ( );
File-2.close ( );
ifstream Fin;        char Ch;
```

Command-Line Arguments

```
for ( int i = 0; i < Argc; i ++ )
{
 Fin.open ( Argv [ i ] )
cout << "Contents of " << Argv [ i ] << "\n";
do {
Fin.get ( Ch );   // read a value
cout << Ch;   // display it
    }    while ( Fin );
cout << " \n \n ";              Fin.close ( );
    return 0;                   } // main ( ) ends here
```

Command-Line Arguments

Output:

Contents of ODD

11 33 55 77 99

Contents of EVEN

22 44 66 88

Templates

Templates is of one of the features added to C++ recently. It is new concept which enables us to define generic classes and functions and thus provides support for generic programming.

Generic programming is an approach where <u>generic types</u> are used as <u>parameters in algorithms</u> so that they work for a variety of suitable <u>data types</u> and <u>data structures</u>.

Templates

A template can be used to create a family of classes or functions.

For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array.

Similarly, we can define a template for a function, say Mul ( ), that would help us create various versions of Mul ( ) for multiplying int, float and double type values.

Templates

A template can be considered as a kind of macro.

When an object of a specific type is defined for actual use, the template definition for that class is substituted with required data type.

Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.

Class Templates

```
class Vector
{       int *v;
        int Size;
    public:
        Vector ( int m )      // Create a null vector
        {
            v = new int [ Size = m ] ;
            for ( int i = 0; i < Size; i++ )
                v [ i ]  = 0;
        }
```

Class Templates

```
Vector ( int *a )     //  create a vector from an array
{
        for ( int i = 0; i < Size; i++ )
                v [ i ]  = a [ i ];
}

int operator* ( Vector &y )     //  scalar product
        {     int Sum = 0;
for ( int i = 0; i < Size; i ++ )
        Sum += this->v [ i ] * y.v [ i ];
                return Sum;
        }
        }; // class definition ends here
```

Class Templates

```
int main ( )
{
int x [ 3 ] = { 1, 2, 3 };
int y [ 3 ] = { 4, 5, 6 };
Vector V-1 ( 3 );
Vector V-2 ( 3 );

V-1 = x;  V-2 = y;
int R = V-1 * V-2;
cout << “R = ”<< R;
return 0;
}  // main ( ) ends here
```

Class Templates

```
template < class T >
class Vector
{          T *v;      // type T vector
                int Size;
                public:
Vector ( int m )      // Create a null vector
{
          v = new T [ Size = m ] ;
          for ( int i = 0; i < Size; i++ )
                    v [ i ]  = 0;
}
```

Class Templates

```
Vector ( T *a )      //   create a vector from an array
{
        for ( int i = 0; i < Size; i++ )
                v [ i ]  = a [ i ];
}

T operator* ( Vector &y )      //  scalar product
        {     T Sum = 0;
for ( int i = 0; i < Size; i ++ )
        Sum += this->v [ i ] * y.v [ i ];
                return Sum;
        }
        }; // class definition ends here
```

Class Templates

```
int main ( )
{
int x [ 3 ] = { 1, 2, 3 };
int y [ 3 ] = { 4, 5, 6 };
Vector < int >V-1;
Vector < int >V-2;
V-1 = x;
V-2 = y;
int R = V-1 * V-2;
cout << “R = ”<< R << endl;
return 0;
} // main ( ) ends here
```

Class Templates

```
template < class T >
class Vector
{         T *v;
int Size;
               public:
Vector ( int m )      // Create a null vector
{
          v = new T [ Size = m ] ;
          for ( int i = 0; i < Size; i++ )
                    v [ i ]  = 0;
}
```

Class Templates

```
Vector ( T *a )      //   create a vector from an array
{
        for ( int i = 0; i < Size; i++ )
                v [ i ]  = a [ i ];
}

T operator* ( Vector &y )      //  scalar product
        {     T Sum = 0;
for ( int i = 0; i < Size; i ++ )
        Sum += this->v [ i ] * y.v [ i ];
                return Sum;
        }
        }; // class definition ends here
```

Class Templates

```
int main ( )
{
float  x [ 3 ] = { 1.1, 2.2, 3.3 };
float y [ 3 ] = { 4.4, 5.5, 6.6 };
Vector < float >V-1;
Vector < float >V-2;
V-1 = x;
V-2 = y;
    int R = V-1 * V-2;
    cout << “R = ”<< R << endl;
        return 0;
} // main ( ) ends here
```

Class Templates With Multiple Parameters

```
#include < iostream.h >
template < class T-1, class T-2 >
class Test
{
T-1  a;
T-2  b;
            public:
Test ( T-1 x, T-2 y )
{ a = x; b = y; }
void Show ( )
{  cout << a << " and " << b << "\n";  }
}; // class definition ends here
```

Class Templates With Multiple Parameters

```
int main ( )
{
Test < float, int > Test-1 ( 1.23, 123 );
Test < int, char > Test-2 ( 100, ‘W’);
Test-1.Show ( );
Test-2.Show ( );
            return 0;
}
```

Function Template

```
# include < iostream.h>
template < class T >
void Swap ( T &x, T &y )
{
        T Temp = x;
        x = y;
        y = Temp;
}
```

Function Template

```
void Fun ( int m, int n, float a, float b )
{
cout << "m and n before swap:  " <<
          m  << "  "  << n << "\n";
Swap ( m, n );
cout << m and n after swap: " <<
          m  << "  "  << n << "\n";
cout << "a and b before swap:  " <<
          a  << "  "  << b << "\n";
Swap ( a, b );
cout << a and b after swap: " <<
          a  << "  "  << b << "\n";
}
```

Function Template

```
int main ( )
{
fun ( 100, 200, 11.22, 33.44 )
                    return 0;
}
```

Function Template

Output

m **and** ***n*** **before swap: 100 200**

m **and** ***n*** **after swap: 200 100**

a **and** ***b*** **before swap: 11.22 33.439999**

a **and** ***b*** **after swap: 33.439999 11.22**

Bubble Sort Using Template Functions

```
# include < iostream.h>
template < class T >
void Bubble ( T a [ ], int n )
{
for ( int i = 0; i < n - 1; i ++ )
        for ( int j = n - 1; i < j; j -- )
                if ( a [ j ] < a [ j - 1 ] )
                        Swap (a [ j ], a [ j - 1 ] );
} // Bubble ends here
```

Bubble Sort Using Template Functions

```
template < class X >
void Swap ( X &a, X &b )
 {
        X Tmp = a;
        a = b;
        b = Tmp;
  }
```

Bubble Sort Using Template Functions

```
int main ( )
{
 int x [ 5 ] = { 10, 50, 30, 40, 20 };
 float y [ 5 ] = { 1.1, 5.5, 3.3, 4.4, 2.2 };
Bubble ( x, 5 );  // calls template function for int values
Bubble ( y, 5 ); // calls template function for float values

cout << "Sorted x- Array: \n";
    for ( int i = 0; i < 5; i ++ )
            cout << x [ i ] << "   ";
                  cout << endl;
```

Bubble Sort Using Template Functions

```
cout << "Sorted y- Array: \n";
    for ( int i = 0; i < 5; i ++ )
            cout << y [ i ] << "   ";
                cout << endl;
                    return 0;

        }        // main ( ) ends here
```

Bubble Sort Using Template Functions

Output:

Sorted *x*- Array 10 20 30 40 50

Sorted *y*- Array 1.1 2.2 3.3 4.4 5.5

Overloading of Template Functions

```
# include < iostream>
# include < string>
template < class T >
void Display ( T x )
{   cout << "Template Display () is invoked:"
            << x  << "\n";       }

void Display ( int x )
{   cout << "Explicit Display is invoked:"
            << x  << "\n";       }
```

Overloading of Template Functions

```
int main ( )
{
        Display ( 100 );
        Display ( 12.34 );
        Display ( 'C');
                return 0;
}
```

Overloading of Template Functions

Output:

Explicit Display () is invoked: 100

Template Display () is invoked: 12.34

Template Display () is invoked: C

Member Function Templates

```
template < class T >
class Vector
{
        T* v;
        int Size;
public:
Vector  ( int m );
Vector ( T* a );
T operator* ( Vector & y );
}; // class definition ends here.
```

Member Function Templates

```
// Member Function templates
template < class T >
Vector < T > :: Vector ( int m )
{
        v = new T [ Size = m ];
        for ( int i = 0; i < Size; i ++ )
                v [ i ] = 0;
}
```

Member Function Templates

```
// Member Function templates
template < class T >
Vector < T > :: Vector ( T* a )
{
        for ( int i = 0; i < Size; i ++ )
                v [ i ] = a [ i ];
}
```

Member Function Templates

```
// Member Function templates
template < class T >
T Vector < T > :: Vector ( Vector & y )
{
        for ( int i = 0; i < Size; i ++ )
                Sum += this->v [ i ] * y.v [ i ];
        return Sum;
}
```

Exception Handling

The most common types of bugs are *logic* errors and *syntactic* errors.

The *logic* errors occur due to poor understanding of the <u>problem</u> and <u>*solution procedure*</u>.

The *syntactic* errors arise due to poor understanding of the <u>language itself</u>.

Exception Handling

We often come across some peculiar problems other than *logic* or *syntax* errors.

They are known as exceptions. Exceptions are *run time* anomalies or unusual conditions that a program may encounter while executing.

Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out memory or disk space.

Exception Handling

When a program encounters an exceptional conditions, it is important that it is identified and dealt with effectively.

Basics of Exception Handling

Exceptions are of two kinds, synchronous exceptions and asynchronous exceptions.

Errors such as “out-of-range index” and “over-flow” belong to synchronous type exceptions.

The errors that are caused by beyond the control of the program (such as keyboard interrupts) are called asynchronous exceptions.

Basics of Exception Handling

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken.

The mechanism suggests a separate error handling code that performs the following tasks:

Basics of Exception Handling

1. Find the problem (Hit the exception).
2. Inform that an error has occurred (Throw the exception).
3. Receive the error information (Catch the exception).
4. Take corrective actions (Handle the exception).

Exception Handling Mechanism

C++ exception handling mechanism is basically built upon three keywords, namely, try, throw and catch.

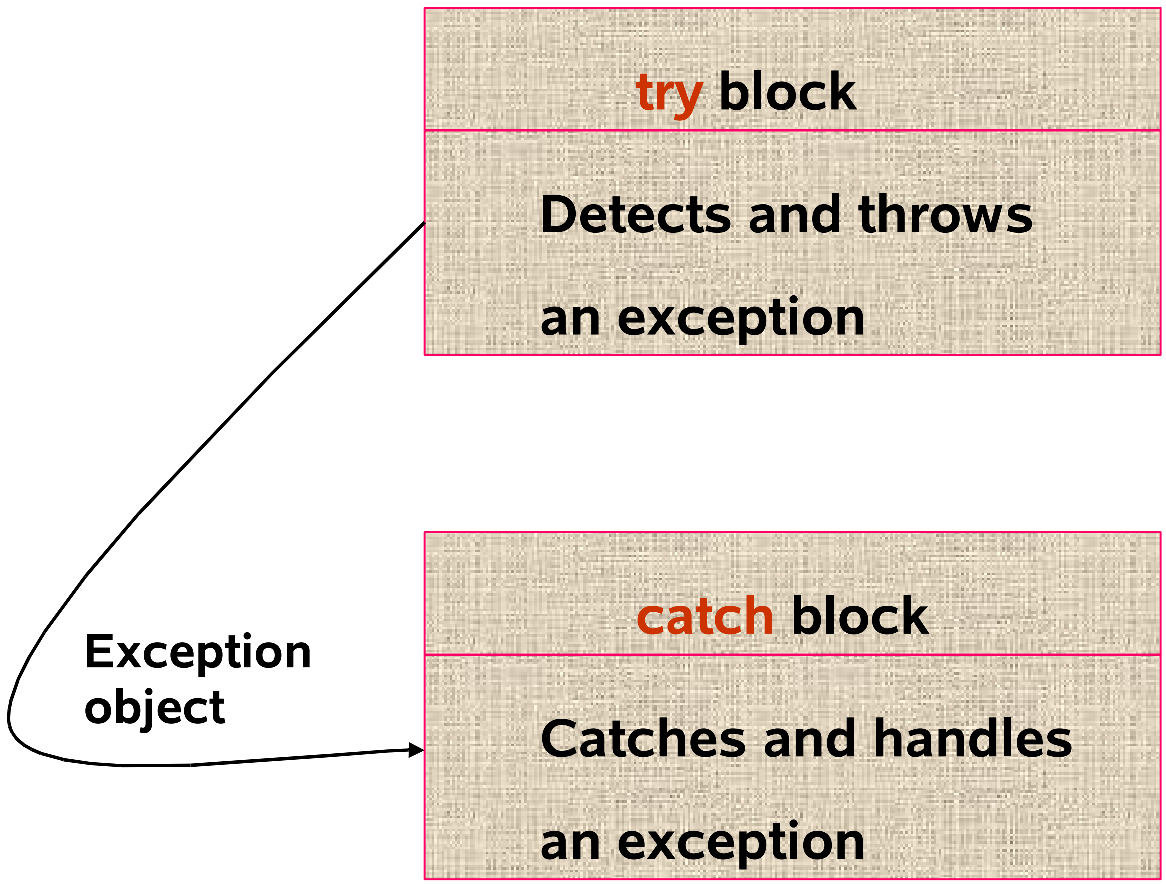
The keyword try is used to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements is known as try block.

Exception Handling Mechanism

When an exception is detected, it is thrown using a **throw** statement in the **try block**.

A **catch block** defined by the keyword catch **'catches'** the exception **'thrown'** by the **throw** statement in the **try block**, and handles it appropriately.

Exception Handling Mechanism



Try block throwing exception

Exception Handling Mechanism

```
#include < iostream >
int main ( )
        {
int a, b;
cout << “Enter Values of a and b \n”;
cin >> a;       cin >> b;
int x = a - b;
        try   {
        if ( x ! = 0 )
cout << “Result (a / x ) = ” << a / x << “\n”;
        else      // there is an exception
   throw ( x ); // throws int object
                } // try block ends here.
```

Exception Handling Mechanism

```
catch ( int i )
{
cout << "Exception caught: x  "<< x << "\n";
}
cout << "END";
return 0;
                    }   // main () ends here
```

Exception Handling Mechanism

First Run

Enter Values of a and b

20 15

Result (*a / x*) = 4

END

Second Run

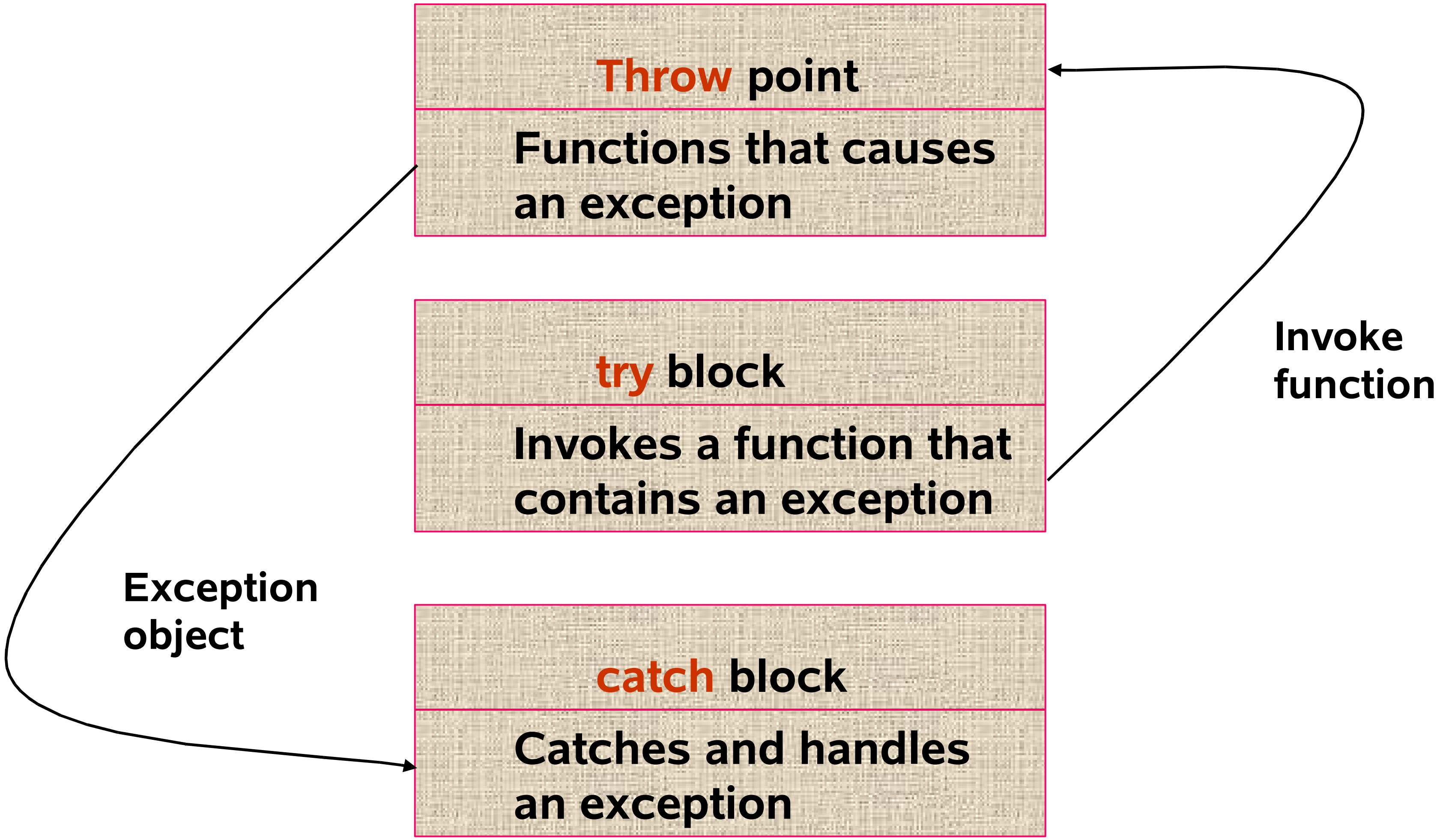
Enter Values of a and b

20 20

Exception caught: x = 0

END

Exception Handling Mechanism



Function invoked by try block throwing exception

Exception Handling Mechanism

```
# include < iostream >
void Divide ( int x, int y, int z )
{
    cout << "\n We are inside the function \n";
    if ( ( x – y ) != 0 )     // It is o.k.
    {
            int R = z / ( x – y );
            cout << "Result = " << R << "\n";
    }
    else          // There is a problem
            throw ( x – y );    // throw point
 }   // definition of Divide ( ) ends here
```

Exception Handling Mechanism

```
int main ()
{
try
{
cout << "\n we are in the try block \n ";
Divide ( 10, 20, 30 );     // invoke Divide ( )
Divide ( 10, 10, 20 ); // invoke Divide ( )
}
catch ( int i )        // catches the exception
cout << " Caught the exception \n";
return 0;
}
```

Exception Handling Mechanism

Output

We are inside the try block

We are inside the function

Result = -3

We are inside the function

Caught the exception

Multiple Catch Statements

```
#include < iostream >
void Test ( int x )
try {
if ( x = = 1 )      throw x;              // int
else
if ( x = = 0 )      throw x;              // char
else
if ( x = = -1 )     throw x;              // double
cout << "End of try-block \n";
      }
```

Multiple Catch Statements

```
catch ( char c )                    // catch 1
{
cout << "Caught a character \n";
}

catch ( int m )                     // catch 2
{
cout << "Caught a integer \n";
}

catch ( double d )                  // catch 3
{
cout << "Caught a double \n";
}
cout << "End of try-catch system \n\n";
        }        // definition of test ( ) ends here.
```

Multiple Catch Statements

```
int main ( )
{
cout << "Testing Multiple Catches \n";
cout << "x == 1 \n";
Test ( 1 );
cout << "x == 0 \n";
Test ( 0 );
cout << "x == -1 \n";
Test ( -1 );
cout << "x == 2 \n";
Test ( 2 );
                    return 0;
}          // main ( ) ends here.
```

Multiple Catch Statements

Output:

Testing Multiple Catches

x == 1

Caught an integer

End of try-catch system

x == 0

Caught an character

End of try-catch system

x == -1

Caught an double

End of try-catch system

Multiple Catch Statements

Output:

x == 2

End of try-block

End of try-catch system

Catch All Exceptions

```
# include < iostream >
void Test ( int x )  {
        try
        {
if ( x == 0 )       throw x;     // int
if ( x == -1 )      throw 'x';   // char
if ( x == 1 )       throw 1.0;   // float
         }  // try block end here
catch ( ..... )     // catch all
 {
cout << "Caught Generic catch \n";
 }
 }  // definition of Test ( ) ends here.
```

Catch All Exceptions

```
int main ( )
{
cout << "Testing Generic Catch";
Test ( -1 );
Test ( 0 );
Test ( 1 );
        return 0;
 } // definition of main ( ) ends here
```

Catch All Exceptions

Output:

Testing Generic Catch

Caught an exception

Caught an exception

Caught an exception

Re-throwing an Exception

```
# include < iostream >
void Divide ( double x, double y )
{
cout << “Inside function  \n”;
try
{
if ( y = 0.0 )
throw y;          // throwing double
else
cout << “Division = ” << x / y << “\n”;
}
```

Re-throwing an Exception

```
        catch ( double )
        {
cout << "Caught double inside function \n";
throw;
        }
        }    // definition of Divide ( ) ends here
```

Re-throwing an Exception

```
int main ( )
{
cout << “Inside main \n”;
try
{
Divide ( 10.5, 2.0 );
Divide ( 20.5, 0.0 );
}
catch ( double )
{
cout << “Caught double inside main \n”;
 }
            return 0;
} // main () ends here
```

Re-throwing an Exception

Output:

```
Inside main
Inside function
Division = 5.25
End of function

Inside function
Caught double inside function
Caught double inside main
End of main
```

Specifying Exception

Output:

Inside main

Inside function

Division = 5.25

End of function

Inside function

Caught double inside function

Caught double inside main

End of main

Specifying Exception

```
#include < iostream >
void Test ( int x )    throw ( int, double )
try {
if ( x = = 0 )      throw x;                    // int
else
if ( x = = 0 )      throw x;                    // char
else
if ( x = = -1 )     throw x;                    // double
cout << “End of try-block \n”;
      }
```

Specifying Exception

```
int main ( )
{
try {
cout << "Testing Throw Restrictions Catches \n";
cout << "x == 0 \n";
Test ( 0 );
cout << "x == 1 \n";
Test ( 1 );
cout << "x == -1 \n";
Test ( -1 );
cout << "x == 2 \n";
Test ( 2 );
}  // try block ends here
```

Specifying Exception

```
catch ( char c )
{
cout << “Caught a character \n”;
}

catch ( int m )
{
cout << “Caught a integer \n”;
}

catch ( double d )
{
cout << “Caught a double \n”;
}
cout << “End of try-catch system \n\n”;
            }           // definition of test ( ) ends here.
                        return 0;
                             }     // main ( ) ends here.
```

Specifying Exception

Output:

Testing Throw restrictions

x **== 0**

Caught a character

End of try-catch system