PROGRAMMING LANGUAGES

It is divided into four categories:

MONOLITHIC PROGRAMMING

- The program size is lengthy.
- It consists of global data and the code is sequential.
- The code is duplicated each time.
- The flow of control achieves through jump.
- Example: **Assembly Language** and **Basic**.

PROCEDURAL PROGRAMMING

- The program consists of sub-routines.
- Data items are global.
- Program controls through jump.
- Repetition of code can be avoided by using sub-routines.
- Suitable for medium sized software applications.
- Difficult to maintain and reuse of the program code.
- Example: **FORTRAN** and **COBOL.**

STRUCTURED PROGRAMMING

- Program can be divided into individual procedures that perform individual task.
- Procedures are independent and have own declaration and processing logic.
- Parameter passing is possible.
- Control of scope of data.
- Declaration of user-defined data type.
- Projects can be broken into small modules.
- Maintenance of large software system is tedious and costly.
- Example: **C** and **Pascal.**

OBJECT ORIENTED PROGRAMMING

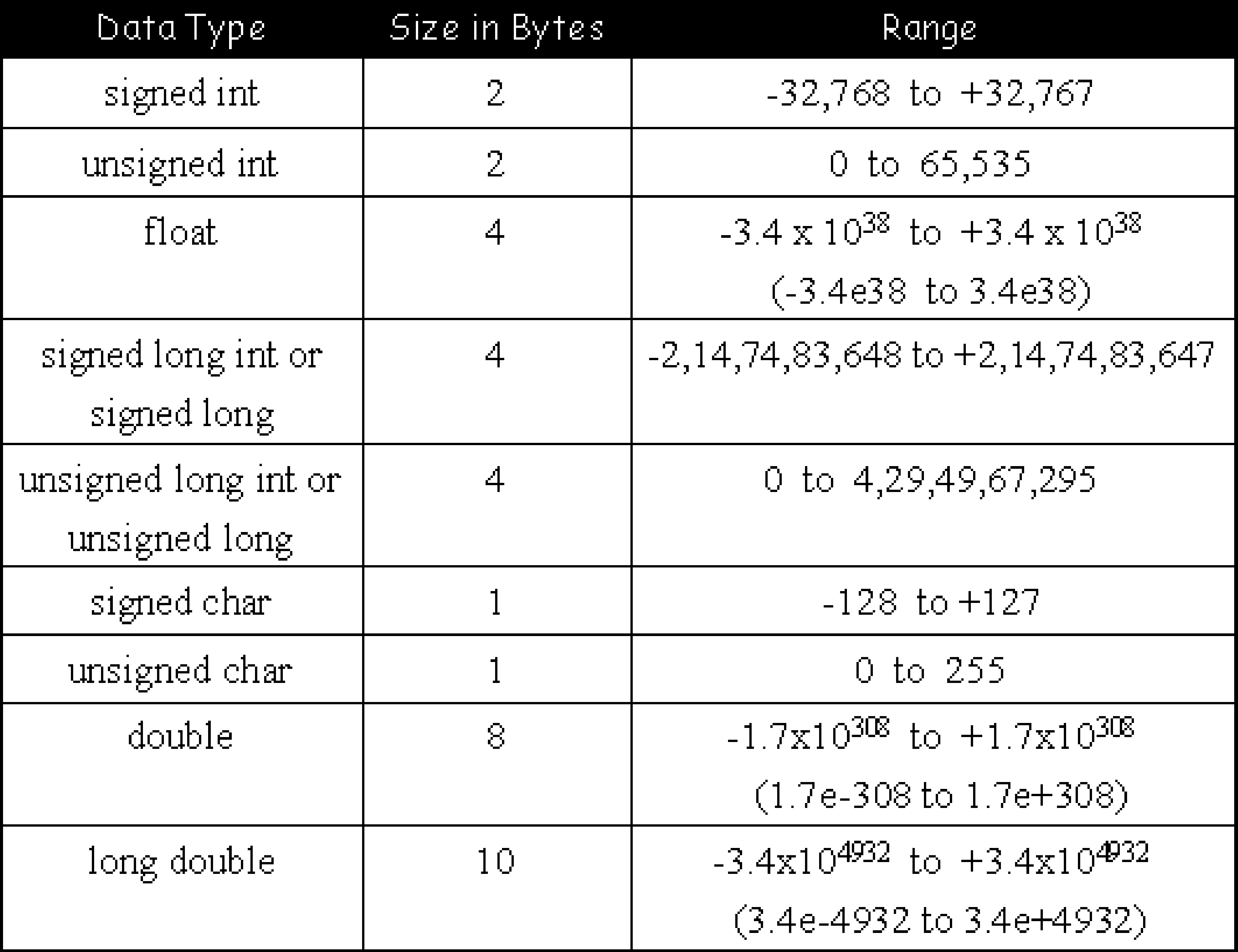
- Data abstraction (new data type creation) is introduced.
- Data and its operations are united together into a single unit.
- Programs are designed around data being operated.
- Relationship can be created between similar data type.
- Example**: C++, *Java*, *Smalltalk*, *Eiffel*, *Sather.*(pure object oriented language)**

INTRODUCTION TO C++

Developed by **Bjarne Stroustrup** in **1979** at Bell Laboratories in Murray Hill, New Jersy.

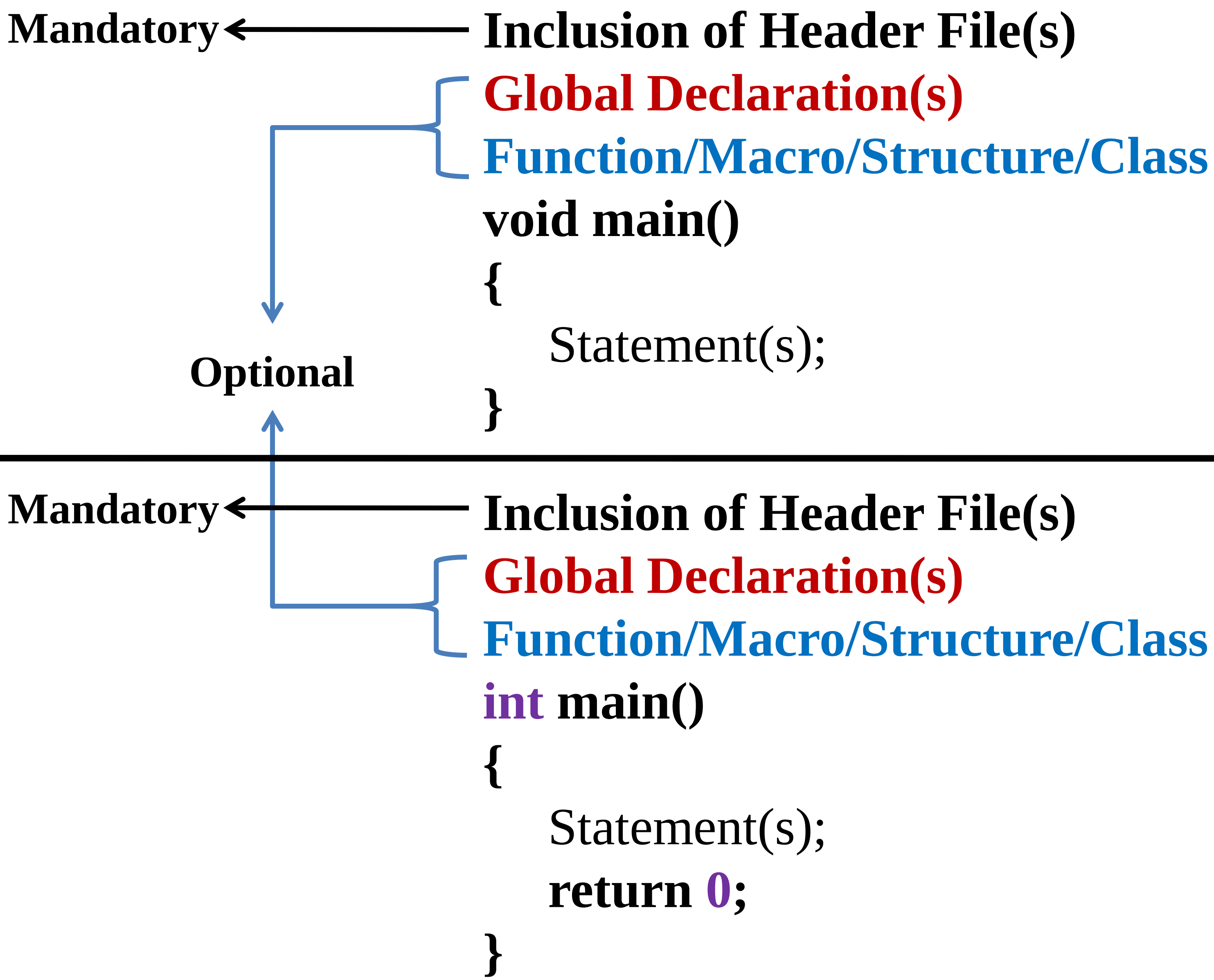
Initially it was referred as **C-Classes**.

In **1984**, the name was changed to **C++** because it is the extension of C-Language.



Data Type	Size in Bytes	Range
signed int	2	-32,768 to +32,767
unsigned int	2	0 to 65,535
float	4	-3.4×10^{38} to $+3.4 \times 10^{38}$ (-3.4e38 to 3.4e38)
signed long int or signed long	4	-2,14,74,83,648 to +2,14,74,83,647
unsigned long int or unsigned long	4	0 to 4,29,49,67,295
signed char	1	-128 to +127
unsigned char	1	0 to 255
double	8	-1.7×10^{308} to $+1.7 \times 10^{308}$ (1.7e-308 to 1.7e+308)
long double	10	-3.4×10^{4932} to $+3.4 \times 10^{4932}$ (3.4e-4932 to 3.4e+4932)

General Structure of C++ Program



Mandatory

Optional

```
Inclusion of Header File(s)
Global Declaration(s)
Function/Macro/Structure/Class
void main()
{
    Statement(s);
}
```

Mandatory

```
Inclusion of Header File(s)
Global Declaration(s)
Function/Macro/Structure/Class
int main()
{
    Statement(s);
    return 0;
}
```

Statements in C++

- Declaration statements
- Input statements
- Assignment statements
- Output statements

KEYWORDS

asm	const	else	friend	namespace	return	template	union
auto	const_cast	enum	goto	new	short	this	unsigned
bool	continue	explicit	if	operator	signed	throw	using
break	default	export	inline	private	sizeof	true	virtual
case	delete	extern	int	protected	static	try	void
catch	do	false	long	public	static_cast	typedef	volatile
char	double	float	mutable	register	struct	typeid	wchar_t
class	dynamic_cast	for		reinterpret_cast	switch	typename	while

Comments in C++

/*……..*/ and **//** are used for commenting a statement (s) in C++.

Examples

/* This is a C+ + program Comment. */

// This is a C + + program Comment.

Literals

In C+ +, we can define a constant or literal in three different ways:

- **const** keyword
- **#define**
- **enum** (enumerated data type)

Examples:

i. Using const keyword

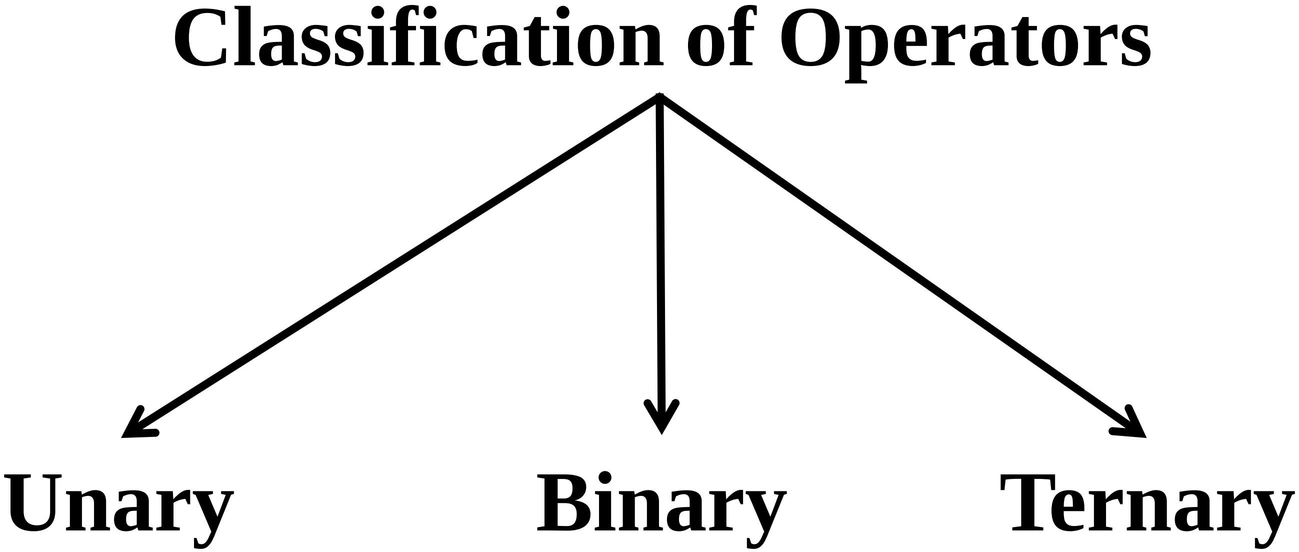
```
   const int a = 66;
   const char *p = “IGIT C++”;  (or) const char p[] = “IGIT C++”;
   const ARS = 3147;  // It considers by default int data type
```

ii. Using #define

```
     #define SevenStars “* * * * * * *\n”
     #define PI 3.142
```

iii. Using enum

```
      enum days
      {
        Sunday, Monday, Tuesday, Wednesday,
        Thursday, Friday, Saturday
       };
       enum Color {Red, Green, Blue, Black, White, Yellow};
```



Classification of Operators

Unary

Binary

Ternary

OPERATORS

1) Assignment (=)
2) Arithmetic (+, -, *, /, %)
3) Unary (++, --)
4) Relational (==, >, <, >=, <=, !=)
5) Logical (||, &&, !)
6) Ternary or Conditional (? :)
7) Scope Resolution (::)
8) Sub-Script ([ ])
9) Address (&)
10) Indirection (*)
11) Dot (.)
12) Insertion or put to (<<)
13) Extraction or Get (>>)
14) Right Arrow (->)

STREAMS

It is a medium through which an output and input operation takes place.

It is of two types: Output stream **(cout)** and Input stream **(cin)**.

These streams are belonged to the header file **<iostream.h>**

OUTPUT STREAM (cout)

It is used for output operation.

Syntax:

cout<<”Any Prompt Message”/Variable(s)/Expression(s);

Example	Output
1) cout<<”My College Name is IGIT”;	My College Name is IGIT
int a=31, b = 47;	
2) cout<<a<<b<<”\n”;	3147
3) cout<<a+b<<”\n”;	78
4) cout<<”a+b=”<<a+b<<”\n”;	a+b=78
5) cout<<a<<”+”<<b<<”=”<<a+b;	31+47=78

Note: The above **output statements (2-5)** are known as **cascaded output operation** since more than one expression is used with **one cout** stream.

INPUT STREAM (cin)

It is used for input operation.

Syntax: cin>>Variable(s);

→ Extraction or Get Operator

Example:

int a, b, c;

cin>>a;

cin>>b;

cin>>c;

The above three statements allow you to inputting three unknown integers.

Or

cin>>a>>b>>c;

The above statement allows you to inputting three unknown integers and it is known as ***cascaded input*** operation because more than one variable used with one ***cin*** stream.

EXAMPLE

```
// Read any three numbers. Find its sum and average.
#include<iostream.h>
void main()
{
   float a, b, c, sum, avg;

cout<<“Enter 1st Number: “;
cin>>a;
cout<<“Enter 2nd Number: “;
cin>>b;
cout<<“Enter 3rd Number: “;
cin>>c;

sum = a+b+c;
avg = sum/3;

cout<<“Sum = “<<sum<<endl;
cout<<“Average = “<<avg;
}
```

EXAMPLE

```
// Read any three numbers. Find its sum and average.
#include<iostream.h>
void main()
{
   float a, b, c, sum, avg;

cout<<“Enter Three Numbers: “;
cin>>a>>b>>c;

sum = a+b+c;
avg = sum/3;

cout<<“Sum = “<<sum<<endl;
cout<<“Average = “<<avg;
}
```

C++ Structure

It combines logically related data items of different data type as well as functions into a single unit.

The data items enclosed within a structure are known as members.

Syntax of Structure Declaration

```
struct structureName
{
DataType1 var1, var2,……;
DataType2 var1, var2,……;
..................................;
..................................;
Function(s) definition;
};
```

Example:

```
struct student
{
char n [20];
int roll;
char branch [20];
int marks;
void assign (char p[], int q, char r[], int s)
{
strcpy (n, p);
roll = q;
strcpy (branch, r);
marks = s;
}
};
```

The size of the above structure, **student** is 20 +2 + 20 + 2 = 44 bytes.

Structure Definition

The structure Definition creates structure variable and allocates storage space for them.

Structure variable can be created at the time of structure declaration or by using the structure name as and when required.

Syntax

```
1) struct structureName
   {
      // variable(s) and function(s);
    } structVar1, structVar2,........;

2) struct  structureName structVar1, structVar2,.......;

3) structureName structVar1, structVar2,.........;

4)  struct
    {
      // variable(s) and function(s);
     } structVar1, structVar2,.......        ;
```

EXAMPLES

```
1)  struct student
    { int r;
       char n[10];
       int mark;
      void Assign (int a, char *p, int b)
      {
         r = a;     strcpy (n, p);     mark = b;
      }
      void Show ( )
      {
       cout<<n<<” “<<r<<” “<<mark<<endl;
      }
    } S;
2) struct student S;
3)  student S;
```

```
4) struct
   {
     int r;
     char n[10];
     int mark;
     void Assign (int a, char *p, int b)
     {
       r = a;
       strcpy(n, p);
       mark = b;
     }
     void Show ( )
     {
       cout<<n<<" "<<r<<" "<<mark<<endl;
      }
      } S;
```

EXAMPLE-1

```
// Read  a name of the student, roll
// number, branch, and total marks
// using structure.
#include<iostream.h>
struct Student
{
   char name[30], branch[20];
   int roll, tmarks;
 };
void main()
{
   Student srec;

cout<<“Enter Name of the Student: “;
cin>>srec.name;
cout<<“Enter Roll Number: “;
cin>>srec.roll;
cout<<“Enter Branch: “;
cin>>srec.branch;
```

```
cout<<“Enter the Total Marks: “;
cin>>srec.tmark;

cout<<“Student Name = “<<srec.name;
cout<<“\nRoll Number = “<<srec.roll;
cout<<“\nBranch = “<<srec.branch;
cout<<“\nTotal Marks = “<<srec.tmark;
}
```

EXAMPLE-2

```
// Read ‘n’ students name, roll
// number, branch, and total marks
// using array of structure.
#include<iostream.h>
struct Student
{
  char name[30], branch[20];
  int roll, tmarks;
 };
void main()
{
  Student srec[100];
  int n;
cout<<“How Many Student Details? “;
cin>>n;
for(int i=0; i<n; i++)
{
cout<<“Enter Name of the Student: “;
cin>>srec[i].name;
```

```
cout<<“Enter Roll Number: “;
cin>>srec[i].roll;
cout<<“Enter Branch: “;
cin>>srec[i].branch;
cout<<“Enter the Total Marks: “;
cin>>srec[i].tmark;
}

cout<<n<<“ Student Details\n”;
for(i=0; i<n; i++)
{
cout<<“\nName = “<<srec[i].name;
cout<<“\nRoll Number = “<<srec[i].roll;
cout<<“\nBranch = “<<srec[i].branch;
cout<<“\nTotal Marks = “<<srec[i].tmark;
}
}
```

EXAMPLE-3

```
// Read  ‘n’  students name, roll
// number, branch, and total marks
// using array of structure.
#include<iostream.h>
struct Student
{
   char name[30], branch[20];
   int roll, tmarks;
   void Read()
   {
 cout<<“Enter Name of the Student: “;
cin>>name;
cout<<“Enter Roll Number: “;
cin>>roll;
cout<<“Enter Branch: “;
cin>>branch;
cout<<“Enter the Total Marks: “;
cin>>tmark;
}
```

```
void Show()
{
cout<<“\nName = “<<name;
cout<<“\nRoll Number = “<<roll;
cout<<“\nBranch = “<<branch;
cout<<“\nTotal Marks = “<<tmark;
}
};
void main()
{
   Student srec[100];
   int n;
cout<<“How Many Student Details? “;
cin>>n;
for(int i=0; i<n; i++)
    srec[i].Read();
cout<<n<<“ Student Details\n”;
for(i=0; i<n; i++)
    srec[i].Show();
}
```

Reading a line of text

1. getline ( ):

It allows inputting any line of text up to the enter key is pressed **with or without spaces**.

Syntax

cin.getline (string_variable, width/size);

Example:

```
char a [20];
cin.getline(a, 20);
```

2. get ( ):

It allows inputting any **string / line of text / a paragraph** in a string variable.

(i) Syntax for Reading a string or line of text:

cin.get (string_variable, width/size);

Example:

```
char a [20];
cin.get(a, 20);
```

(ii) Syntax for Reading a paragraph:

```
cin.get (string_var, width, control_character);
```

Example:

```
char a [20];
cin.get(a, 20, '*');
```

Manipulators

i) endl [end line]:

To bring the cursor to the next line or it places the cursor in the new line.

Syntax:

```
                    cout<<endl;
```

ii) setw ( ) [set width]:

It assigns a column width for displaying the variable or constant or expression.

It belongs to the header file **<iomanip.h>**.

It is a right aligned.

Syntax:

```
cout<<setw (int value);
```

iii) setiosflags ( ) and setprecision ( ):

To control decimal point and precision in a float data.

Syntax:

cout<<**setiosflags (ios::showpoint)** <<**setprecision (no_of_decimal_places)**;

Example:

cout<<setiosflags (ios::showpoint) <<setprecision (4) <<7;

Output:

```
          7.0000
```

Ternary / Conditional Operator (?:)

- It replaces if..else statement.
- It is a single line statement.
- It performs at a time one statement.
- It can also perform compound statement; such statements are included in parentheses.
- It can return a value.

Syntax of Ternary Operator (? :)

No Return Value and Single Statement

(Expression) ? Statement-1 : Statement-2;

No Return Value and Compound statement

**Expression? (Statement-1, Statement-2, ...) :
(Statement-1, Statement-2, ...);**

Return Value

variableName = Expression ? Statement-1 : Statement-2;

Syntax of Nested Ternary Operator (? : ? : .... :)

No Return Value and Single Statement

(Expression-1) ? Statement-1 : (Expression-2)?
Statement-2 : .... : Statement-n;

No Return Value and Compound statement

Expression-1? (Statement-1, Statement-2, ...) :
Expression-2 ? (Statement-1, Statement-2, ...) : .... :
(Statement-1, Statement-2, ...);

Return Value

variableName = Expression-1 ? Statement-1 :
Expression-2 ? Statement-2 : ....: Statement-n;

Types of variables

1)Value variable:

It holds a standard value.

Syntax: DataType variable1, variable2………..;

Examples:

```
int a=9;
float b =67.76;
char c = '#';
```

2) Reference variable or Variable Aliases:

It refers to a value variable with another name.

Syntax:

DataType &referenceVariable = valueVariable;

Examples:

```
char ch, &chi = ch; // chi is an alias of char ch

int b, &a =b;          // a is an alias of int b

float y, &x =y;        // x is an alias of float y float y, &x;

x = y;                 // invalid
```

3) Pointer Variable:

It holds address of value variable or pointer variable of same data type.

Syntax: **DataType *pointerVariable;**

Here ***** is known as **referencing** or **indirection operator**.

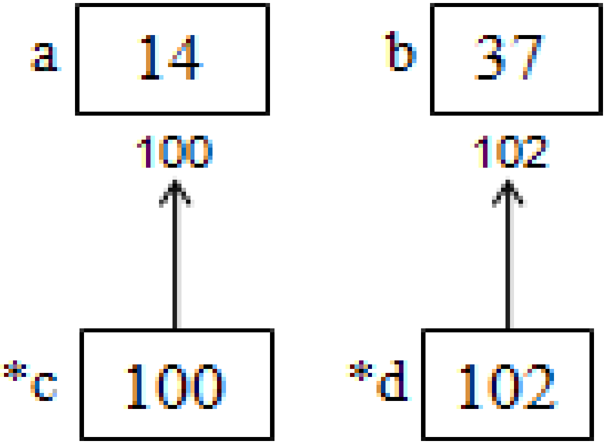
Examples: char *a;
int *b;
float *ab;

Here pointer variables a, b, and ab can point to **char**, **int**, and **float** memory blocks respectively.

```
int a = 14, b = 37, *c = &a, *d = &b;
```

OR

```
int a = 14, b = 37, *c, *d;
c = &a;
d = &b;
```



```
cout<<”Direct Access”<<endl;
cout<<”A=”<<a<<”B=”<<b<<endl;
cout<<”Sum=”<<(a+b)<<endl;
```

```
cout<<”Indirect Access”<<endl;
cout<<”A=”<<*c<<”B=”<<*d<<endl;
cout<<”Sum=”<<(*c+*d)<<endl;
```

Types of Pointers

1) **General pointer**
2) **void pointer**
3) **this pointer**

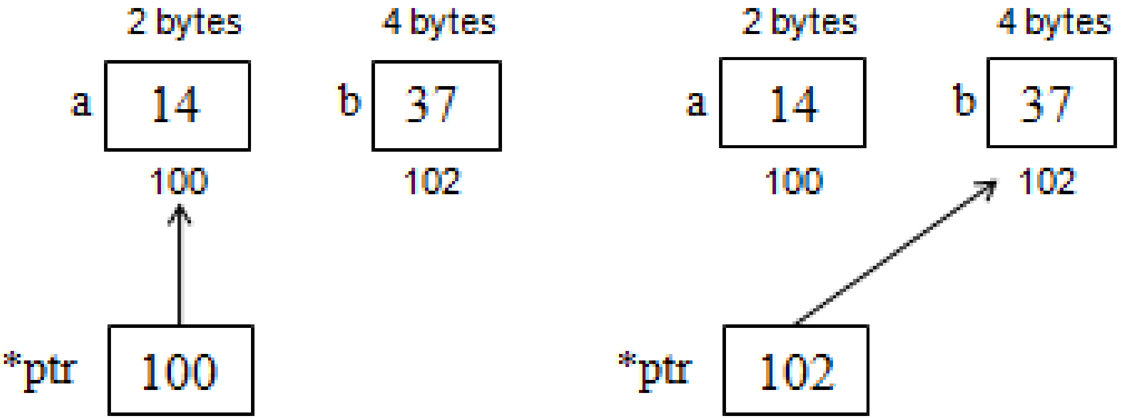
void pointer:

It can hold address of any kind of memory location or variable.

Syntax: **void *pointerName;**

Example:

```
void *ptr;  // The pointer ptr can point to any data type
            // location address.
```



```
void main()
{
int a=14;
float b=37;
void *ptr;

ptr = &a; // initialization of void pointer to int location address
cout<<”Value of a using void pointer *ptr=”;
cout<<*((int *)ptr)<<endl;

ptr = &b; // initialization of void pointer to float location address
cout<<”Value of b using void pointer *ptr = “;
cout<<*((float *)ptr)<<endl;
}
```

new operator:

To allocate memory during run time of a program and the allocated memory's address assigned to a pointer variable.

Syntax for one memory block:

```
dataType *pointerVariale = new dataType;
```

Examples:

```
int *ptr = new int;
float *a = new float;
char *cptr = new char;
```

Syntax for sequential memory blocks i.e. array of memory blocks:

```
dataType *pointerVariale = new dataType [size];
```

Examples:

```
int *ptr = new int[7];
float *qtr = new float[10];
```

delete operator:

To free the dynamically allocate memory block.

Syntax: **delete pointerVariable;**

Examples:

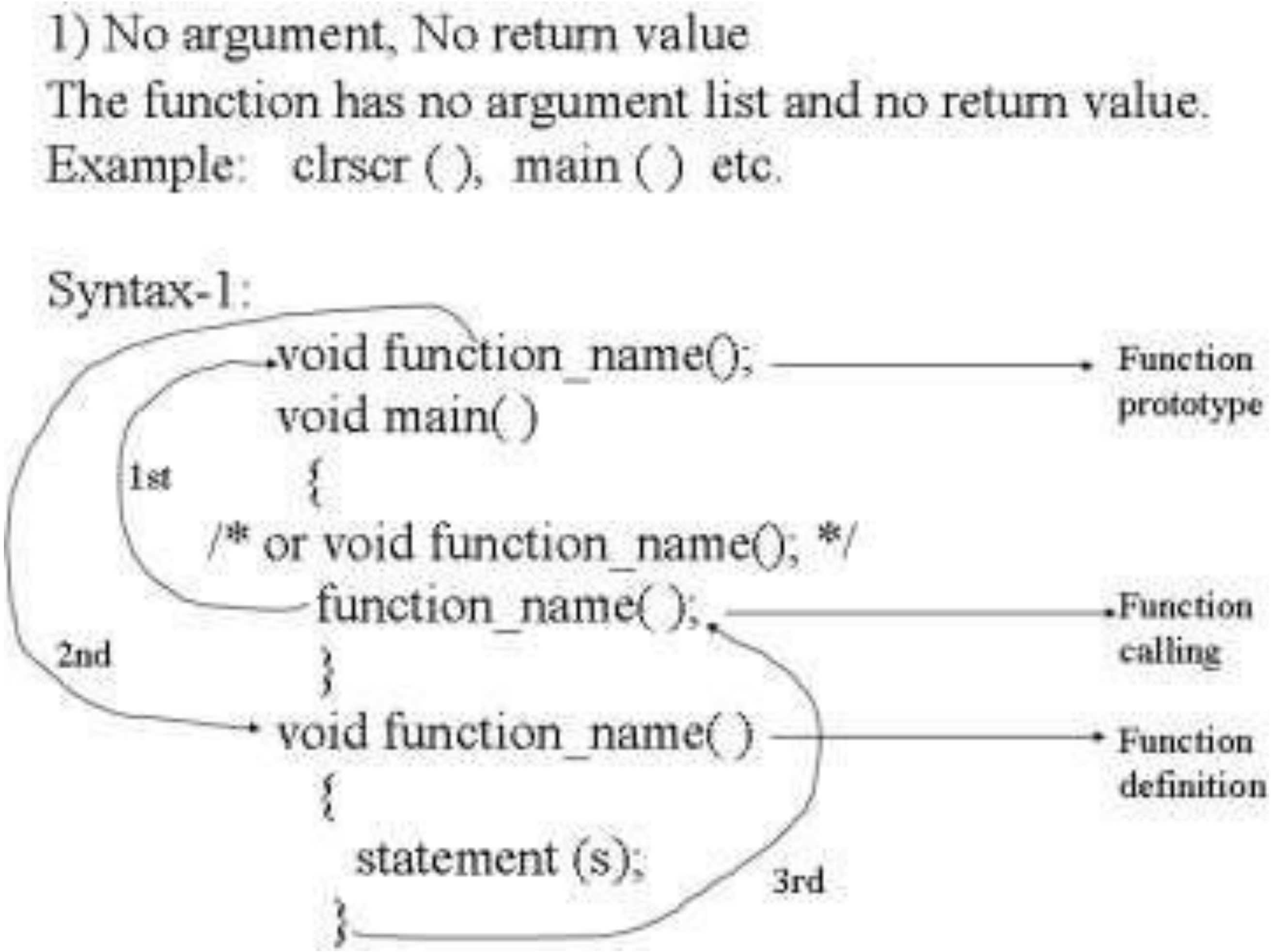
```
int *p = new int;  //Allocates one memory block of int kind.
int *q = new int [7];  //Allocates seven memory blocks of int kind.
*p = 66;
q[0] = 1;  or  *(q+0) = 1;
q[1] = 5;  or  *(q+1) = 5;

delete p, q;  // It releases one and seven memory blocks of int kind
              // held by pointers p and q.
```

User-Defined Functions

To define a statement (s) with a unique name for performing a particular task.

It is of four types.

1) No argument, No return value
2) No argument, return value
3) Argument, No return value
4) Argument, return value



1) No argument, No return value

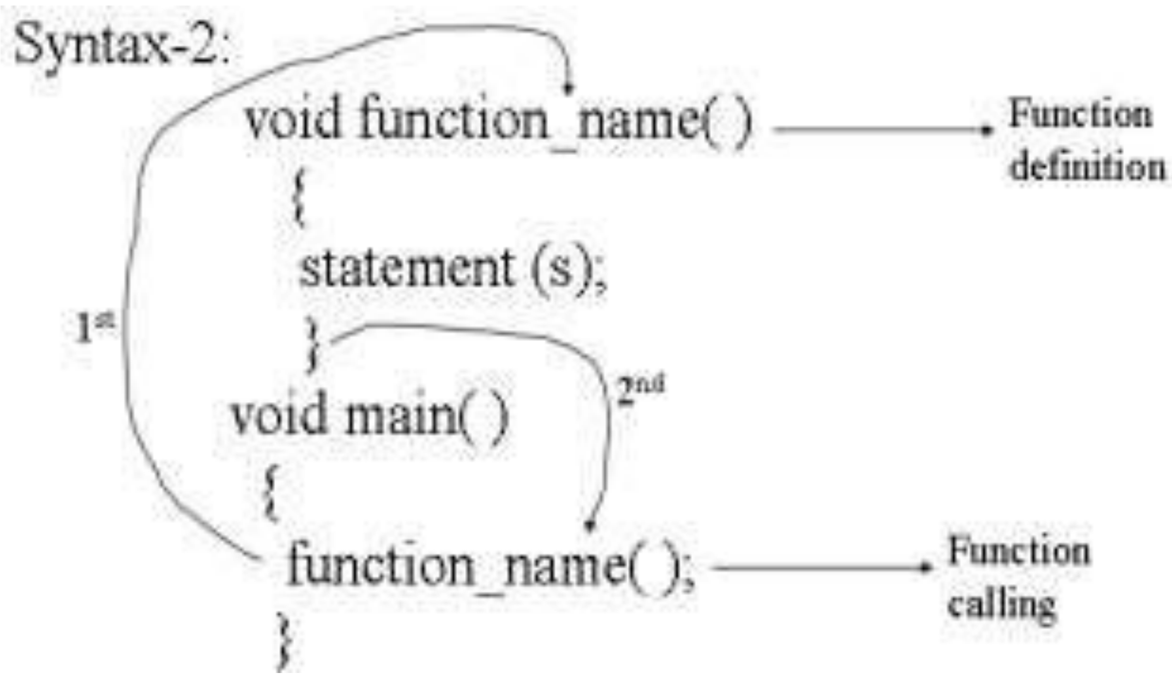
The function has no argument list and no return value.

Example: clrscr ( ), main ( ) etc.

Syntax-1:

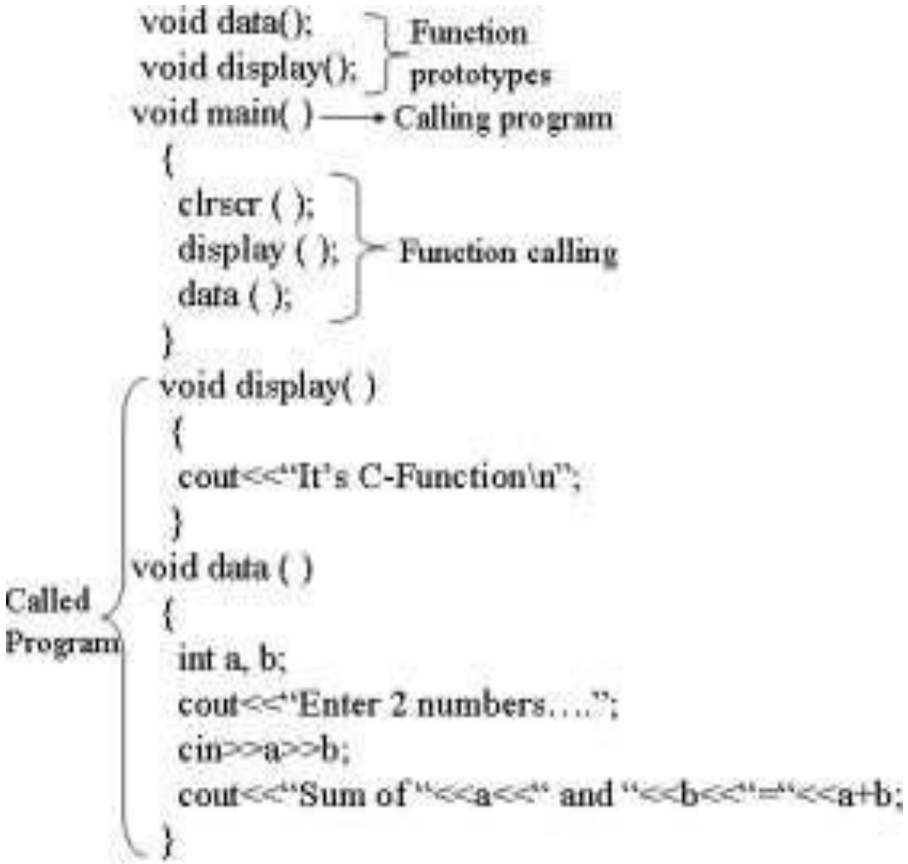
```
        void function_name();
        void main( )
          {
      /* or void function_name(); */
           function_name( );
          }
        void function_name( )
          {
            statement (s);
          }
```

Function Prototype: Declaration of function with argument list data type and return type in the declaration part of main as local or global way.



Syntax-2:

```
      void function_name( )
       {
        statement (s);
        }
     void main( )
      {
       function_name( );
      }
```



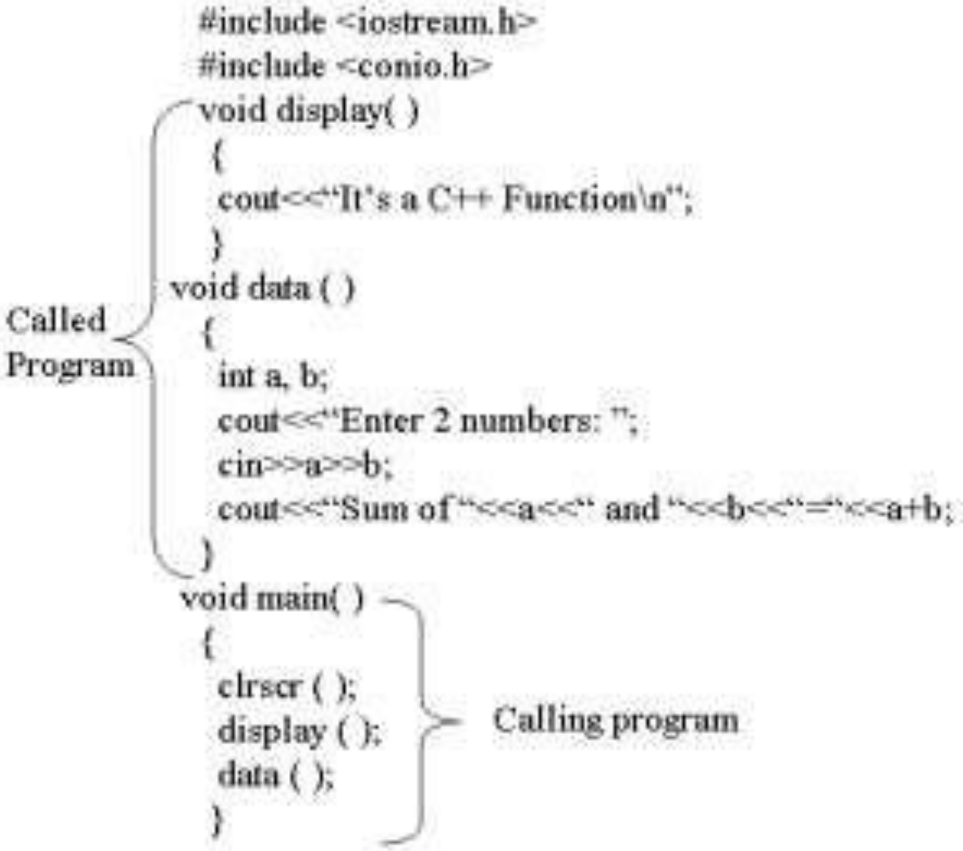
```
void data();
void display();
void main( )
 {
  clrscr ( );
  display ( );
  data ( );
 }
 void display( )
  {
  cout<<“It’s C-Function\n”;
  }
void data ( )
 {
  int a, b;
  cout<<“Enter 2 numbers….”;
  cin>>a>>b;
  cout<<“Sum of “<<a<<“ and “<<b<<“=“<<a+b;
 }
```

Function prototypes

Calling program

Function calling

Called Program



```
#include <iostream.h>
#include <conio.h>
void display( )
 {
  cout<<“It’s a C++ Function\n”;
 }
void data ( )
 {
  int a, b;
  cout<<“Enter 2 numbers: ”;
  cin>>a>>b;
  cout<<“Sum of “<<a<<“ and “<<b<<“=“<<a+b;
 }
void main( )
 {
  clrscr ( );
  display ( );
  data ( );
 }
```

Called Program

Calling program

2) No argument, return value

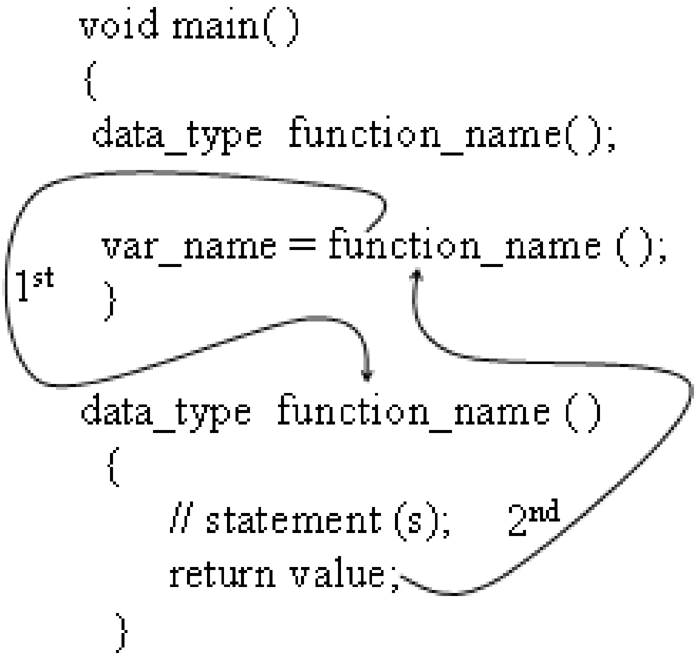
It has no argument but a return value. A function can return one value at a time with the help of return statement. The accepting and processing are done in the called program, where as printing is done in the calling program.

Syntax: return_type function_name ( );

Syntax of return: return (value);
└──→ constant / variable / expression

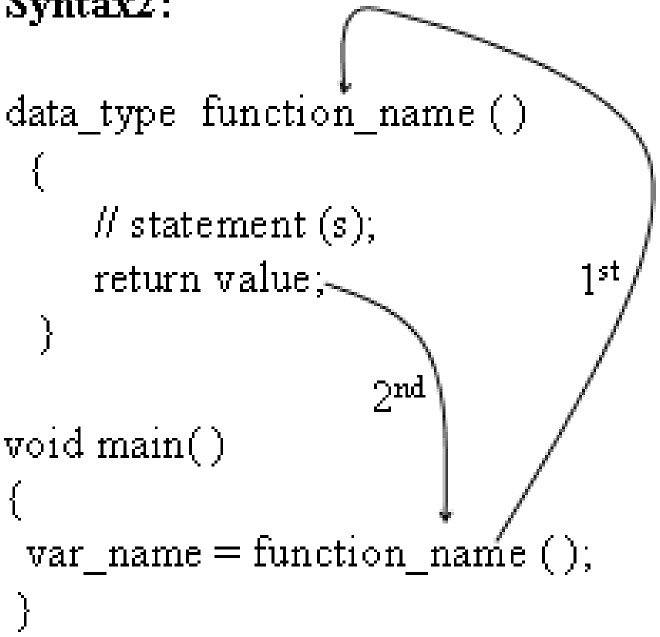
Example:
```
return (12);
return (a);
return (a+2);
return (‘d’);
return (13.7);
```

Syntax1:



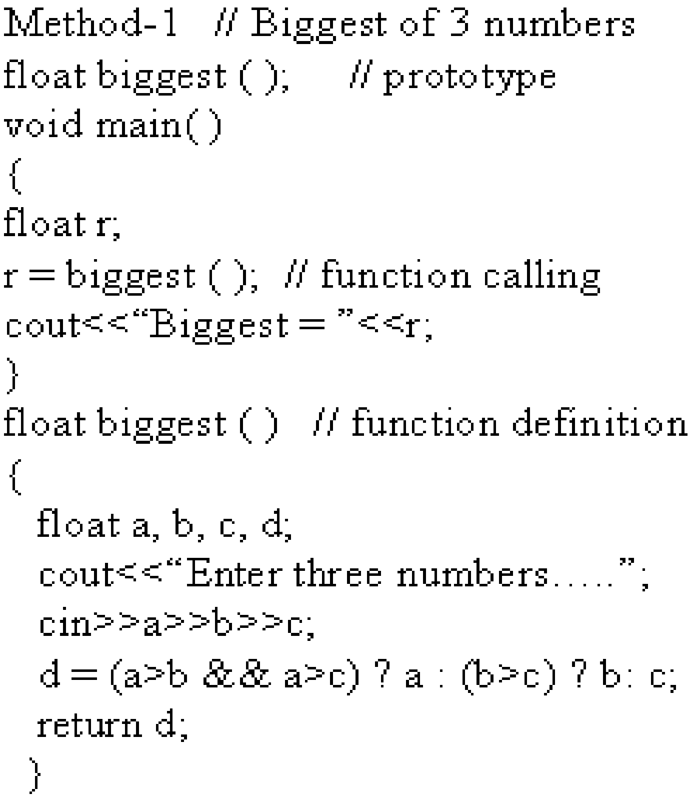
```
void main( )
{
 data_type  function_name( );

  var_name = function_name ( );
  }
data_type  function_name ( )
  {
       // statement (s);
       return value;
   }
```

Syntax2:

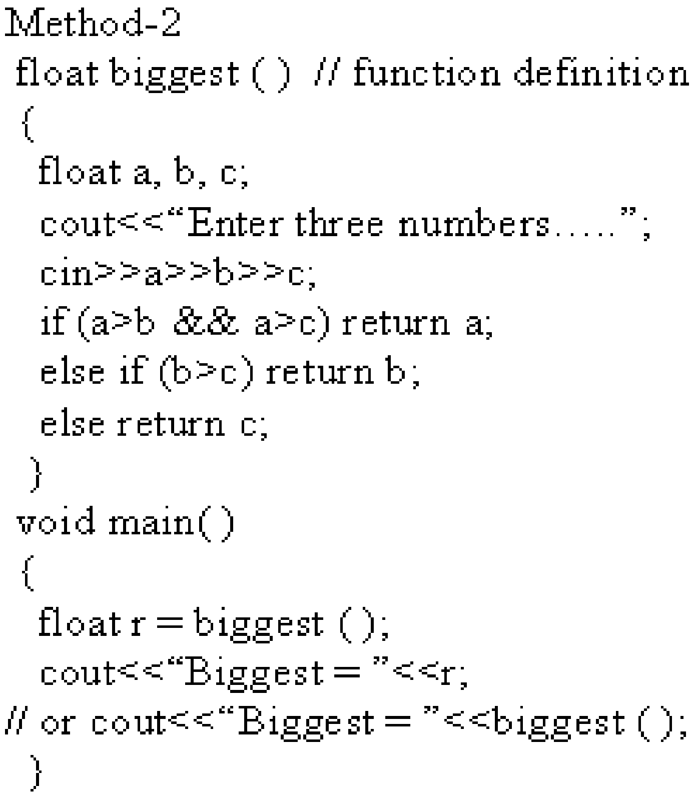


```
data_type  function_name ( )
  {
       // statement (s);
       return value;
  }

void main( )
{
 var_name = function_name ( );
}
```

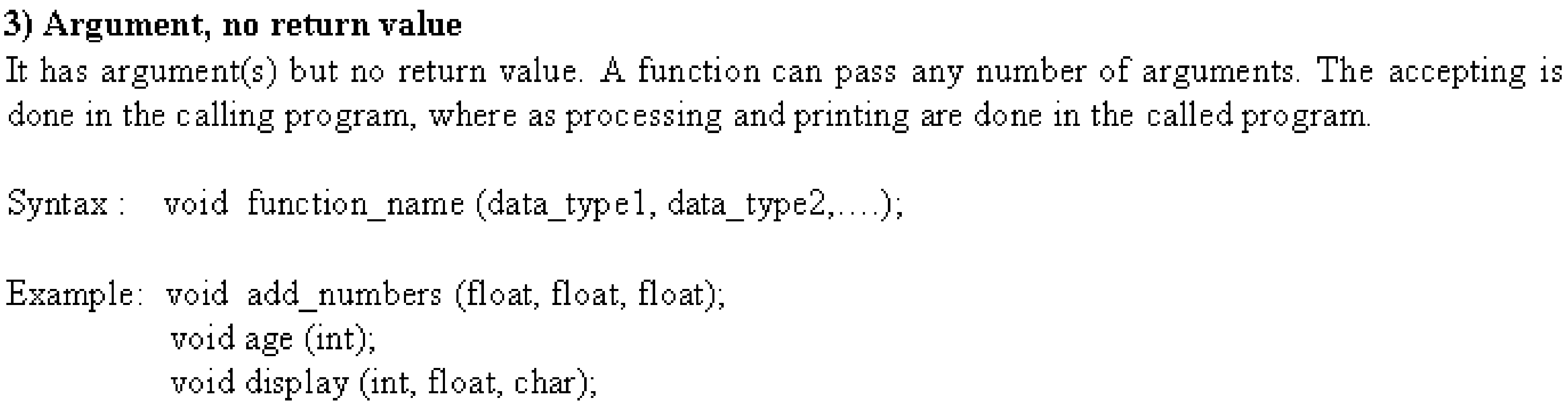
Example:



```
Method-1   // Biggest of 3 numbers
float biggest ( );     // prototype
void main( )
{
float r;
r = biggest ( );  // function calling
cout<<“Biggest = ”<<r;
}
float biggest ( )   // function definition
{
   float a, b, c, d;
   cout<<“Enter three numbers…..”;
   cin>>a>>b>>c;
   d = (a>b && a>c) ? a : (b>c) ? b: c;
   return d;
  }
```



```
Method-2
 float biggest ( )  // function definition
 {
   float a, b, c;
   cout<<“Enter three numbers…..”;
   cin>>a>>b>>c;
   if (a>b  && a>c) return a;
   else if (b>c) return b;
   else return c;
  }
 void main( )
 {
   float r = biggest ( );
   cout<<“Biggest = ”<<r;
// or cout<<“Biggest = ”<<biggest ( );
  }
```



3) Argument, no return value

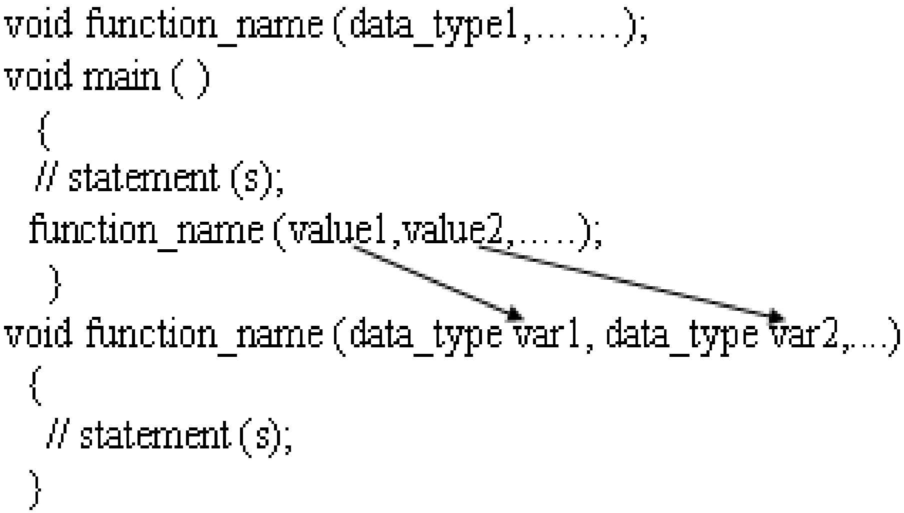
It has argument(s) but no return value. A function can pass any number of arguments. The accepting is done in the calling program, where as processing and printing are done in the called program.

```
Syntax :    void  function_name (data_type1, data_type2,....);

Example:  void  add_numbers (float, float, float);
           void age (int);
           void display (int, float, char);
```

Actual Argument: It is a constant or variable or expression to be passed through a function from the calling program.

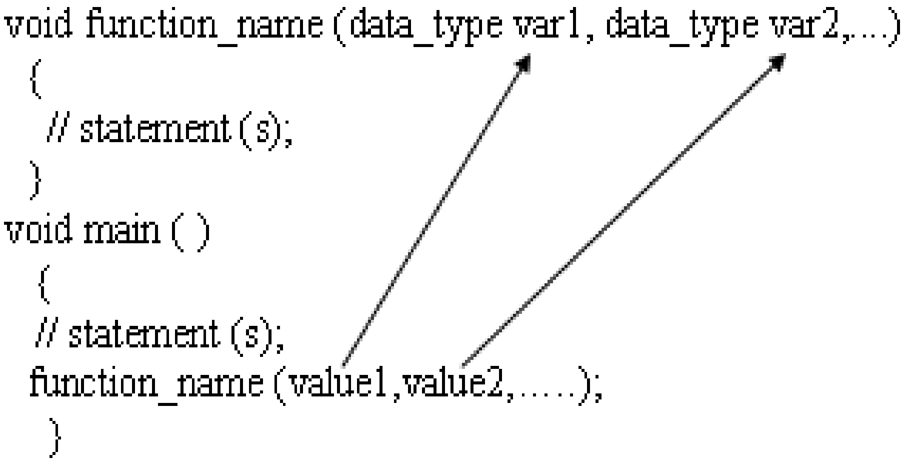
Formal Argument: The number of variables to be declared along with the function definition. The formal argument must match in terms of number and data type order.

Syntax1:



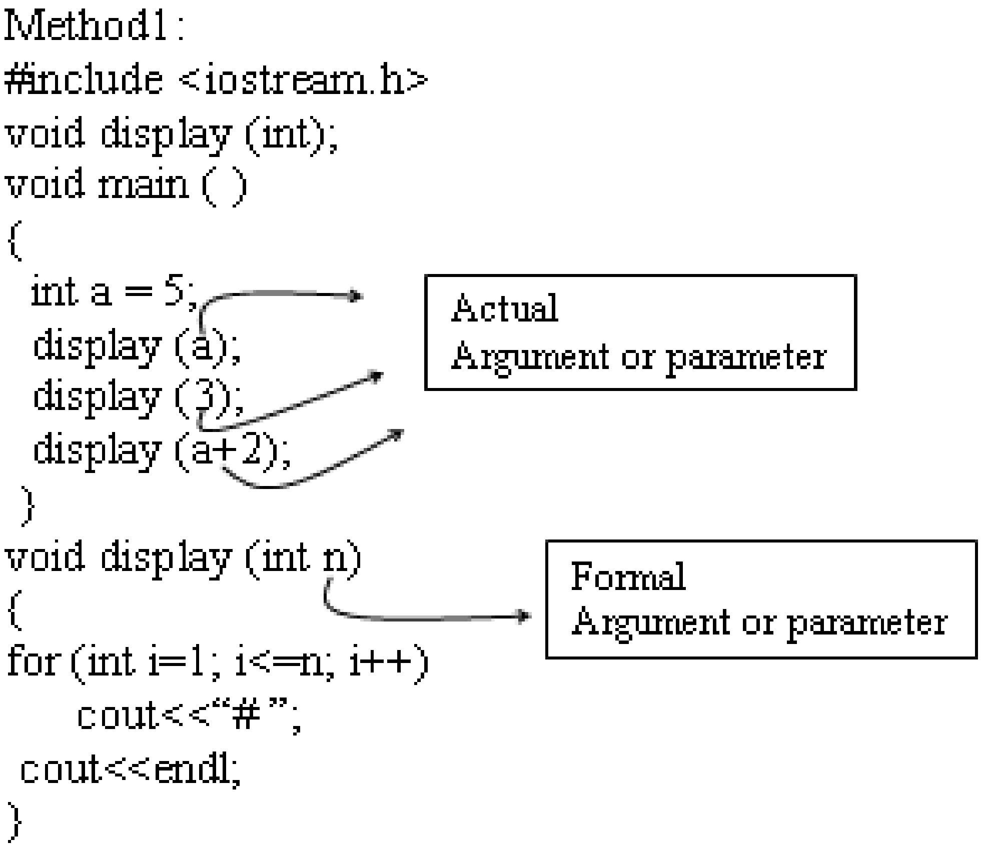
```
void function_name (data_type1,…….);
void main ( )
  {
 // statement (s);
 function_name (value1,value2,…..);
   }
void function_name (data_type var1, data_type var2,…)
 {
  // statement (s);
 }
```

Syntax2:



```
void function_name (data_type var1, data_type var2,…)
 {
  // statement (s);
 }
void main ( )
  {
  // statement (s);
 function_name (value1,value2,…..);
   }
```

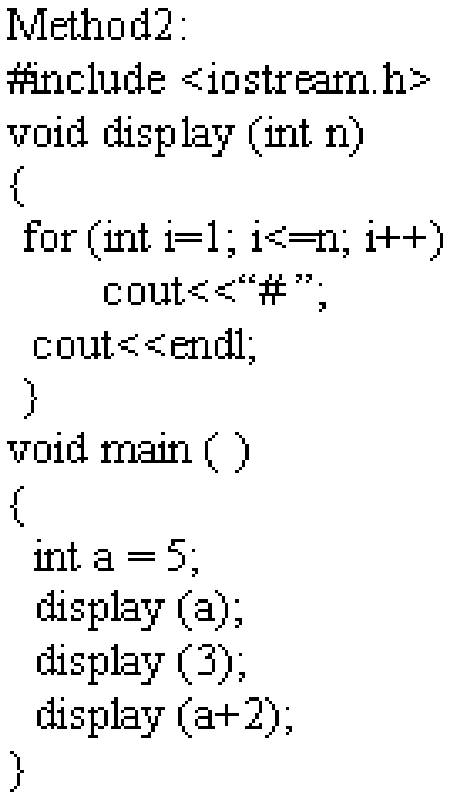
Example



```
Method1:
#include <iostream.h>
void display (int);
void main ( )
{
  int a = 5;
  display (a);
  display (3);
  display (a+2);
 }
void display (int n)
{
for (int i=1; i<=n; i++)
      cout<<"# ";
 cout<<endl;
}
```

Actual
Argument or parameter

Formal
Argument or parameter



```
Method2:
#include <iostream.h>
void display (int n)
{
 for (int i=1; i<=n; i++)
        cout<<"# ";
  cout<<endl;
 }
void main ( )
{
  int a = 5;
  display (a);
  display (3);
  display (a+2);
}
```

4) Argument, Return value

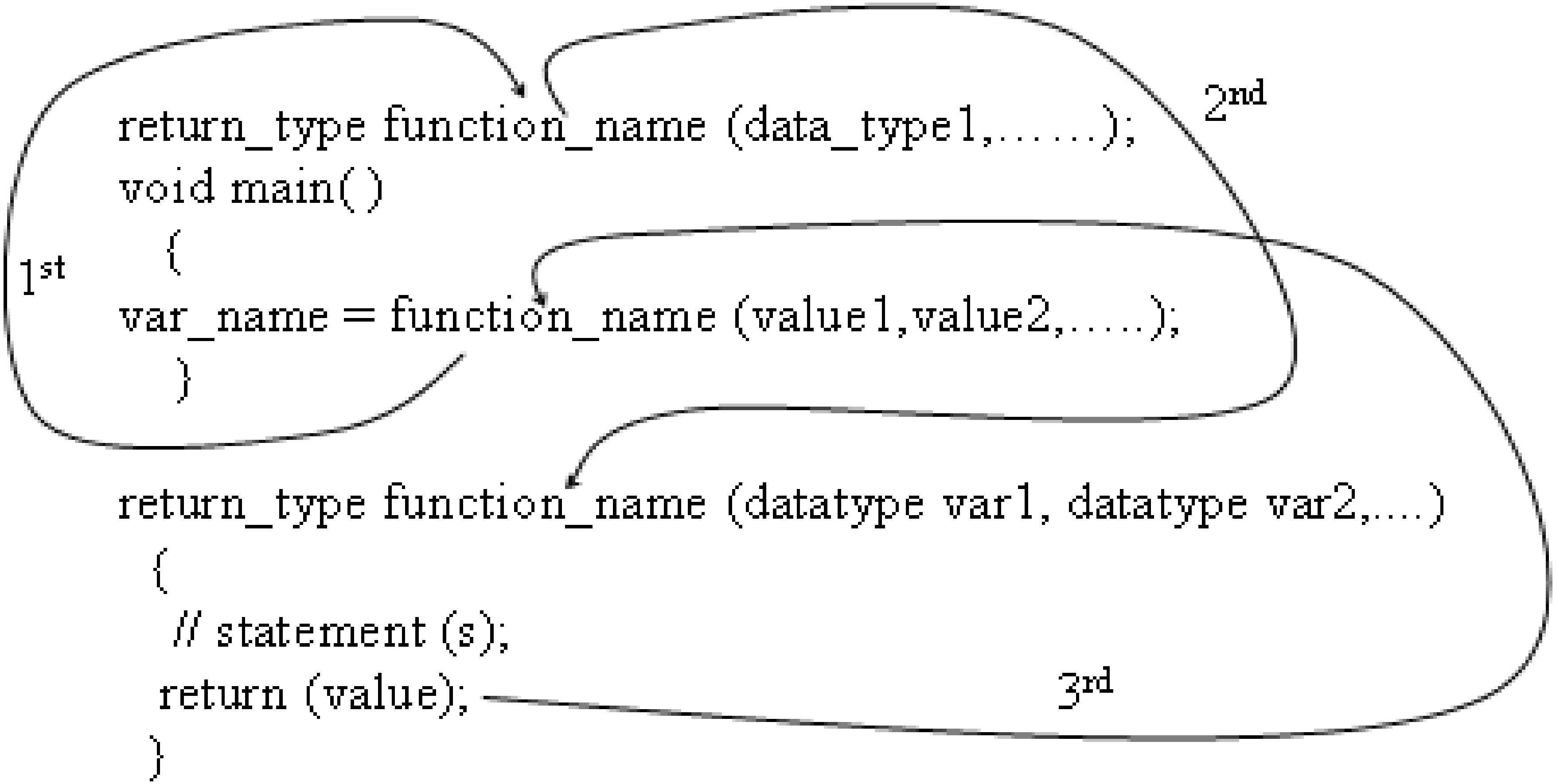
It has argument(s) and a return value. The accepting and printing are done in the calling program, where as processing is done in the called program.

Syntax : return_type function_name (data_type1, data_type2,….);

Example: float add_numbers (float, float, float);

```
char big (char, char);

float average ( float, float, float, float);

int digit_sum (long int);
```

Syntax:



```
return_type function_name (data_type1,……);
void main( )
   {
var_name = function_name (value1,value2,…..);
    }
return_type function_name (datatype var1, datatype var2,….)
  {
   // statement (s);
  return (value);
  }
```

Example

Method-1

```
int sum (int, int); //prototype
void main ( )
{
int m, n, res;
cout<<"Enter 2 numbers :";
cin>>m>>n;
res = sum (m, n);
cout<<“Sum = ”<<res<<endl;
}
int sum (int a, int b)
{
return a+b;
}
```

Method-2

```
int sum (int a, int b)
{
int c = a + b;
return c;
}
void main ( )
{
int m, n, res;
cout<<"Enter 2 numbers :";
cin>>m>>n;
res = sum (m, n);
cout<<“Sum = ”<<res<<endl;
}
```

Default Argument or Default value in a function

- If a formal argument is allowed with an initialization, such formal arguments are treated as default argument.

- The initializations always take place from **right to left**.

- During function calling, it is not necessary to pass that many values as arguments or parameters.

Syntax:

```
Return_Type FunctionName(DataType Var1=Value, DataType Var2=Value, …..)
{
   // Statement(s);
}
```

Example-1:

```
void Display(int a, int b=0)
{
   cout<<“A=“<<a<<“ B=“<<b<<endl;
}

void main()
{
   Display(12, 9);
   Display(14);
//  Display(5, 6, 7);   Too many arguments
//  Display( );          Too few arguments
}
```

Default argument or value

Output:

```
A=12 B=9
A=14 B=0
```

Example-2:

```
int sum (int p = 0, int q = 1, int r = 2)
{
    return p+q+r;     Default Arguments
}
void main( )
{
  int r1 = sum(4, 6, 10);
  int r2 = sum(10, 6);
  int r3 = sum(7);
  int r4 = sum( );

cout<<“Result-1=“<<r1<<endl;
cout<<“Result-2=“<<r2<<endl;
cout<<“Result-3=“<<r3<<endl;
cout<<“Result-4=“<<r4<<endl;
}
```

Output:

```
Result-1=20
Result-2=16
Result-3=8
Result-4=3
```

Inline Function

- It is just like macro in C.
- Its definition is prefixed by the keyword inline.

Syntax:

```
inline return_type function_name (argument(s) list, if any)
{
        // body
}
```

```
inline float Big (float a, float b, float c)
{
float d = a>b && a>c ? a : b>c ? b : c;
return d;
}
void main ( )
{
float p, q, r;
cout<<"Enter Three Numbers : ";
cin>>p>>q>>r;
float res = Big (p, q, r);
cout<<"Biggest : "<<res;

// (or) cout<<"Biggest : "<<Big (p, q, r);
}
```

FUNDAMENTAL FEATURES OF OOP

1. ENCAPSULATION
2. DATA ABSTARCTION
3. INHERITANCE (SINGLE)
4. INHERITANCE (MULTIPLE)
5. POLYMORPHISM
6. MESSAGE PASSING
7. DELEGATION
8. GENERICITY
9. EXTENSIBILITY
10. PERSISTENCE (not supported by C++)

WHAT IS OOP?

It is a programming methodology that associates **data structures** with a set of **operations** which act upon it.

OBJECTS

Every object will have data structures called **attributes** and behaviours called **operations**.

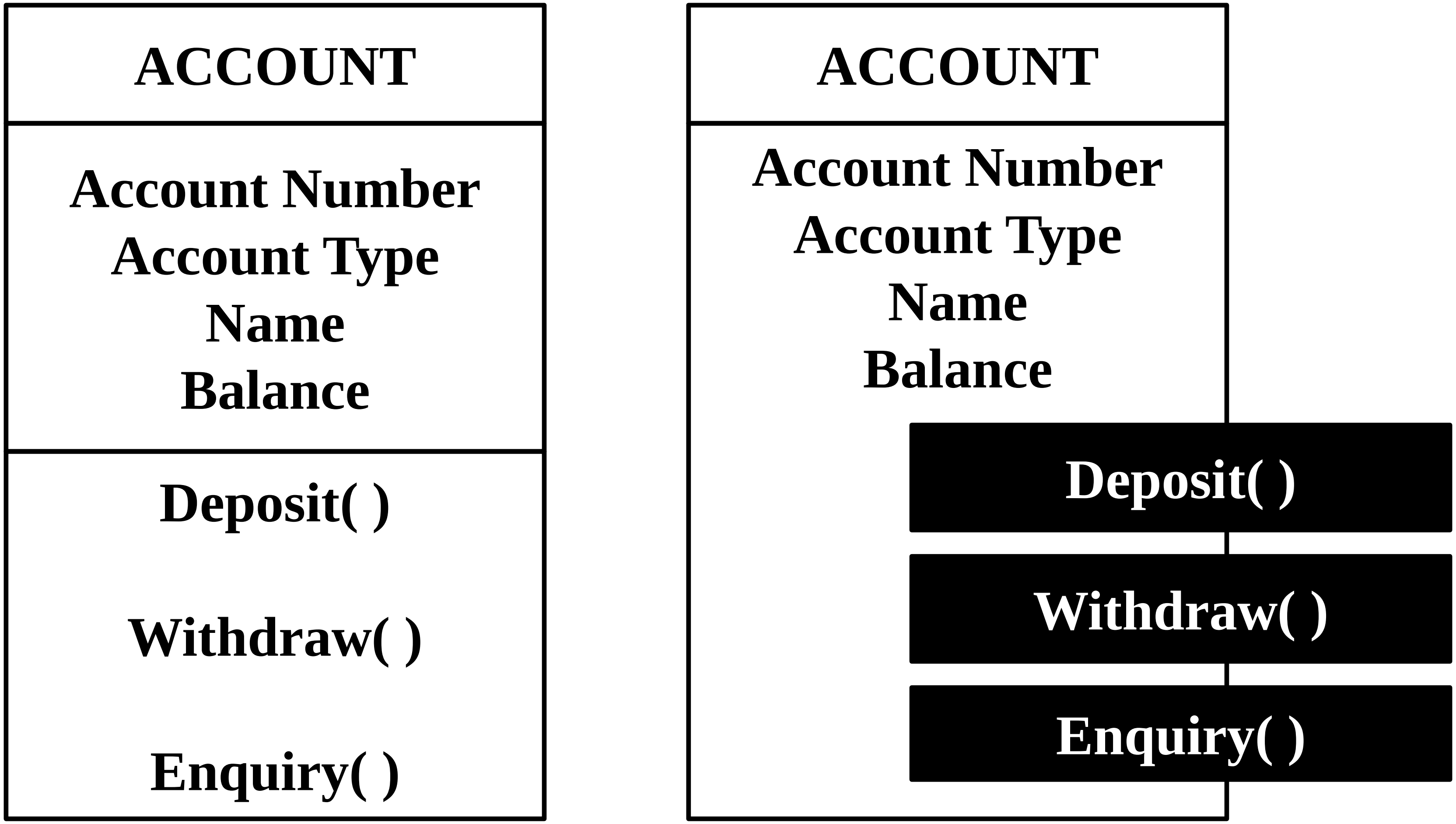
Example1:

An object called **ACCOUNT** having the **attributes**: Account Number, Account Type, Name, and Balance and **operations** are: **Deposit**, **Withdraw**, and **Enquiry**.

Example2:

An object called **PERSON** having the **attributes:** Name, Age, and Sex but they are not equal technically and **operations** are: **Speak**, **Listen**, and **Walk**.

CLASS	ATTRIBUTES	OPERATIONS
Vehicle	Name, Model, Color	Start, Stop, Speed
Person	Name, Age, Sex, Eye Color, Height	Speak, Walk, Eat
Polygon	Vertices, Border, Color, Fill Color	Draw, Erase, Move
Accounts	Account Number, Account Type, Name, Balance	Withdraw, Deposit, Enquiry



ACCOUNT

Account Number
Account Type
Name
Balance

Deposit( )

Withdraw( )

Enquiry( )

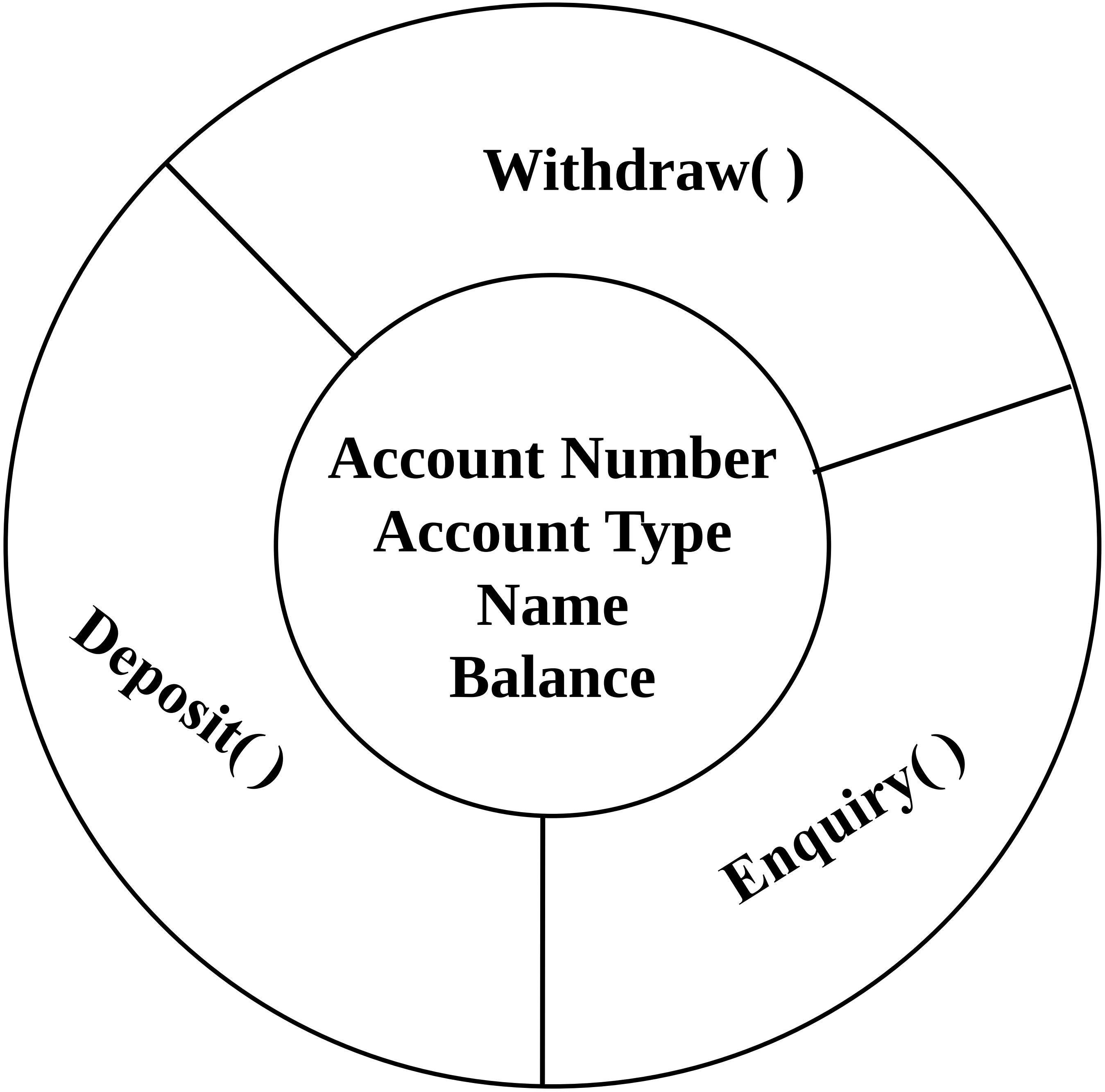
ACCOUNT

Account Number
Account Type
Name
Balance

Deposit( )

Withdraw( )

Enquiry( )



Withdraw( )

Account Number
Account Type
Name
Balance

Deposit( )

Enquiry( )

CLASS

- A class encloses both the data and function that operate on the data into a single unit.

- The **variables** and **functions** enclosed in a class are called **Data members** and **Member functions**.

- **Member function** defines the ***permissible operation*** on the **data members** of a class.

SYNTAX OF CLASS SPECIFICATION OR DECLARATION OR DEFINITION

```
class class_name
{
private:
      data member (s);
public:
      member function (s);
};
```

```
class class_name
{
      data member (s);
public:
      member function (s);
};
```

```
class values
{
private:
      int a;
      float b;
      char c;
public:
      void assign( int x, float y, char z)
      {
        a = x;
        b = y;
        c = z;
       }
      void display( )
      {
        cout<<a<<” “<<b<<” “<<c<<endl;
       }
};
```

The size of class is the combination of size of all the data members.

So the size of **values** class is 2 + 4 + 1 = 7 bytes.

OBJECTS

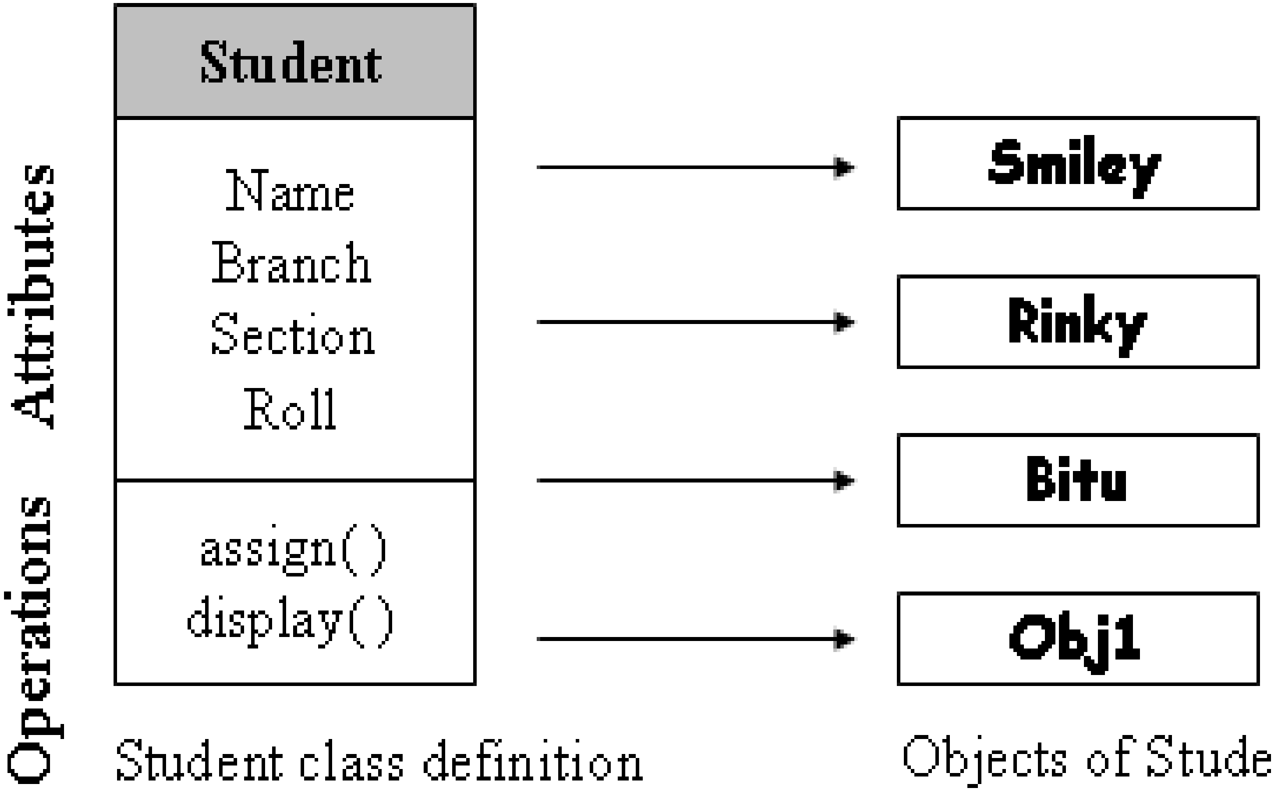
Defining variables of a class data type is known as ***class instantiation*** and such variables are called objects.

Syntax of defining object:

```
1)  class class_name
     {
       // body;
     } <object1>, <object2>,……;

2)  class class_name  <object1>,….;

3)  class_name  <object1>,........;
```

```
class Student
{
private:
   char name[20], branch[15], sec;
   int roll;
public:
   void assign( char a[], char b[], char c, int d)
// void assign( char *a, char *b, char c, int d)
   {
    strcpy (name, a);
    strcpy (branch, b);
    sec = c;
    roll = d;
   }
  void display( )
  {
    cout<<name<<” “<<branch<<” “<<sec<<” “<<roll<<endl;
   }
} Smiley, Rinky, Bitu, Obj1;
// Or    class Student Smiley, Rinky, Bitu, Obj1;
// Or    Student Smiley, Rinky, Bitu, Obj1;
```

```
void main()
{
// object assignments

Smiley.assign(“Smiley”,”ETC”,'A', 3147);
Rinky.assign(“Rinky”,”CSE”,'A', 1347);
Bitu.assign(“Bitu”,”IT”,'C', 1122);
Obj1.assign(“Kumar”,”Chemical”,'B', 2157);

// Display the contents of the object

Smiley.display();
Rinky.display();
Bitu.display();
Obj1.display();
}
```



Student

Name
Branch
Section
Roll

assign( )
display( )

Attributes

Operations

Smiley

Rinky

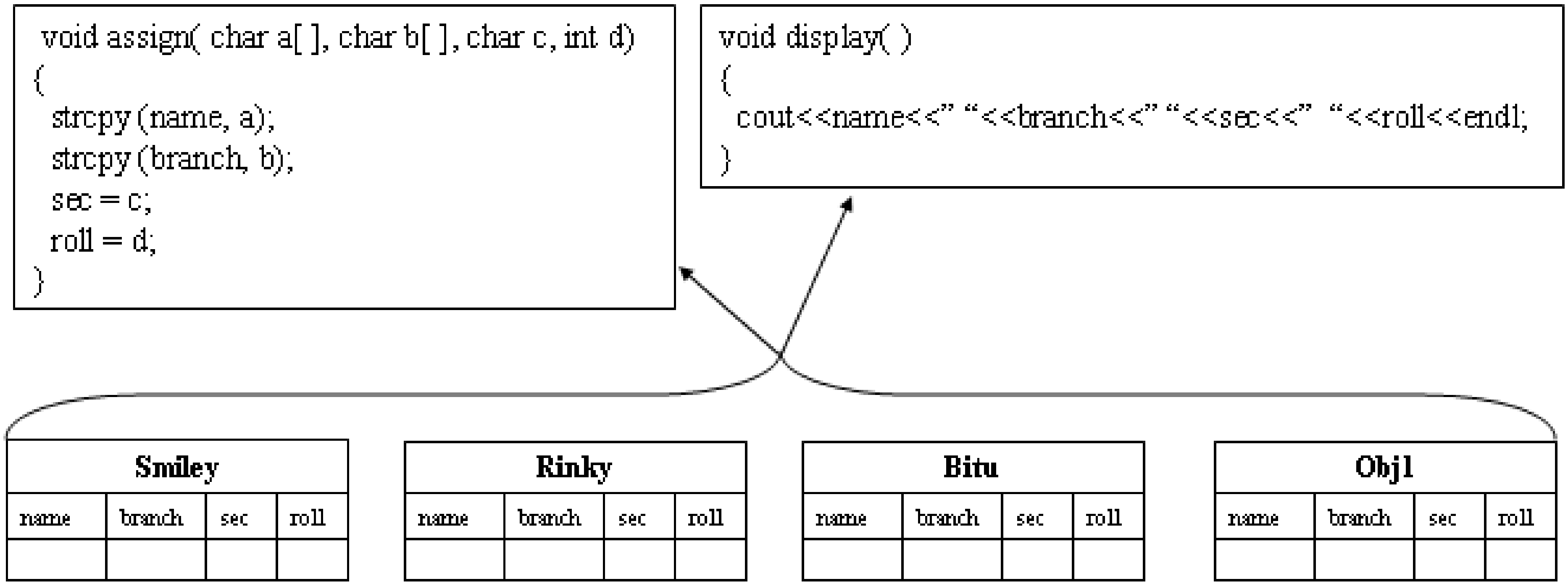
Bitu

Obj1

Student class definition

Objects of Student kind

How memory allocates in the respective objects of Student class?



```
 void assign( char a[ ], char b[ ], char c, int d)
{
  strcpy (name, a);
  strcpy (branch, b);
  sec = c;
  roll = d;
}
```

```
void display( )
{
  cout<<name<<” “<<branch<<” “<<sec<<”  “<<roll<<endl;
}
```

Smiley

name	branch	sec	roll

Rinky

name	branch	sec	roll

Bitu

name	branch	sec	roll

Obj1

name	branch	sec	roll

Objects of Student kind can access the member functions assign ( ) and display ( )

To assign the respective data in an object, the member function assign (char *, char *, char, int) or assign (char [ ], char [ ], char, int) is used.

To show the data of an object, the member function display ( ) is used.

These member functions are to be invoked or called through the respective object by using member access operator (. [Dot]).

Addition of two integer numbers.

(i) Using General Approach:

```
#include<iostream.h>
void main( )
{
int a, b, c;
cout<<”Enter two integers:”;
cin>>a>>b;
c = a+b;
cout<<”Sum of “<<a<<” and “<<b<<” = “<<c<<endl;
}
```

(ii) Using Function Approach:

```
#include<iostream.h>
int Sum(int a, int b)
{
  return(a+b);
}
void main( )
{
int a, b, c;
cout<<”Enter two integers:”;
cin>>a>>b;
c = Sum(a, b);
cout<<”Sum of “<<a<<” and “<<b<<” = “<<c<<endl;
}
```

(iii) Using Structure Approach:

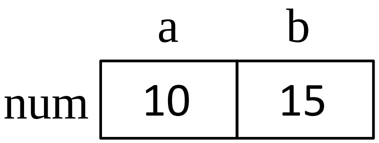
```
#include<iostream.h>
struct Numbers
{
int a, b;
};
void main( )
{
Numbers num;
cout<<”Enter two integers:”;
cin>>num.a>>num.b;
int c = num.a+num.b;
cout<<”Sum of “<<num.a<<” and “<<num.b<<” = “<<c<<endl;
}
```

(OR) Using Structure with Functions Approach:

```
struct Numbers
{
int a, b;
void Read( )
{
cout<<”Enter two integers:”;
cin>>a>>b;
}
void Result( )
{
int c = a+b;
cout<<”Sum of “<<a<<” and “<<b<<” = “<<c<<endl;
}
};
void main( )
{
Numbers num;
num.Read( );
num.Result( );
}
```

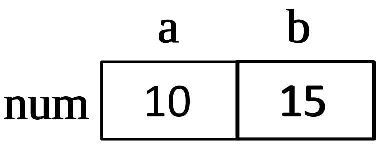
(v) Using Object Oriented Approach(Method-1):

```
#include<iostream.h>
class Numbers
{
private:
int a, b;
public:
void Read( )
{
cout<<”Enter two integers:”;
cin>>a>>b;
}
void Result( )
{
int c = a+b;
cout<<”Sum of “<<a<<” and “<<b<<” = “<<c<<endl;
}
};
void main( )
{
Numbers num;
num.Read( );
num.Result( );
}
```



(vi) Using Object Oriented Approach(Method-2):

```
#include<iostream.h>
class Numbers
{
private:
int a, b;
public:
void Read(int p, int q)
{
a = p;
b = q;
}
void Result( )
{
int c = a+b;
cout<<”Sum of “<<a<<” and “<<b<<” = “<<c<<endl;
}
};
```



	a	b
num	10	15

```
void main( )
{
Numbers num;
int a, b;
cout<<”Enter two integers:”;
cin>>a>>b;
num.Read(a, b);
num.Result( );
}
```

ACCESS CONTROL SPECIFIERS (or MODIFIERS)

Each user has different access privileges to the object.

private: In general data members are declared as private kind. Once the data members are declared as private, it becomes completely hidden to the outside of the world i.e. to the main program.

public: Generally the member functions are declared as public kind because the private data members can be accessed through the public member functions.

protected: If any data members are declared as protected then it is not visible to the main program and just act as private kind.

But these data members are visible to another class which is taking its properties for creation. It is important in inheritance technique of solving problems.

```
class PrivateProtected
{
private:
  int a;
protected:
  int b;
public:
  int c;
 void Assign(int p, int q)
{
    a = p;
    b = q;
}
void Show()
{
  cout<<”A=”<<a<<”B=”<<”C=”<<c<<endl;
}
};
```

	a	b	c
PP	5	8	12

```
void main()
{
PrivateProtected PP;
PP.Assign(5, 8);
PP.c=12;
PP.Show();
cout<<”A=”<<PP.a<<”B=”<<PP.b;  // invalid
cout<<”C=”<<PP.c;   // valid
}
```

Defining Member Functions in a class

1. Inline Member function

The definition is given inside the class declaration.

2. Outline Member function definition

- The prototype or declaration is given inside the class declaration; where as definition is given outside the class declaration.
- Loop control statements are allowed.

3. Outline Member function as Inline kind

The prototype or declaration is given inside the class declaration; where as definition is given outside the class declaration by prefixing the keyword **inline**.

```
// Sum of Two Numbers using Inline Member Function

class TwoNumberSum
{
  float a, b;
  public:
      // inline member functions definitions
        void Assign(float m, float n)
        {
           a = m;
           b = n;
         }
       void Sum()
        {
         float c = a + b;
         cout<<”Sum of “<<a<<” and “<<b<<” = “<<c<<endl;
        }
};
```

```
void main()
{
   TwoNumbrsSum  A;
   A.Assign(4.5, 10);
   A.Sum();
}
```

```
// Sum of Two Numbers using Outline Member Function
class TwoNumberSum
{
  float a, b;
  public:
// outline member function declaration or prototype
  void Assign(float , float);  // void Assign(float m, float n);
  void Sum();
 };
 // outline member function definition
void  TwoNumberSum::Assign(float m, float n)
{
   a = m;
   b = n;
}
// outline member function definition
void TwoNumberSum::Sum( )
{
   float c = a + b;
   cout<<”Sum of “<<a<<” and “<<b<<” = “<<c<<endl;
}
```

```
void main()
{
   TwoNumbrsSum  A;
   A.Assign(4.5, 10);
   A.Sum();
}
```

:: is known as scope resolution operator

```
// Multiplication table of number using Outline
// Member Function
class Table
{
  int num;
  public:
// inline member function definition
  void Assign(int n)
  {
     num = n;
   }
// outline member function declaration
  void ShowTable();
 };
 // outline member function definition
void Table::ShowTable( )
{
   for(int i=1; i<=10; i++)
   {
      int res = num * i;
      cout<<num<<” X “<<i<<” = “<<res<<endl;
    }
 }
```

```
void main()
{
   int num;
   Table  T;
   cout<<”Enter a number for Table=”;
cin>>num;
   T.Assign(num);
   T.ShowTable( );
}
```

```
// Sum of Two Numbers using Outline Member Function
// as inline kind.

class TwoNumberSum
{
  float a, b;
  public:
// outline member function declaration or prototype
  void Assign(float , float);  // void Assign(float m, float n);
  void Sum();
 };
 // outline member function definition as inline kind
inline void  TwoNumberSum::Assign(float m, float n)
{
   a = m;
   b = n;
}
inline void TwoNumberSum::Sum( )
{
   float c = a + b;
   cout<<”Sum of “<<a<<” and “<<b<<” = “<<c<<endl;
}
```

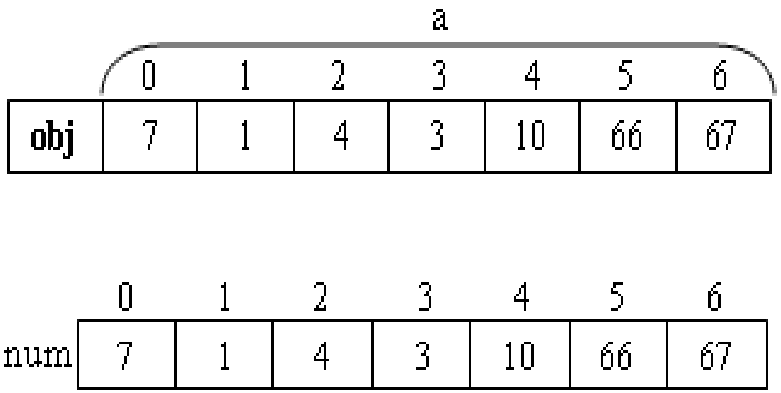
```
void main()
{
   TwoNumbrsSum  A;
   A.Assign(4.5, 10);
   A.Sum();
}
```

Single dimension array as data member in a class

Assign {7, 1, 4, 3, 10, 66, 67} in an array and display it using class and object.

```
class Array
{
private:
      int a[7];
public:
      void Assign (int []);  // void Assign(int *);  prototype
      void Show();              // prototype
};
void Array::Assign (int b[]) // void Array::Assign (int *b) definition
{
      for (int i=0; i<7; i++)
              a[i] = b[i];
}
```

```
void Array::Show ( ) // definition
{
    cout<<”All 7 Numbers\n”;
          for (int i=0; i<7; i++)
               cout<<a[i]<<” “;
    cout<<endl;
}
void main ( )
{
Array obj;
int num[ ]={7, 1, 4, 3, 10,66, 67};
obj.Assign (num);
obj.Show( );
}
```



a

	0	1	2	3	4	5	6
obj	7	1	4	3	10	66	67

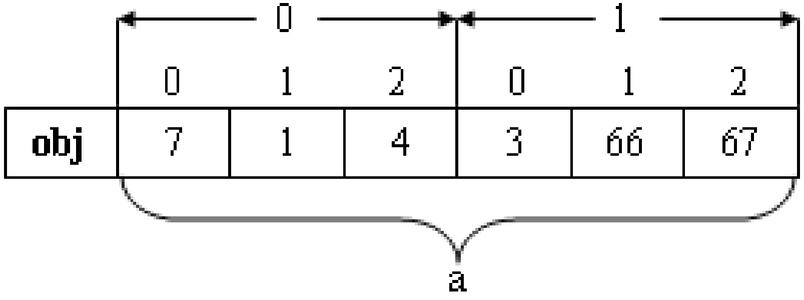
	0	1	2	3	4	5	6
num	7	1	4	3	10	66	67

Double dimension array as data member in a class

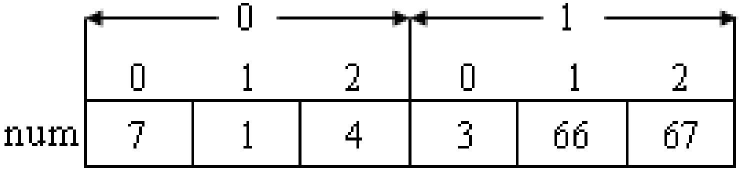
Assign {7, 1, 4, 3, 66, 67} in a 2X3 matrix and display it using class and object.

```
class matrix
{
          int a[2][3];
public:
           void Assign (int [][3]); // void Assign (int [2][3]);
           void Show ( );
};

void matrix::Assign (int b[][3])
{
   for (int i=0;i<2; i++)
  {
    for (int j=0; j<3; j++)
      a[i][j] = b[i][j];
   }
}
```

```
void matrix::Show ()
{
   for (int i=0;i<2;i++)
  {
     for (int j=0; j<3; j++)
          cout<<setw(4)<<a[i][j];
     cout<<endl;
   }
}

void main ( )
{
int mat[][3] = {{1, 4, 3}, {7, 66, 67}};
matrix obj;
obj.Assign (mat);
obj.Show ( );
}
```



	0			1		
	0	1	2	0	1	2
obj	7	1	4	3	66	67



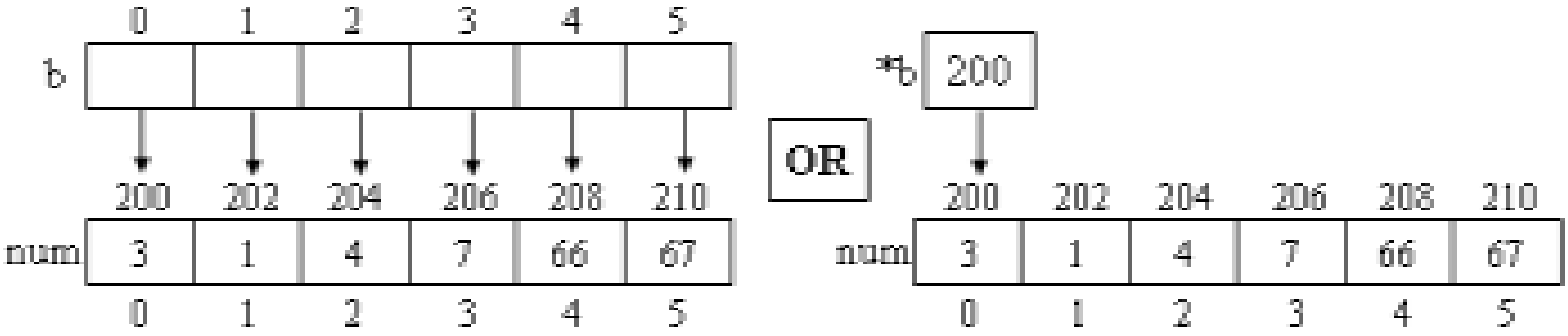
	0			1		
	0	1	2	0	1	2
num	7	1	4	3	66	67

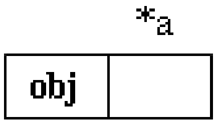
Pointer as data member in a class

Assign 3, 1, 4, 7, 66, and 67 in a pointer. And display it.

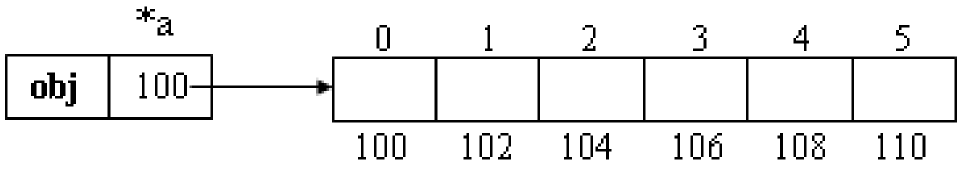
```
 class Pointer
{
private:
        int *a;
public:
      void Assign(int *);   // prototype
       void Show();          //prototype
};

void Pointer::Assign (int b[ ])  // Assign (int *b)
{
    a = new int[6];       // memory allocation to pointer variable ‘a’
 for (int i=0; i<6;i++)
        a[i] =b[i];          // *(a+i) =*(b+i);
}
```

```
void Pointer::Show ( ) // definition
{
 cout<<"All 6 Numbers\n";
for (int i=0; i<6; i++)
       cout<<a[i]<<" ";    // cout<<*(a+i)<<" ";
cout<<endl;
// to free the allocated memory from the pointer*a
delete a;
}
// main program
void main ()
{
 Pointer obj;
 int num[ ]={3, 1, 4, 7, 66, 67};
 obj.Assign (num);
 obj.Show ( );
}
```

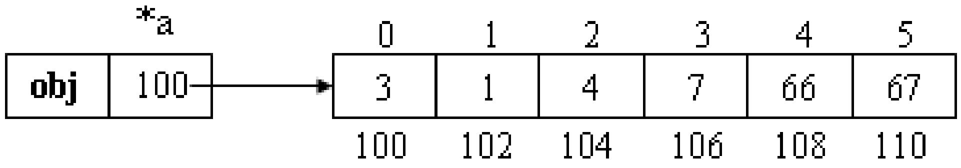




Object is created with *a as data member



Memory is allocated to the pointer *a



Array num is assigned to the allocated memory

Array of Objects

Declaration of a set of sequential memory blocks of class kind.

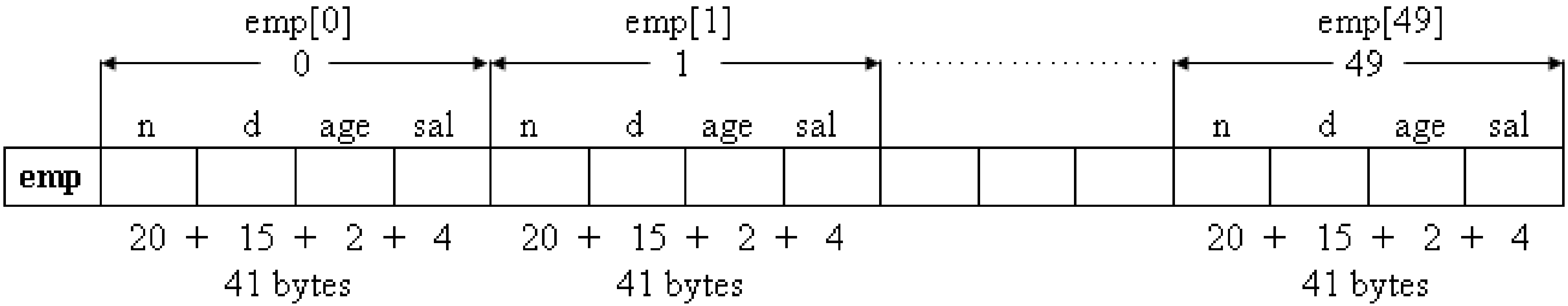
Syntax:

```
class class_name
{
private:
// data member (s);
public:
// member function (s);
};

class_name object_name [size];
```

Read n employees name, designation, age, and salary in an array of object. Then display in this format: Name Designation Age Salary

```
class Employee
{
       char n[20], d[15];
       int age;
       float sal;
public:
  void Get ( )
  {
     cin>>n>>d>>age>>sal;
   }
 void Out ( )
 {
    cout<<setw (20)<<n<<setw (15)<<d<<setw (5)<<age<<setw (7)<<sal<<endl;
  }
};
```

```
void main ( )
{
  Employee emp[50];
  int n;
  cout<<”How many employee details to be input ?”;
  cin>>n;
for (int i=0; i<n; i++)
{
cout<<”Enter employee “<<i+1<<” Name, Designation, Age, and Salary : “;
emp[i].Get( );
}

cout<<setw(20)<<”Name”<<setw(15)<<”Designation”<<setw(5);
cout<<”Age”<<setw(7)<<” Salary<<endl;
for (i=0; i<n; i++)
    emp[i].Out( );
}
```



Size of array of object **emp** = 50X41 = 2050 bytes

Nested Class

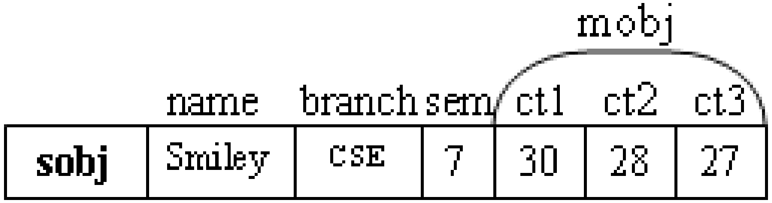
Declaring a class inside another class is known as nested class. The inner class declaration must be under public kind.

Syntax:

```
class outer_class_name
{
private:
     Data_member (s);
public:
     Member_function (s);
     class inner_class_name
    {
       private:
            data_member (s);
       public:
            member_function (s);
      } object_name;
};
```

Nested Class Example

```
class student
{
private:
      char name[20], branch[10];
      int sem;
public:
   void Assign (char a[ ], char b[ ], int c)
  {
    strcpy (name, a);    strcpy (branch, b);
    sem = c;
   }
   void Show ( )
  {
    cout<<name<<”   “<<branch<<”  ”<<sem<<endl;
   }
  class Marks
  {
    private:
          int ct1, ct2, ct3;
    public:
         void Assign (int a, int b, int c)
         {
            ct1= a;    ct2= b;  ct3= c;
         }
   void Show ( )
   {
     int tot = ct1 + ct2 + ct3;
     cout<<"Internal Total="<<tot<<endl;
   }
    } mobj;
};

void main()
{
student sobj;

sobj.Assign("Smiley", "CSE", 7);
sobj.mobj.Assign(30, 28, 27);

sobj.Show ( );
sobj.mobj.Show ( );
}
```



				mobj		
	name	branch	sem	ct1	ct2	ct3
sobj	Smiley	CSE	7	30	28	27

Passing Objects As Arguments

Pass – by –value
Pass – by –reference
Pass – by – pointer or address

1) Pass – by –value

A copy of the object is passed to the function and any modification made to the object inside the function is not reflected in the object used in the calling function.

i. Object as argument in a member function Syntax:

```
class class_name
{
private:
            // data member (s);
public:
        void function_name (class_name object1, class_name object2,……)
       {
            // statement(s);
        }
};
```

```
// Addition of complex numbers 7 + i14 and -6 +
i7.
class Complex
{
    int real, img;
public:
  void Assign (int a, int b)
  {
    real = a;
    img = b;
  }
  void Display ( )
  {
    cout<<real<<" + i "<<img<<endl;
   }
 void Add (Complex A, Complex B)
 {
   real = A.real + B.real;
   img = A.img + B.img;
  }
};
```

	real	img
A	7	14

	real	img
B	-6	7

```
void main ( )
{
Complex C1, C2, C3;

// Creation of 1st complex number
C1.Assign(7,14);

// Creation of 2nd complex number
C2.Assign (-6, 7);

// Addition of complex numbers
C3.Add (C1, C2);
cout<<"1st Complex Number : ";
C1.Display ();
cout<<"2nd Complex Number : ";
C2.Display ();
cout<<"Addition of Complex Numbers :";
C3.Display ();
}
```

	real	img
C1		

	real	img
C2		

	real	img
C3		

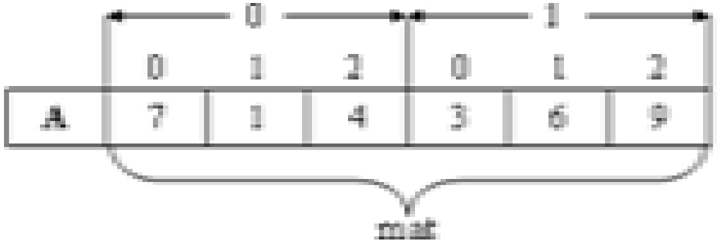
	real	img
C1	7	14

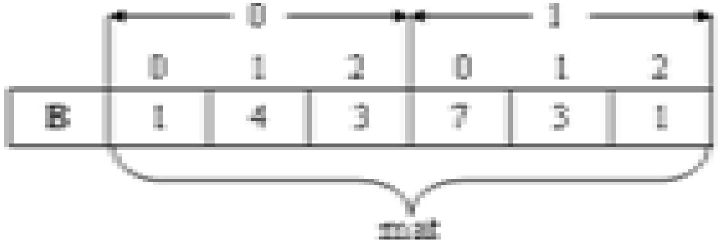
	real	img
C2	-6	7

	real	img
C3	1	21

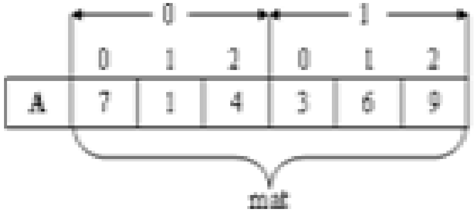
```
// Addition and subtraction of two 2X3 matrices.
classMatrix   // Size of class Matrix = (2X3)X4 bytes = 24bytes
{
     float mat[2][3];
public:
// outline member function’s prototype
   void Get ( );
   void Display ( );
   void Add (Matrix, Matrix);
   void Sub (Matrix, Matrix);
} ;
// outline member function’s definition
void Matrix :: Get ( )
{
for (int i=0, j; i<2; i++)
    for (int j=0; j<3; j++)
   {
      cout<<”Enter a Number : “;
      cin>>mat[i][j];
    }
}
```

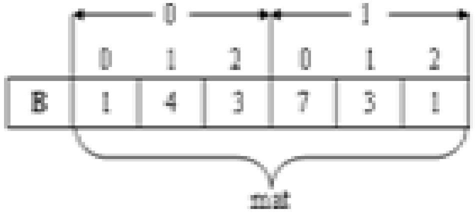
```
void Matrix :: Display ( )
{
for (int i=0; i<2; i++)
{
for (int j=0; j<3; j++)
       cout<<mat[i][j]<<”  “;
cout<<endl;
}
}
void Matrix :: Add (Matrix A, Matrix B)
{
    for (int i=0, j; i<2; i++)
        for (int j=0; j<3; j++)
             mat[i][j] = A.mat[i][j] + B.mat[i][j];
}
```

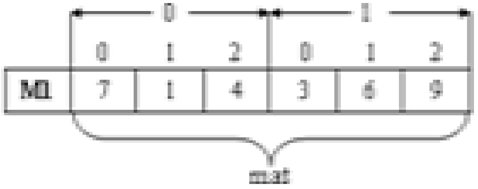


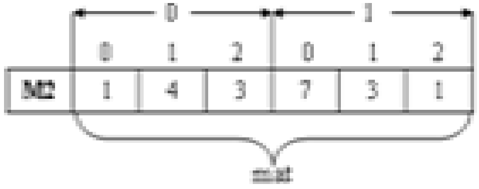


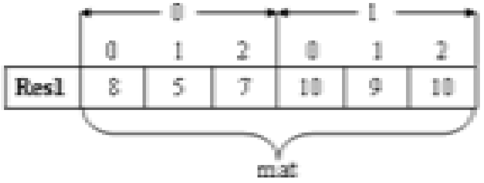
```
void Matrix :: Sub (Matrix A, Matrix B)
{
for (int i=0, j; i<2; i++)
   for (int j=0; j<3; j++)
        mat[i][j] = A.mat[i][j] - B.mat[i][j];
}
void main ( )
{
Matrix M1, M2, Res1, Res2 ;
cout<<"Enter Numbers in 1st Matrix\n";
M1.Get ( );
cout<<"Enter Numbers in 2nd Matrix\n";
M2.Get ( );
Res1.Add (M1, M2);
Res2.Sub (M1, M2);
cout<<"1st Matrix\n";
M1.Display ( );
cout<<"2nd Matrix\n";
M2.Display ( );
cout<<"Matrix Addition\n";
Res1.Display ( );
cout<<"Matrix Subtraction\n";
Res2.Display ( );
}
```

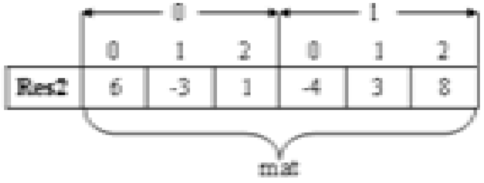












ii. Object as return value from a member function Syntax:

```
class class_name
{
private:
           // data member (s);
public:
  class_name function_name (class_name object1, class_name object2,……)
  {
      // statement(s);
         return object;
   }
};
```

(2) Pass-by-reference

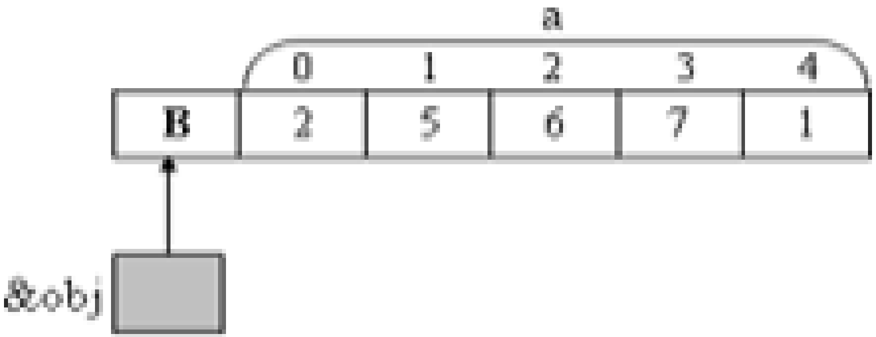
The value object is passed to the function and any change made to the object inside the function is reflected in the actual object.

Syntax:

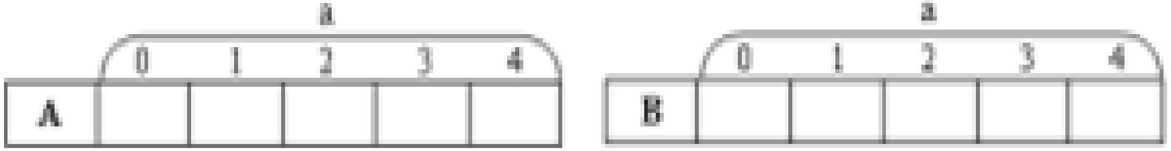
```
class class_name
{
public:
return_type function_name (class_name &object1, class_name &object2,……)
{
// statement(s);
}
};
```

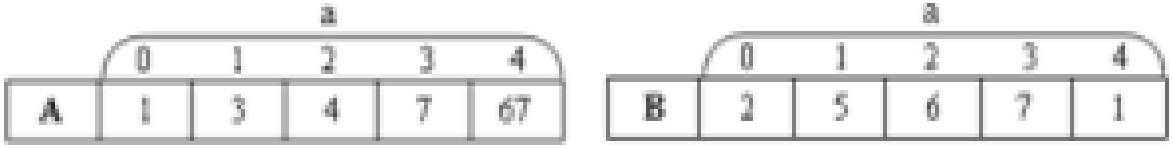
Assign {1, 3, 4, 7, 67} in an array. Change every element by incrementing by 2 using reference objects in a member function.

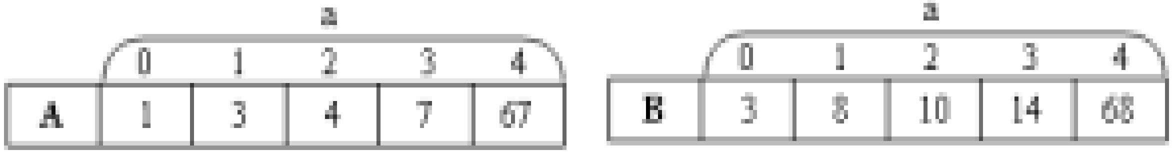
```
class Array
{
       int a[5];
   public:
      void Assign (int []);
      void Change (Array &);
      void Show ( );
};
void Array::Assign (int b[])
{
for (int i=0; i<5; i++)
          a[i] = b[i];
}
void Array::Change (Array &obj)
{
   for (int i=0; i<5; i++)
      obj.a[i]=obj.a[i] + a[i];
}
```



```
void Array::Show ( )
{
for (int i=0; i<5; i++)
  cout<<a[i]<<”  “;
cout<<endl;
}
void main ( )
{
int arr1[ ] = {1, 3, 4,7, 67};
int arr2[ ] = {2, 5, 6, 7, 1};
Array A, B;

A.Assign(arr1);
B.Assign(arr2);
cout<<”1st Array Elements\n”;
A.Show ();
cout<<”2nd Array Elements\n”;
B.Show ( );

A.Change(B);
cout<<”After change the 2nd Array Elements\n”;
B.Show ( );
}
```







(3) Pass-by-address or pointer

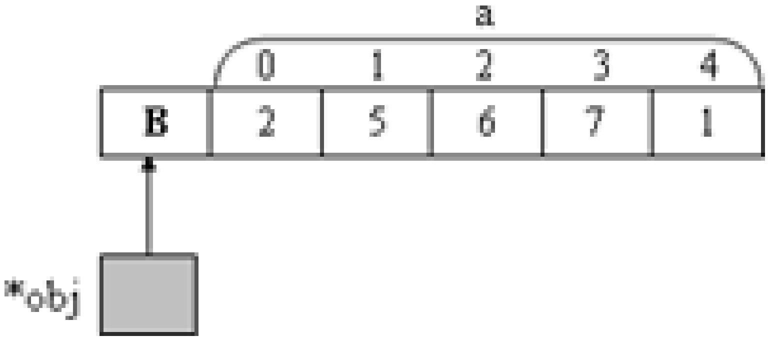
An address of the object is passed to the function and any change made to the object inside the function is reflected in the actual object.

Syntax:

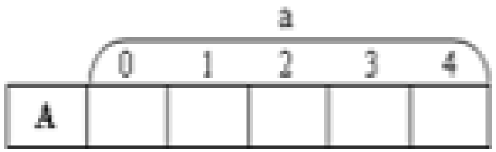
```
class class_name
{
public:
return_type function_name (class_name *object1, class_name
*object2,……...)
{
    // statement(s);
}
};
```

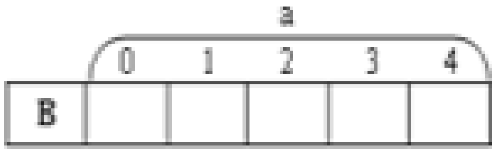
Assign {1, 3, 4, 7, 67} in an array. Change every element by incrementing by 2 using pointer to object in a member function.

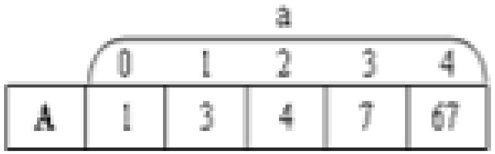
```
class Array
{
int a[5];
public:
     void Assign (int []);
     void Change (Array *);
     void Show ( );
};
void Array::Assign (int b[])
{
for (int i=0; i<5; i++)
      a[i] = b[i];
}
void Array::Change (Array *obj)
{
    for (int i=0; i<5; i++)
     obj→a[i]=obj→a[i] + a[i];
}
```

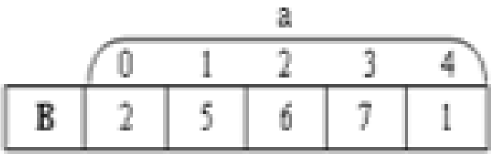


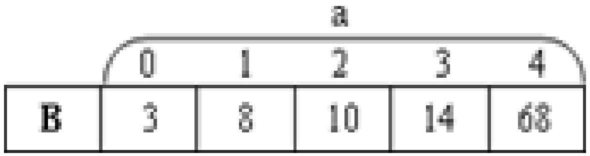
```
void Array::Show ( )
{
for (int i=0; i<5; i++)
cout<<a[i]<<"";
cout<<endl;
}
void main ( )
{
int arr1[] = {1, 3, 4, 7, 67};
int arr2[] = {2, 5, 6, 7, 1};
Array A, B;
A.Assign (arr1);
B.Assign(arr2);
cout<<"1st Array Elements\n";
A.Show ();
cout<<"2nd Array Elements\n";
B.Show ( );
A.Change (&B);
cout<<"After change the 2nd Array Elements\n";
B.Show ( );
}
```









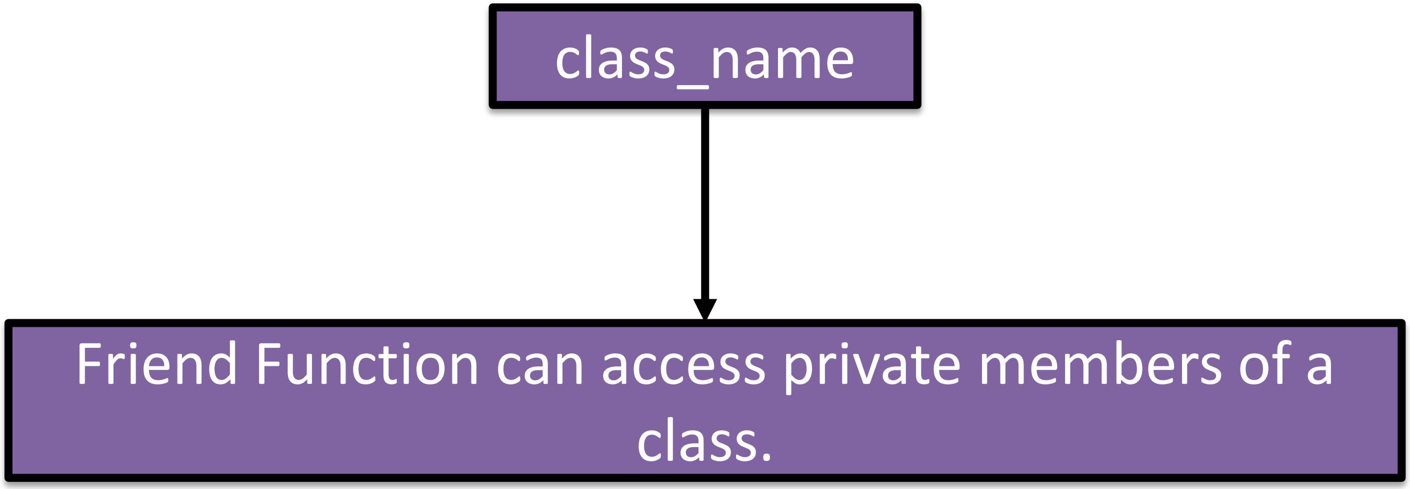


Friend Functions

- To allow functions outside a class to access and manipulate the private members of the class.
- In C++ it is achieved by using the concept of friends.
- The function declaration must be prefixed by the keyword **friend** where as the function definition must not.
- The definition of a friend function outside of the class as a normal function definition.
- A friend function’s prototype can be given anywhere in a class.

Syntax for accessing private data members of a class in a friend function:

```
class class_name
{
     friend return_type function_name (class_name);
};

return_type function_name (class_name object)
{
// statement (s);
}
```

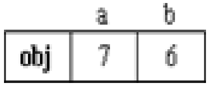


class_name

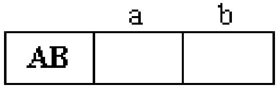
Friend Function can access private members of a class.

Find the sum of any two numbers using friend function.

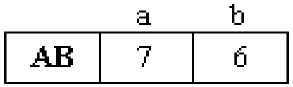
```
#include <iostream.h>
class NumberAdd
{
    float a, b;
public:
void Read ()
{
  cout<<"Enter two Numbers:";
  cin>>a>>b;
}
friend float Add (NumberAdd obj)  // friend function inline definition
{
    return obj.a + obj.b;
```



```
}
};
void main ( )
{
NumberAdd AB;
```



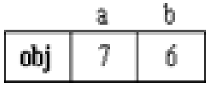
```
AB.Read ( );
```

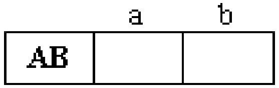


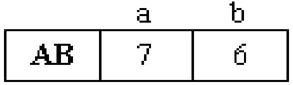
```
float res = Add (AB); // calling friend function
cout<<"Sum = "<<res;
}
```

(OR)

```
class NumberAdd
{
     float a, b;
public:
void Read ()
{
cout<<"Enter two Numbers:";
cin>>a>>b;
}
friend float Add(NumberAdd);  // friend function declaration
};
// friend function outline definition
float Add(NumberAdd obj)
{
    return obj.a + obj.b;
}
void main ( )
{
NumberAdd AB;
AB.Read ( );
float res = Add (AB); // calling friend function
cout<<"Sum : "<<res;
}
```



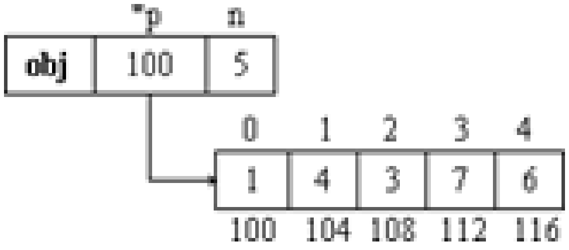


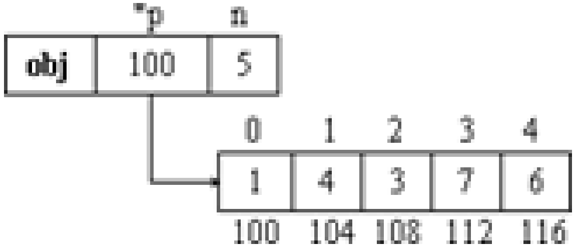


Find smallest and biggest of ‘n’ numbers using friend function.(Two friend functions)

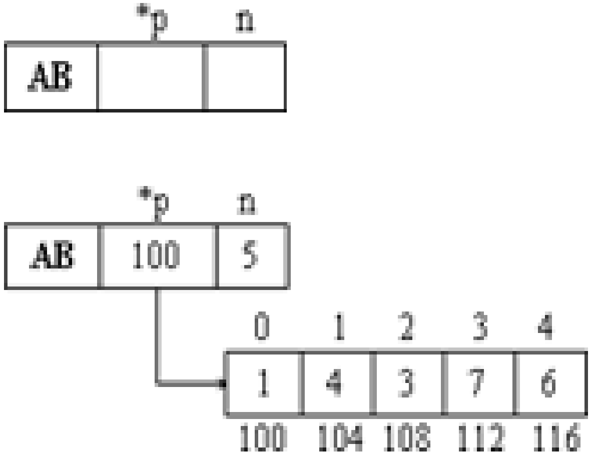
```
class MaxMin
{
    float *p;
    int n ;
public:
    void Read ( );
    void Show ( );
    friend float Max (MaxMin);   // friend function declaration
    friend float Min (MaxMin) ;  // friend function declaration
};
void MaxMin::Read ( )
{
   cout<<”Enter Size of Array :”;
   cin>>n;
   p = new float[n];
    for (int i=0; i<n; i++)
    {
      cout<<”Enter a Number :”;
      cin>>p[i];
    }
}
```

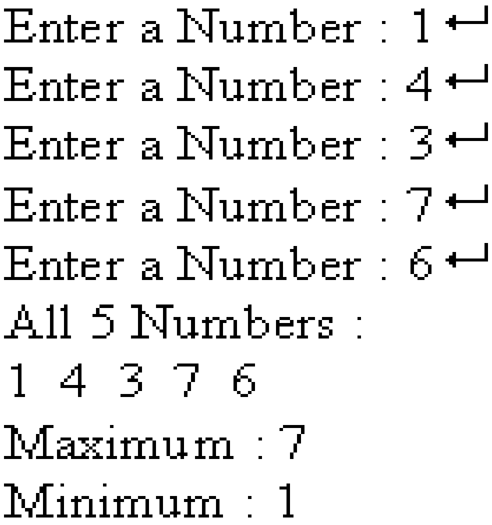
```
void MaxMin::Show ( )
{
cout<<”All ”<<n<<” Numbers :\n”;
for (int i=0; i<n; i++)
   cout<<p[i]<<” “;
delete p;
}

float Max (MaxMin obj) // friend function definition
{
float a = obj.p[0];
for (int i=1; i<obj.n; i++)
   if (obj.p[i] > a) a = obj.p[i];
return a;
}

float Min (MaxMin obj) // friend function definition
{
float a = obj.p[0];
for (int i=1; i<obj.n; i++)
     if (obj.p[i] < a) a = obj.p[i];
return a;
}
```





```
void main()
{
MaxMin AB;
AB.Read ( );
float a = Max (AB); // calling friend function
float b = Min (AB); // calling friend function
AB.Show ( );
cout<"Maximum : "<<a<<endl;
cout<<"Minimum : "<<b<<endl;
}
```

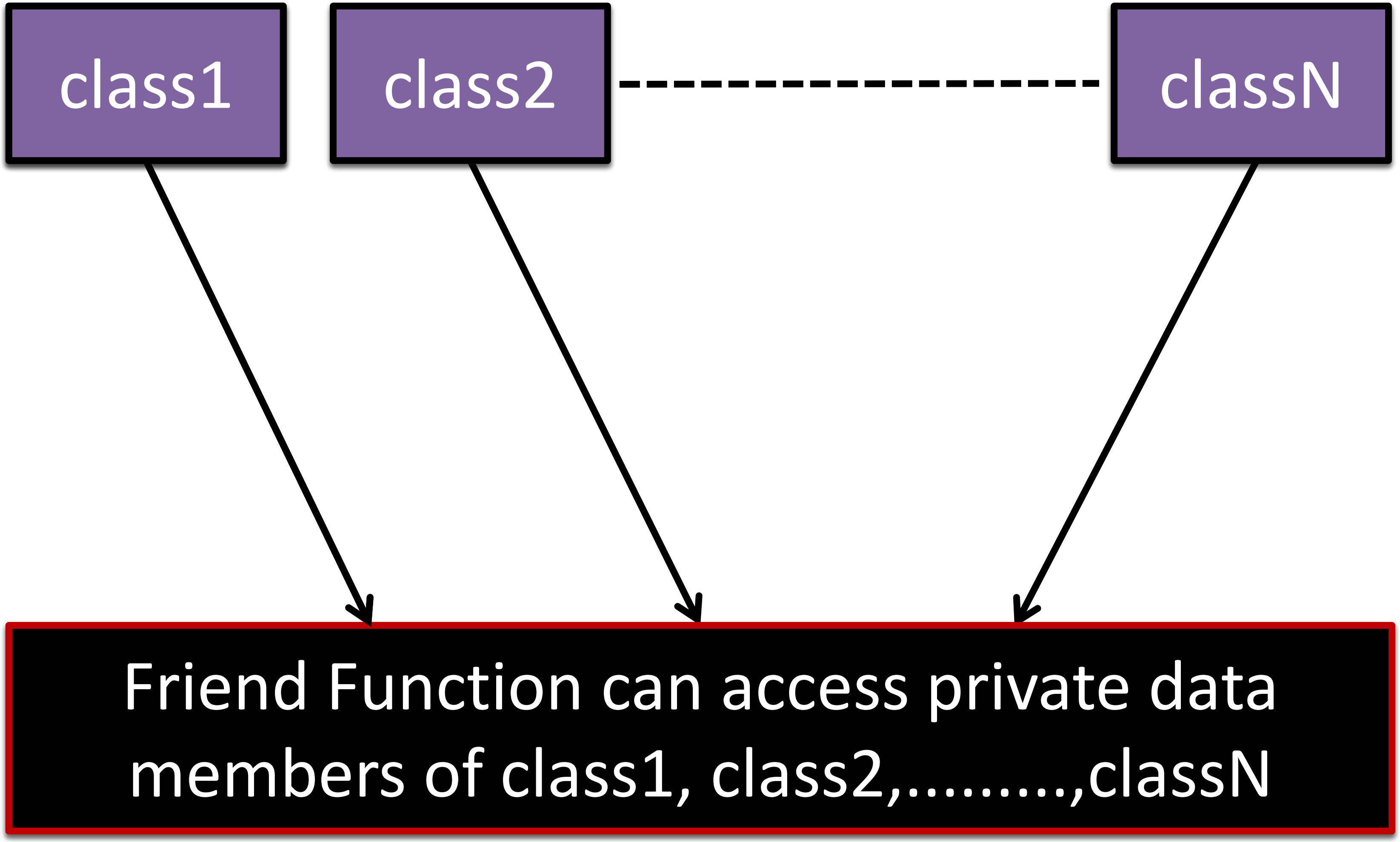




```
Enter a Number : 1 ↵
Enter a Number : 4 ↵
Enter a Number : 3 ↵
Enter a Number : 7 ↵
Enter a Number : 6 ↵
All 5 Numbers :
1  4  3  7  6
Maximum : 7
Minimum : 1
```

Syntax for accessing private data members of more than one class in a friend function:

```
class class_name2;
…………………….
…………………….
class class_nameN;
class class_name1
{
   friend return_type function name (class_name1,….., class_nameN);
};
class class_name2
{
   friend return_type function_name (class_name1,….., class_nameN);
};
……………………………..
……………………………..
class class_nameN
{
   friend return_type function_name (class_name1,….., class_nameN);
};
   return_type function_name (class_name1 object1,….., class_nameN objectN)
   {
    // statement (s);
   }
```



class1

class2

classN

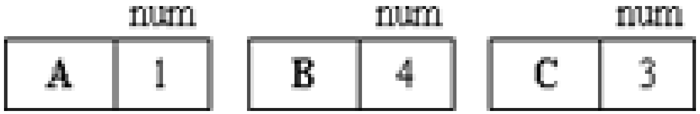
Friend Function can access private data members of class1, class2,.........,classN

Addition of three integers using friend functions.

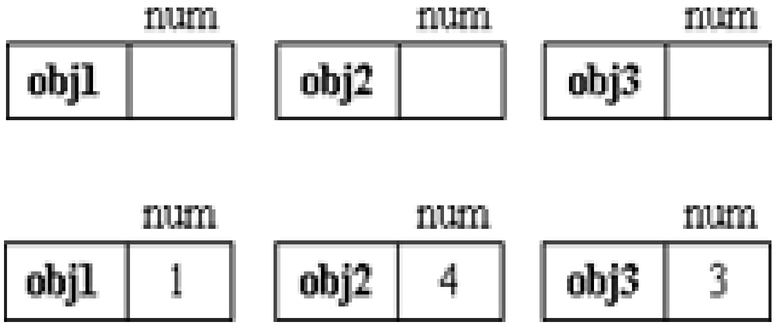
```
class Three;   // forward references
class Two;     // forward references
class One
{
   int num;
public:
void Get ( )
{
cout<<”Enter 1st Number:”;
cin>>num;
}
void Out ( )
{
cout<<”First Number : “<<num<<endl;
}
    friend void Sum (One,Two,Three);        // friend function prototype
};
```

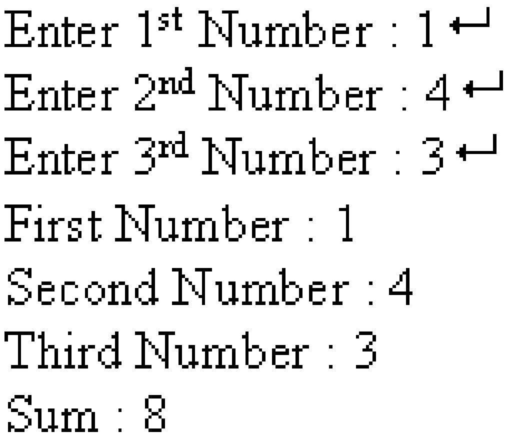
```
class Two
{
   int num;
public:
void Get ( )
{
cout<<”Enter 2nd Number:”;
cin>>num;
}
void Out ( )
{
cout<<”Second Number : “<<num<<endl;
}
   friend void Sum (One,Two,Three);        // friend function prototype
};
```

```
class Three
{
    int num;
public:
void Get ( )
{
cout<<”Enter 3rd Number:”;
cin>>num;
}
void Out ( )
{
cout<<”Third Number : “<<num<<endl;
}
    friend void Sum (One,Two,Three);        // friend function prototype
};
void Sum (One A, Two B, Three C) // friend function definition outside the class
{
   int r = A.num + B.num + C.num;
   cout<<”Sum : “<<r<<endl;
}
```



```
// main program
void main ( )
{
One obj1;
Two obj2;
Three obj3;
obj1.Get ( );
obj2.Get ( );
obj3.Get ( );
obj1.Out ( );
obj2.Out ( );
obj3.Out ( );
Sum (obj1, obj2, obj3); // calling friend function
}
```





```
Enter 1st Number : 1 ↵
Enter 2nd Number : 4 ↵
Enter 3rd Number : 3 ↵
First Number : 1
Second Number : 4
Third Number : 3
Sum : 8
```

Constructor

It is a special member function having name of class itself.

It has no return value but may or may not have arguments.

It invokes automatically when class gets instantiated.

There are four types of constructors.

(i) Automatic or default constructor:

- It has no argument.
- It invokes automatically as soon as class gets instantiated.

Syntax:

```
class class_name
{
private:
     // data member(s);
public:
        class_name( );   // constructor prototype
};
// constructor definition
class_name :: class_name ( )
{
     // statement(s);
}
```

```
// Using constructor
class Number
{
      int a;
public:
Number ( ) // default constructor definition
{
    a = 0;
}
void Show ( )
{
     cout<<"A="<<a<<endl;
}
};
void main ()
{
Number A, B, C;
A.Show ();
B.Show ();
C.Show ();
}
```

```
// Without constructor
class Number
{
       int a;
public:
void Assign ( )
{
     a = 0;
}
void Show ( )
{
     cout<<"A="<<a<<endl;
}
};
void main( )
{
  Number A, B, C;
A.Assign();   B.Assign();  C.Assign();
A.Show ();
B.Show ();
C.Show ();
}
```

(ii) Parameterized constructor:

- It has argument (s).
- Two different ways the constructor get invoked automatically.
- They are implicit way and explicit way.
- To assign different values to the data members of objects at the time of declaration.

Syntax:

```
class class_name
{
private:
// data member(s);
public:
class_name (data_type1 var1, ……)
{
// statement(s);
}
};
void main ( )
{
// implicit call
class_name  object_name (value1, value2,……);

// explicit call
class_name  object name = class_name (value1, value2 …);
}
```

```
class Number
{
    int a;
public:
Number (int); // parameterized constructor declaration or prototype
void Show ( )
{
cout<<"A="<<a<<endl;
}
};
Number :: Number (int b) // parameterized constructor definition
{
a = b;
}
void main ( )
{
Number A(6);                     // Implicit call
Number B = Number (7);    // Explicit call
A.Show ();
B.Show ();
}
```

Constructor Overloading:

To declare a constructor more than once with different parameter list.

```
class Number
{
	int a;
public:
Number ( ) // default constructor definition
{
a = 0;
}
Number (int b) // parameterized constructor definition
{
a = b;
}
void Show ( )
{
cout<<”A=”<<a<<endl;
}
};
```

```
void main ()
{
    Number  A;
    Number  B(6);                        // Implicit call
    Number C = Number (7);      // Explicit call
    Number D = 67;                      // Explicit call

   A.Show ( );
   B.Show ();
   C.Show ();
   D.Show ();
}
```

Constructor with default argument

```
class Number
{
int a;
public:
Number (int b=0)  // parameterized constructor
                              // definition with default argument
{
a = b;
}
void Show ( )
{
cout<<”A=”<<a<<endl;
}
};
```

```
void main ()
{
 // Automatic call
   Number  A;
 // Implicit call
   Number B(6);
// Explicit call
   Number C = Number ( );
// Explicit call
   Number D= 67;
A.Show ();
B.Show ();
C.Show ();
D.Show ();
}
```

(iii) Copy constructor:

The constructor's argument as reference object kind then that constructor is said to be copy constructor.

Syntax:

```
class class_name
{
public:
    class_name (class_name &object_name)
   {
     // statement(s);
    }
};
```

```
class ARS
{
   int a;
public:
ARS ( )
{
   a=7;
}
ARS (int b)
{
   a=b;
}
ARS (ARS &obj)  // copy constructor
{
    a=obj.a;
}
void show( )
{
   cout<<"A="<<a<<endl;
}
};
```

```
void main ( )
{
ARS A;
ARS B(6);
ARS C(A);  // invokes copy constructor
ARS D = B; // invokes copy constructor
A.show ( );
B.show ( );
C.show ( );
D.show ( );
}
```

(iv) Dynamic Constructor:

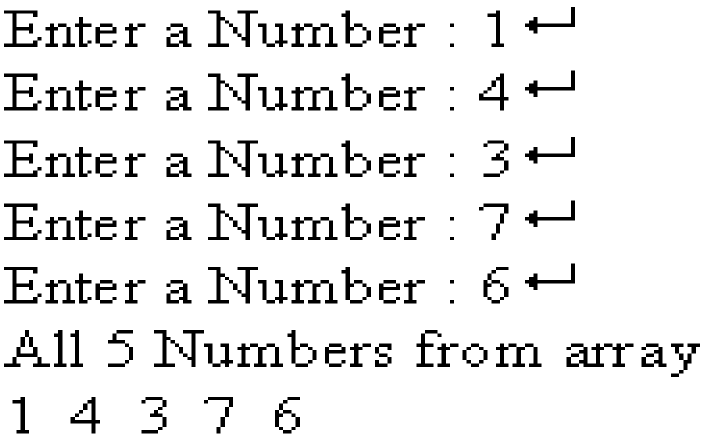
When a class gets instantiated and allocates memory dynamically to the pointer data member of an object.

Syntax:

```
class class_name
{
public:
   class_name (data_type size)
  {
        Pointer_data_member = new data_type [size];
   }
};
```

```
class Dynamic
{
    int *a;
public:
// dynamic default or automatic constructor
Dynamic( )
{
    a = new int [5];
}
void get ( );
void show ( );
};

void Dynamic :: get( )
{
for (int i=0; i<5;i++)
{
cout<<"Enter a Number :";
cin>>a[i];
}
}
```

```
void Dynamic::show ( )
{
cout<<"All 5 numbers from array\n";
for (int i=0; i<5; i++)
    cout<<a[i]<<"";
delete a;  // to free the allocatedmemory
}

void main ( )
{
Dynamic AB;      // invokes default
                 // dynamic constructor
AB.get ( );
AB.show ( );
}
```

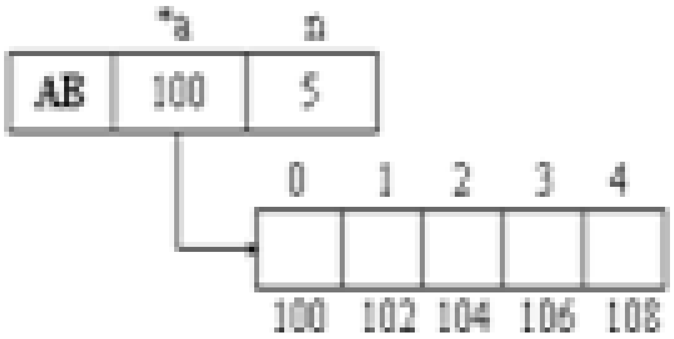


```
Enter a Number : 1 ↵
Enter a Number : 4 ↵
Enter a Number : 3 ↵
Enter a Number : 7 ↵
Enter a Number : 6 ↵
All 5 Numbers from array
1  4  3  7  6
```

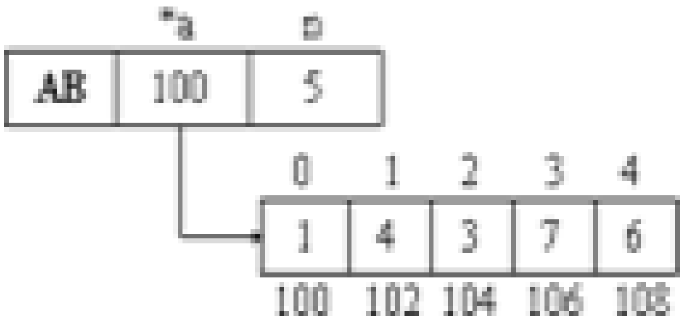
```
class Dynamic
{
      int *a, n;
public:
// dynamic parameterized constructor
Dynamic (int b)
{
   n = b;
   a = new int[n];
}
void get ( );
void show ( );
};

void Dynamic :: get ()
{
for (int i=0; i<n;i++)
{
cout<<"Enter a Number :";
cin>>a[i];
}
}
```

```
void Dynamic :: show ( )
{
cout<<"All "<<n<<" numbers from array\n";
for (int i=0; i<n; i++)
     cout<<a[i]<<"";

delete a;   // to free the allocated memory
}

void main ( )
{
int n;
cout<<"Enter size of the array : ";
cin>>n;
// invokes parameterized dynamic constructor
Dynamic AB (n);
AB.get ( );
AB.show ( );
}
```

```
Enter size of the array : 5 ↵
```



```
Enter a Number : 1 ↵
Enter a Number : 4 ↵
Enter a Number : 3 ↵
Enter a Number : 7 ↵
Enter a Number : 6 ↵
All 5 Numbers from array
1  4  3  7  6
```



Destructor

- It is also a special member function whose name is the class itself and its declaration starts with the operator **tilde (~)**.

- It invokes automatically as soon as the object leaves from the locality where it was created earlier.

- A destructor has no argument and no return value. So a destructor never be overloaded i.e. a class is having only one destructor but can have any number of constructors.

Syntax:

```
class class_name
{
private:
   // data member(s);
public:
    ~ class_name ( )
      {
         // statement(s);
       }
};
```

Example:

```
class ARS
{
public:
ARS ( )
{
    cout<<"Constructor Invoked\n";
}
~ARS ( )
 {
   cout<<"Destructor Invoked\n";
  }
};
void main ()
{
   ARS A, B;
}
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

```
int count;  // global variable with initial value zero (0)
class ARS
{
public:
ARS( )
{
count++;
cout<<”Object “<<count<<” Invoked Constructor\n”;
}
~ARS( )
{
cout<<”Object = “<<count<<” Invoked Destructor\n;
count--;
}
};

void main( )
{
ARS A, B, C;
}
```

```
Output:
Object 1 Invoked Constructor
Object 2 Invoked Constructor
Object 3 Invoked Constructor
Object 3 Invoked Destructor
Object 2 Invoked Destructor
Object 1 Invoked Destructor
```

Static Members

A member is said to be static when the member is declared as static. There are two types of static members.

1) Static data member

- The static data member’s declaration is given inside the class declaration by prefixing the keyword static.

- Its definition is given outside the class with a qualifier i.e. data type followed by class name and scope resolution operator (::).

- These static members are part of a class declaration and it is common to every object of the class.

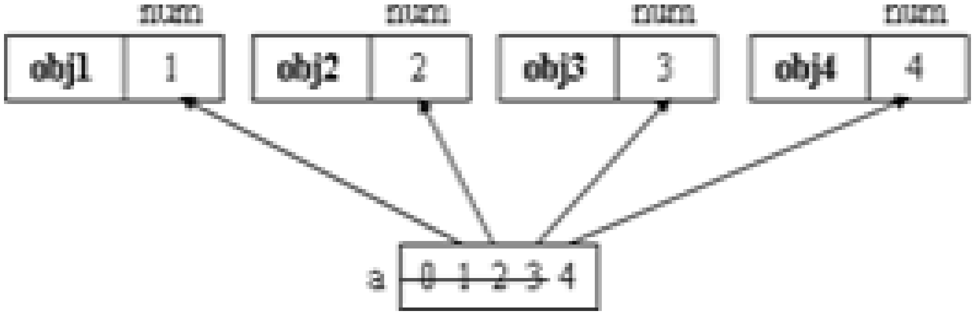
- It is also known as class variable.

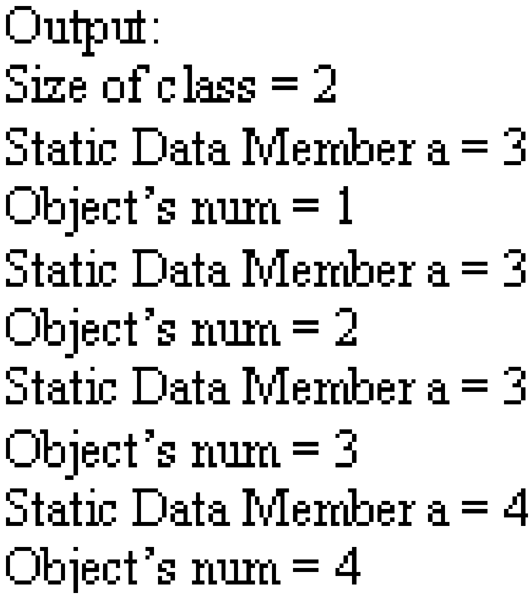
Syntax:

```
class class_name
{
private:
    static data_type member1, member2,…….;
public:
// member_function(s);
};

data_type class_name::member1 = value;
```

```
 // size of class is 2 bytes (only considers data members, not static member)
class STATIC
{
    int num;
   static int a;        // static data member declaration
public:
STATIC ( ) // default constructor
{
num = ++a;
}
void display( )
{
cout<<"Static Data Member a = "<< a << endl;
cout<<"Object‟s num = "<< num << endl;
}
};
```

```
int STATIC::a;     // static data member definition (default value is 0)
void main ( )
{
cout<<"Size of class = "<<sizeof (STATIC)<<endl;
STATIC obj1, obj2, obj3;
obj1.display ( );
obj2.display ( );
obj3.display ( );
STATIC obj4;
obj4.display ( );
}
```





```
Output:
Size of class = 2
Static Data Member a = 3
Object's num = 1
Static Data Member a = 3
Object's num = 2
Static Data Member a = 3
Object's num = 3
Static Data Member a = 4
Object's num = 4
```

Static Member Functions

- The static member functions declaration is prefixing by the keyword static.
- The static member function can have access to only other static members declared in the same class.
- The static member function can be called using the class name instead of its objects.
- It is also known as class function.

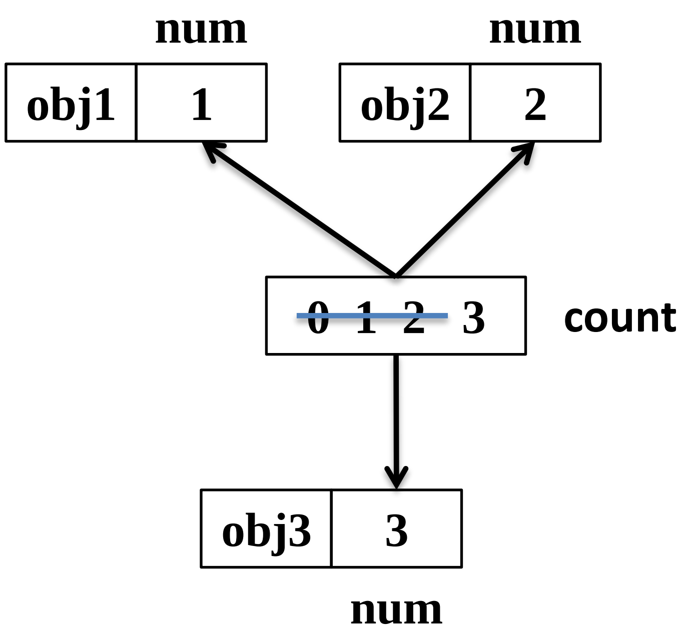
Syntax:

```
class class_name
{
public:
static return_type member_function (datatype1 var1,……if any)
{
      // statement (s);
}
};
void main ( )
{
    class_name :: member_function (Argument(s), if any);
}
```

Example:

```
class static_function
{
    int num;
    static int count;            // static data member declaration
public:
     void Assign ( )
    {
      num = ++count;
     }
     void Display( )
    {
     cout<<num<<" Object is Created\n";
     }
   static void Total( )
   {
     cout<<”Total Objects : “<<count<<endl;
    }
};

int static_function::count;
```

```
void main ( )
{
static_function obj1, obj2;
obj1.Assign( );
obj2.Assign( );
obj1.Display( );
obj2.Display( );
static_function::Total( );

static_function obj3;
obj3.Assign( );
obj3.Display( );
static_function::Total( );
}
```



Output:

```
1 Object is Created
2 Object is Created
Total Objects : 2
3 Object is Created
Total Objects : 3
```

Inheritance

- The technique of building new classes from the existing classes is called inheritance.

- The derived class inherits all the capabilities of the base class and can add refinements and extensions of its own.

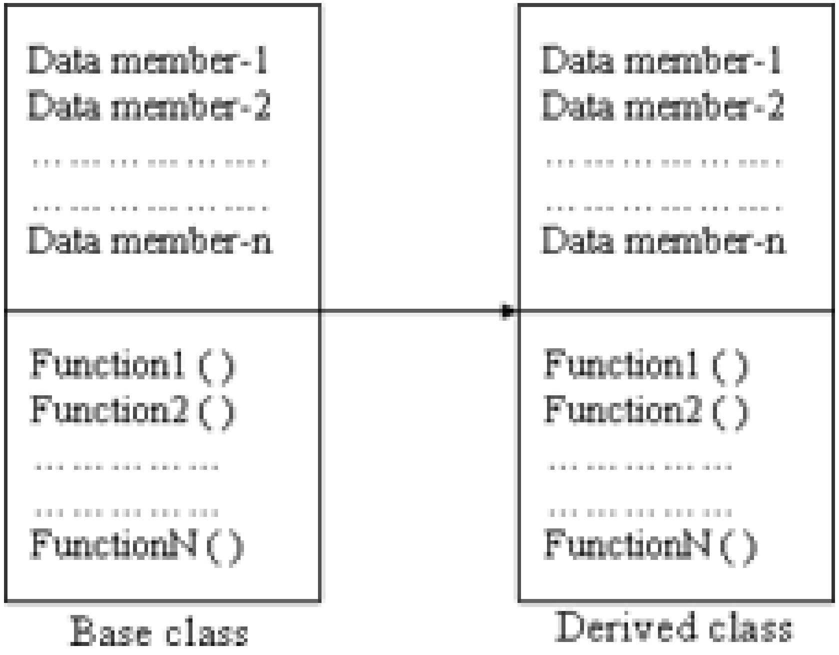
- There are six types of inheritances.
  1. **Single**
  2. **Multiple**
  3. **Multilevel**
  4. **Hierarchical**
  5. **Multipath**
  6. **Hybrid**

1. Single Inheritance

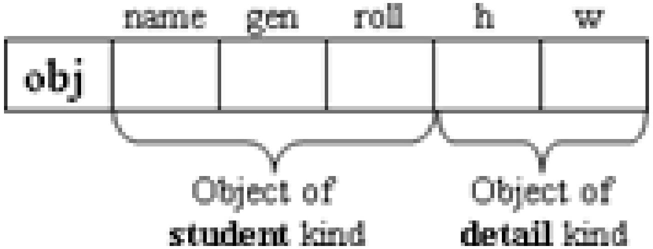
Deriving a new class from an existing class is known as single inheritance.

Syntax:

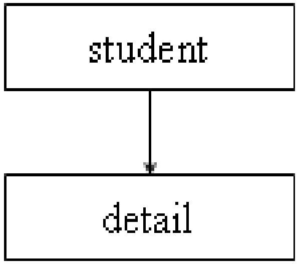
```
class base
{
     protected:
         // data member(s);
     public:
        // member function(s);
};

class derived : public base
{
    private:
        // data member(s);
    public:
        // member function(s);
};
```



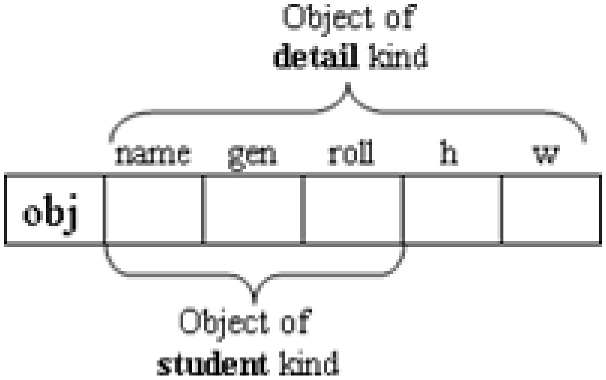
```
// base class declaration
classs student
{
protected:
    char name [20], gen[7];
    int roll;
public:
void accept ( )
{
cout << “Enter name, roll, and gender :”;
cin >> name>> roll>>gen;
}
};
```



```
//derived class declaration
class detail : public student
{
float h, w;
public:
void get ( )
{
cout << “ Enter Height and weight :”;
cin >> h >> w;
}
void out ( )
{
cout <<name << “ “ << roll << “ “ << gen<<” “
<< h << “ “ << w << endl;
}
};
void main ( )
{
detail obj;    //object of derived class kind
obj.accept ();
obj.get ();
obj.out ();
}
```



```
class student           // base class declaration
{
private:
char name [20], gen[7];
int roll;
public:
void accept ( )
{
cout << “Enter name, roll, and gender :”;
cin >> name>> roll>>gen;
}
void display ( )
{
cout <<name << “ “ << roll << “ “ << gen<<” “;
}
};
```



```
//derived class declaration
class detail :public student
{   float h, w;
public:
void accept ( )
{
student::accept ( );
cout << “ Enter Height and weight :”;
cin >> h >> w;
}
void display ( )
{
student::display ( );
cout << h << “ “ << w << endl;
}
};
void main ( )
{  //object of derived class kind
detail obj;
// size of obj = 20 + 7 + 2 + 4 + 4 = 37 bytes
obj.accept ( );
obj.display ( );
}
```

2. Multiple Inheritances

A new class is to be derived or created from more than one base or existing classes.

Syntax:

```
class base1
{
protected:
//data member (s);
public:
//member function (s);
};
class base2
{
protected:
//data member (s);
public:
//member function (s);
};
……………………………
……………………………
```

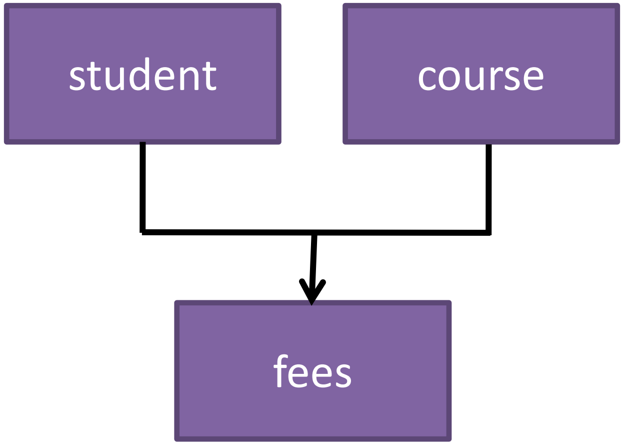
```
class baseN
{
protected:
//data member (s);
public:
//member function (s);
};

class derived:public base1, public base2,….., public baseN
{
private:
//data member;
public:
//member function;
};
```

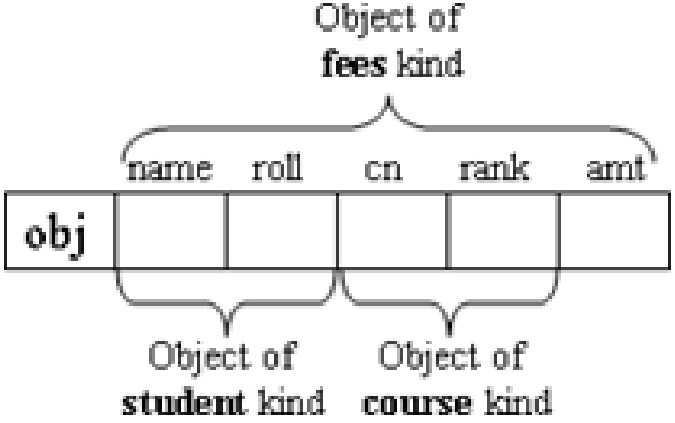
```
class student  // base class
{
protected :
   char name [20];
   int roll;
public :
void accept ( )
{
cout << “Enter name:”
cin >> name ;
cout << “ Enter Roll number:”
cin >> roll;
}
};
```

```
class course // base class
{
protected:
   char cn [20];
   int rank;
public:
void accept ( )
{
cout<<“Enter name of the course”;
cin >> cn;
cout << “Enter rank:”
cin >> rank;
}
};
```

```
 // derived class
class fees : public student, course
{
    float amt;
public :
void accept ( )
{
student :: accept ( );
course :: accept ( );
cout << “Enter the course fees:”;
cin >> amt;
}
void display ( )
{
cout<<“Name :”<< name<<endl;
cout<<“Roll :”<< roll<<endl;
cout<<“Course Name :”<< cn<<endl;
cout<<“Rank :”<< rank<<endl;
cout << “Course Fees :” << amt;
}
};
```

```
void main ( )
{
fees obj;
obj.accept ();
cout << “student Details…\n”;
obj.display ();
}
```



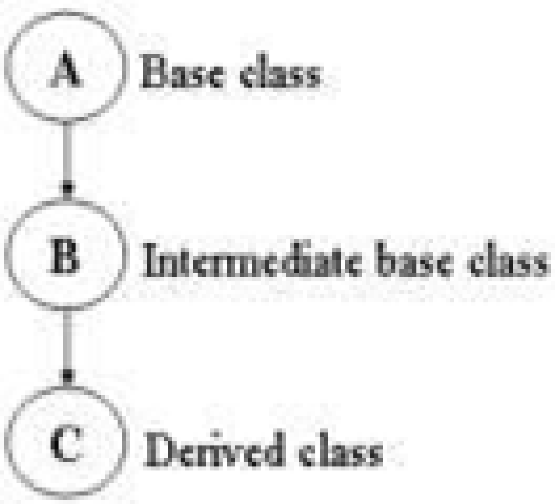


3. Multilevel Inheritances

Derivation of a class from another derived class is called multilevel inheritance. It is a chain process. The top most class is called **base class**, the bottom most class is called **derived class**. In between the top and bottom class, the remaining classes are called **intermediate base classes**.

Syntax:

```
class A
{
Statement(s);
};

class B : public A
{
Statement(s);
};

class C : public B
{
Statement(s);
};
```

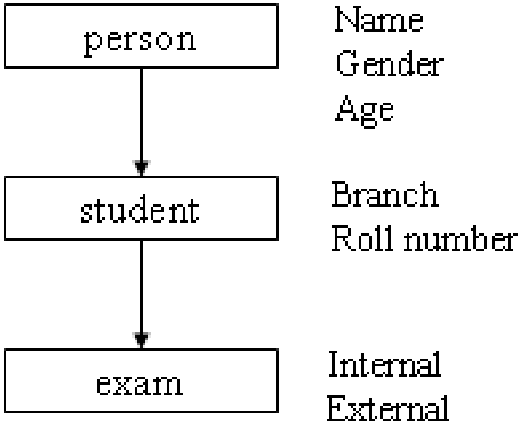


```
class person
{
protected:
    char n[20], gen[6];
    int age;
public:
void Get ( )
{
cout<<”Enter Name, Gender, and Age : “;
cin>>n>>gen>>age;
}
};
```

```
class student : public person
{
protected:
   char b[15];
    int roll;
public:
void Get ( )
{
person::Get ( );
cout<<”Enter Branch and Roll : “;
cin>>b>>roll;
}
};
```

```
class exam : public student
{
private:
   int m1, m2;
public:
void Get ( )
{
student::Get ( );
cout<<”Enter Internal and External Marks : “;
cin>>m1>>m2;
}
void Out ( )
{
cout<<”Name :”<<n<<endl;
cout<<”Gender :”<<gen<<endl;
cout<<”Age :”<<age<<endl;
cout<<”Branch :”<<b<<endl;
cout<<”Roll :”<<roll<<endl;
cout<<”Internal Marks :”<<m1;
cout<<”\nExternal Marks :”<<m2<<endl;
}
};
```

```
void main ( )
{
exam obj;
obj.Get ( );
obj.Out ( );
}
```

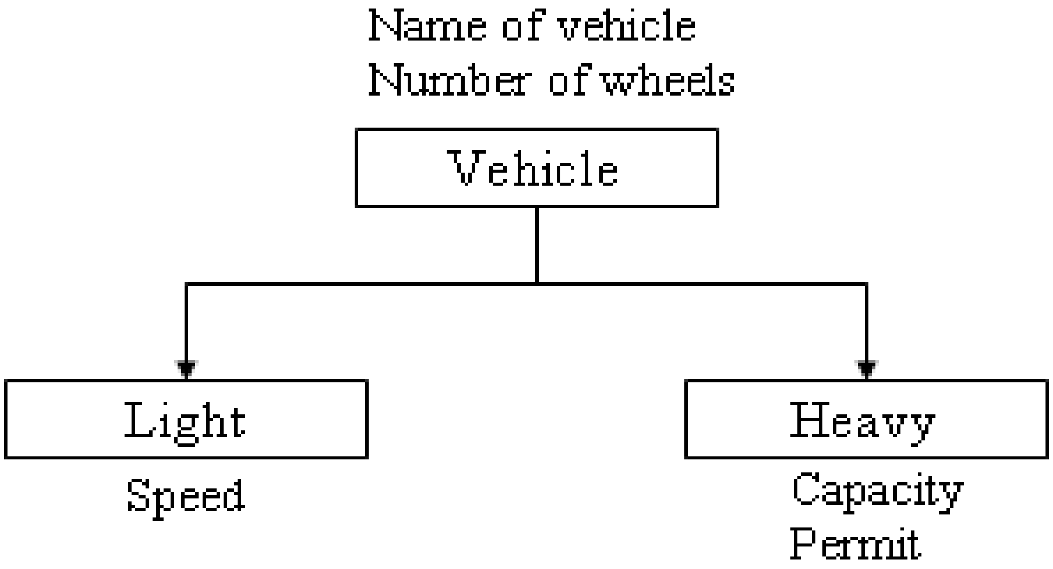


4. Hierarchical Inheritances

The super class (base class) includes the features that are common to all the sub-classes (derived classes).

A sub-class is created by inheriting the properties of the base class and adding some of its own features.

The sub-class can serve as the super class of the lower level classes again and so on.



5. Multipath Inheritances

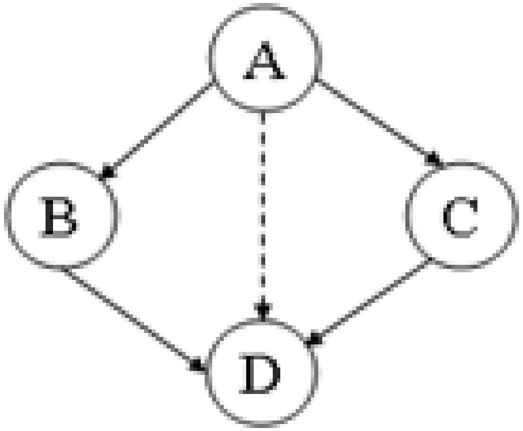
It is the form of inheritance which derives a new class by multiple inheritances of base classes, which are derived earlier from the same base class.

Syntax:

```
class A
{
Statement(s);
};

class B : public virtual A
{
Statement(s);
};

class C : virtual public A
{
Statement(s);
};
```

```
class D : public B, public C  // derived class
{
Statement(s);
};
```

Class B and **C** are referred as **direct base** classes.
Class A is referred as **indirect base** class.



The public and protected members of **Class A** are inherited into the child **class D** twice, first via **B class** and then via **C class**.

Therefore the child **class D** would have duplicate sets of members of **Class A** which leads to ***ambiguity*** during compilation.

Virtual Base Class

It is achieved by making the common base class as a virtual base class while declaring the direct base classes.

So that it inherits indirect base class as virtual to the direct base classes.

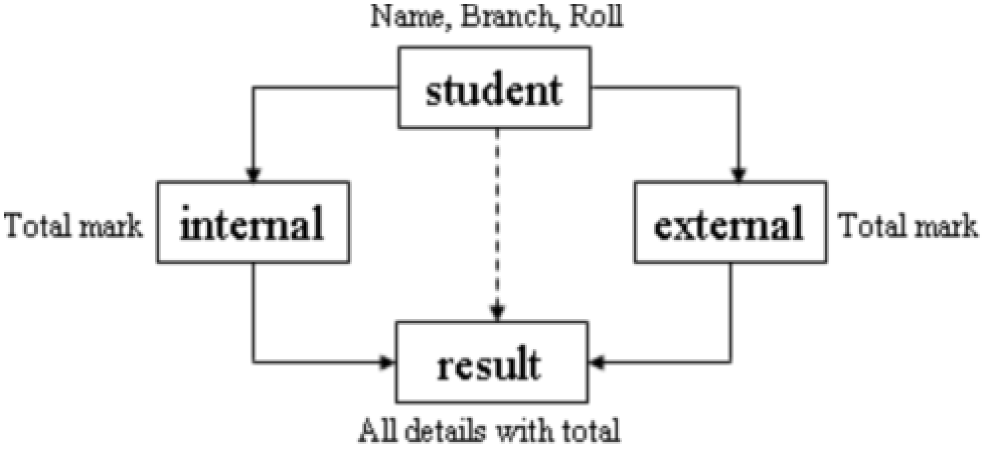
Hence the properties of indirect base classes are inherited into the derived class only once through the virtual path.

```
class student          // indirect base class
{
protected:
   char n[20], b[10];
   int roll;
public:
   void Get ( )
   {
     cout<<"Enter Name, Branch, and Roll : ";
     cin>>n>>b>>roll;
    }
};
// direct base class
class internal : public virtual student
{
protected:
  int imark;
public:
  void Get ( )
  {
    cout<<"Enter Internal Mark : ";
    cin>>imark;
   }
};
```

```
// direct base class
class external : public virtual student
{
protected:
   int emark;
public:
void Get ( )
{
cout<<"Enter External Mark : ";
cin>>emark;
}
};
```

```
class result : public internal, external
{
     int tot;
public:
void Get ( )
{
// invoking of virtual base class member
// function which avoids ambiguity
student :: Get ( );
internal :: Get ();
external :: Get ();
}

void Out ( )
{
tot = imark + emark;
cout<<"Name : "<<n<<endl;
cout<<"Branch : "<<b<<endl;
cout<<"Roll : "<<roll<<endl;
cout<<"Total Marks : "<<tot<<endl;
}
};
```

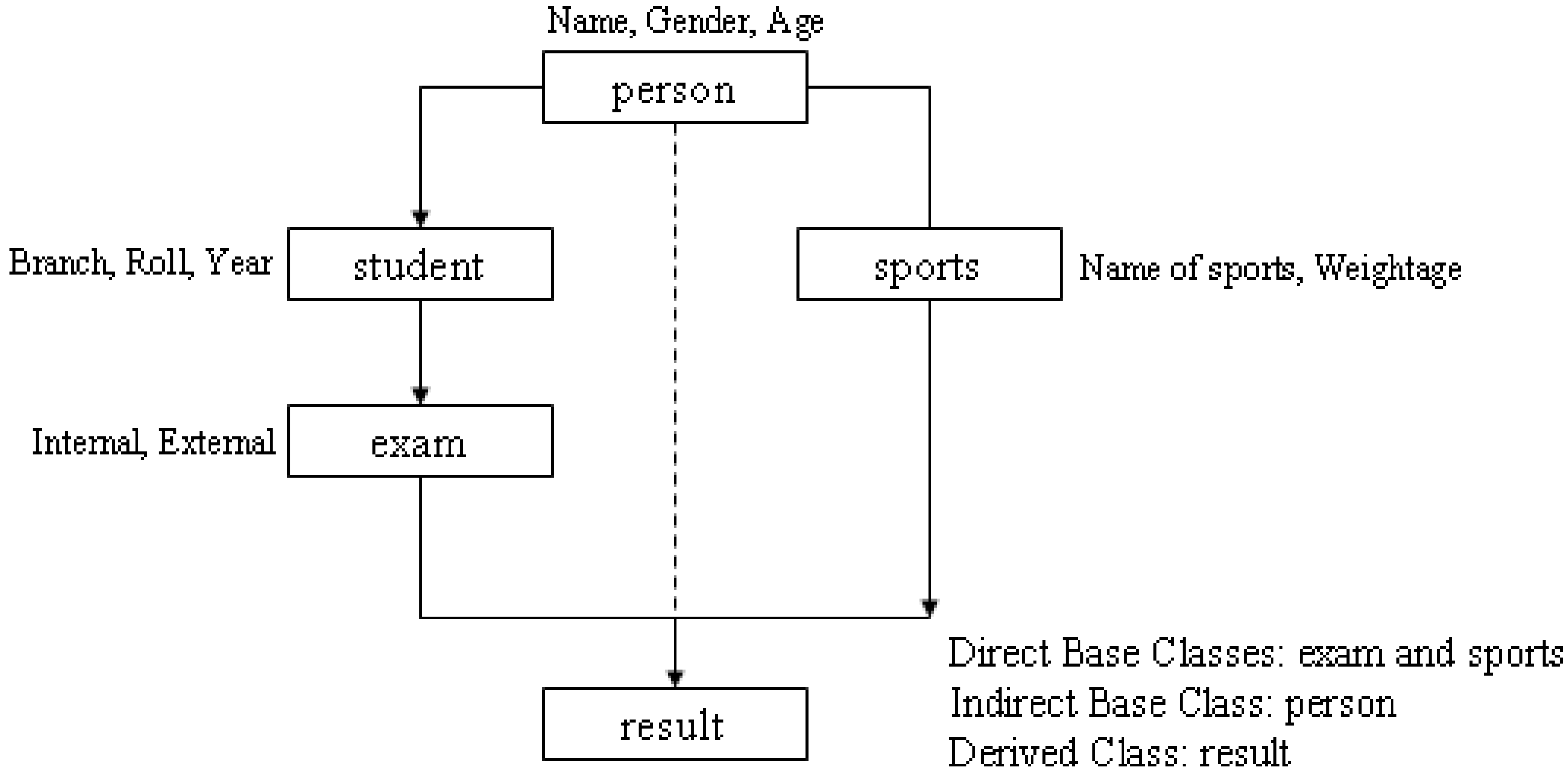
```
void main ( )
{
result obj;
obj.Get ();
obj.Out ();
}
```



Indirect Base Class: student
Direct Base Classes: internal and external

6. Hybrid Inheritances

Hybrid means combination of multiple and multilevel inheritances together.



Direct Base Classes: exam and sports
Indirect Base Class: person
Derived Class: result

Containership or Container Classes or Delegation or Has-a-relationship:

When an object of a class is declared as a data member of another class is known as container class.

Syntax:

```
class class1
{
……..
……..
};
class class2
{
……..
……..
};
```

```
……….
……….
……….
class classN
{
……..
……..
};
```

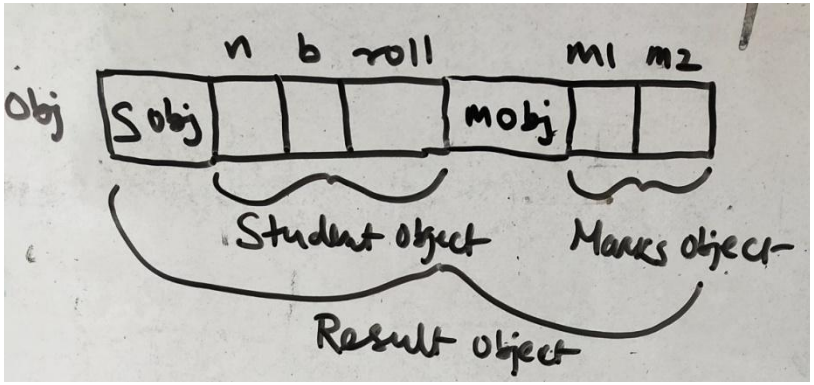
```
class container_class
{
private:
    class1 object1,……………;
    class2 object1,……………;
    ……………………….
    classN object1,………....;
public:
………..
………..
};
```

```
class Student
{
private:
    char n[20], b[10];
    int roll;
public:
void Get ( )
{
cout<<”Enter Name, Branch, and Roll : “;
cin>>n>>b>>roll;
}
void Out ( )
{
cout<<n<<” “<<b<<” “<<roll<<endl;
}
};
```

```
class Marks
{
private:
        int m1, m2;
public:
void Get ( )
{
cout<<”Enter Internal and External Marks : “;
cin>>m1>>m2;
}
void Out ( )
{
cout<<m1<<” “<<m2<<endl;
}
};
```

```
class Result   // container class or containership
{
private:
   Student sobj;   // object of Student kind
   Marks mobj;   // object of Marks kind
public:
void Get ( )
{
sobj.Get ( );
mobj.Get ( );
}
void Out ( )
{
sobj.Out ( );
mobj.Out ( );
}
};
```

```
void main ( )
{
Result obj; // object of container class
obj.Get ();
obj.Out ();
}
```



Constructors in Inheritances

I) Constructors in Single Inheritance

i) Default Constructor in Single Inheritance

```
class Base
{
public:
Base( )
{
cout<<”Base Class Constructor\n”;
}
};
class Derived : public Base
{
public:
Derived( )
{
cout<<”Derived Class Constructor\n”;
}
};
void main()
{
Derived obj;
}
```

Output:

```
Base Class Constructor
Derived Class Constructor
```

ii) Parameterized Constructor in Single Inheritance

```
class Base
{
public:
Base ( )
{
cout<<”Base Class Constructor\n”;
}
};
class Derived : public Base
{
Derived (char *a)
{
cout<<a<<endl;
}
};
void main ( )
{
Derived obj(”Parameterized Constructor in Derived Class”);
}
```

Output:

```
Base Class Constructor
Parameterized Constructor Derived Class
```

iii) Parameterized Constructor in both Base and Derived class Single Inheritance

```
class Base
{
public:
Base ( )
{
}
Base (int a)
{
cout<<"Parameterized Base Class Constructor\n";
}
};
class Derived : public Base
{
public:
Derived (int a)
{
cout<<"Parameterized Constructor in Derived Class\n";
}
};
void main ( )
{
Derived obj(7);
}
```

Output:

```
Parameterized Constructor Derived Class
```

The object **obj** cannot invoke base class parameterized constructor. If the default constructor is removed from the base class, then error occurs.

iv) Parameterized Constructor in both Base and Derived class Single Inheritance

```
#include<iostream.h>
class Base
{
public:
Base (int a)
{
cout<<”Base a = ”<<a<<endl;
}
};
class Derived : public Base
{
public:
Derived (int b) : Base(b) // To invoke the parameterized base constructor
{
cout<<”Derived b = ”<<b<<endl;
}
};
void main ( )
{
Derived obj(7);
}
```

Output:

```
Base a = 7
Derived b = 7
```

II) Constructors in Multiple Inheritance

i) Default Constructor in Multiple Inheritance

```
class Base1
{
public:
Base1 ( )
{
cout<<”Base1 Class Constructor\n”;
}
};
class Base2
{
public:
Base2 ( )
{
cout<<”Base2 Class Constructor\n”;
}
};
```

```
class Derived : public Base1, Base2
{
public:
Derived ( )
{
cout<<”Derived Class Constructor\n”;
}
};
void main ()
{
Derived obj;
}
```

Output:

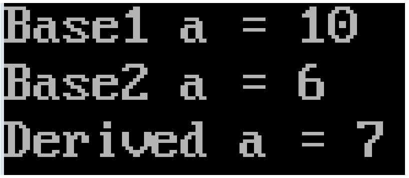
```
Base1 Class Constructor
Base2 Class Constructor
Derived Class Constructor
```

II) Constructors in Multiple Inheritance

i) Parameterized Constructor in Multiple Inheritance

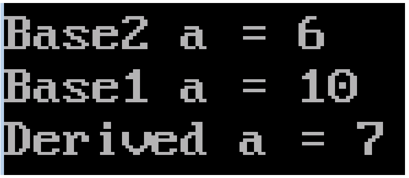
```
class Base1
{
    int a;
public:
Base1(int x)
{
   a = x;
   cout<<”Base1 a = ”<<a<<endl;
}
};
class Base2
{
  int a;
public:
Base2(int x)
{
   a = x;
   cout<<”Base2 a = ”<<a<<endl;
}
};
```

```
class Derived : public Base1, Base2   // case-1
// class Derived : public Base2, Base1 // case-2
{
   int a;
public:
Derived(int a, int b, int c):Base1(a), Base2(b)
{
 a = c;
 cout<<”Derived a = ”<<a<<endl;
}
};
void main ()
{
Derived obj(10, 6, 7);
}
```



```
Base1 a = 10
Base2 a = 6
Derived a = 7
```

Case-1



```
Base2 a = 6
Base1 a = 10
Derived a = 7
```

Case-2

III) Constructors in Multilevel Inheritance

i) Parameterized Constructor in Multilevel Inheritance

```
class Base
{
   int a;
public:
Base (int x)
{
  a = x;
  cout<<”Base a = ”<<a<<endl;
}
};
class Intermediate : public Base
{
    int b;
public:
Intermediate ( int p, int q) : Base(p)
{
  b = q;
  cout<<”Intermediate b = ”<<b<<endl;
}
};
```

```
class Derived : public Intermediate
{
  int c;
public:
Derived (int p, int q, int r): Intermediate (p, q)
{
cout<<”Derived c = ”<<c<<endl;
}
};
void main ( )
{
   Derived obj(5, 6, 7);
}
```

Output:

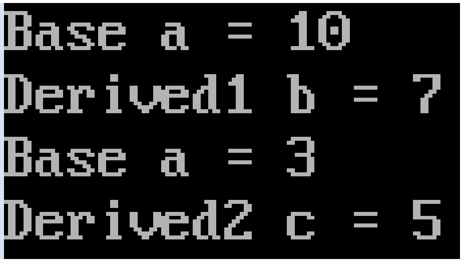
```
Base a = 5
Intermediate b = 6
Derived c = 7
```

IV) Constructors in Hierarchical Inheritance

i) Parameterized Constructor in Hierarchical Inheritance

```
class Base
{
    int a;
public:
Base(int x)
{
   a = x;
   cout<<"Base a = "<<a<<endl;
}
};

class Derived1:public Base
{
    int b;
public:
Derived1(int x, int y):Base(x)
{
   b = y;
   cout<<"Derived1 b = "<<b<<endl;
}
};
```

```
class Derived2:public Base
{
    int c;
public:
Derived2(int p, int q):Base(p)
{
   c = q;
   cout<<"Derived2 c = "<<c<<endl;
}
};

void main()
{
   Derived1 obj1(10, 7);
   Derived2 obj2(3, 5);
}
```



```
Base a = 10
Derived1 b = 7
Base a = 3
Derived2 c = 5
```

i) Destructors in Single Inheritances

```
class Base
{
public:
Base ( )
{
cout<<”Base Constructor\n”;
}
~Base ( )
{
cout<<”Base Destructor\n”;
}
};
```

```
class Derived : public Base
{
public:
Derived ( )
{
cout<<” Derived Constructor\n”;
}
~ Derived ( )
{
cout<<” Derived Destructor\n”;
}
};
void main ()
{
Derived obj;
}
```

Output:

```
Base Constructor
Derived Constructor
Derived Destructor
Base Destructor
```

ii) Destructors in Multiple Inheritances

```
class Base1
{
public:
Base1 ( )
{
cout<<”Base1 Constructor\n”;
}
~Base1 ( )
{
cout<<”Base1 Destructor\n”;
}
};
```

```
class Base2
{
public:
Base2( )
{
cout<<”Base2 Constructor\n”;
}
~Base2( )
{
cout<<”Base2 Destructor\n”;
}
};
```

```
class Derived : public Base1, Base2
{
public:
Derived( )
{
cout<<” Derived Constructor\n”;
}
~ Derived( )
{
cout<<” Derived Destructor\n”;
}
};
```

```
void main()
{
Derived obj;
}
```

Output:

```
Base1 Constructor
Base2 Constructor
Derived Constructor
Derived Destructor
Base2 Destructor
Base1 Destructor
```

iii) Destructors in Multilevel Inheritances

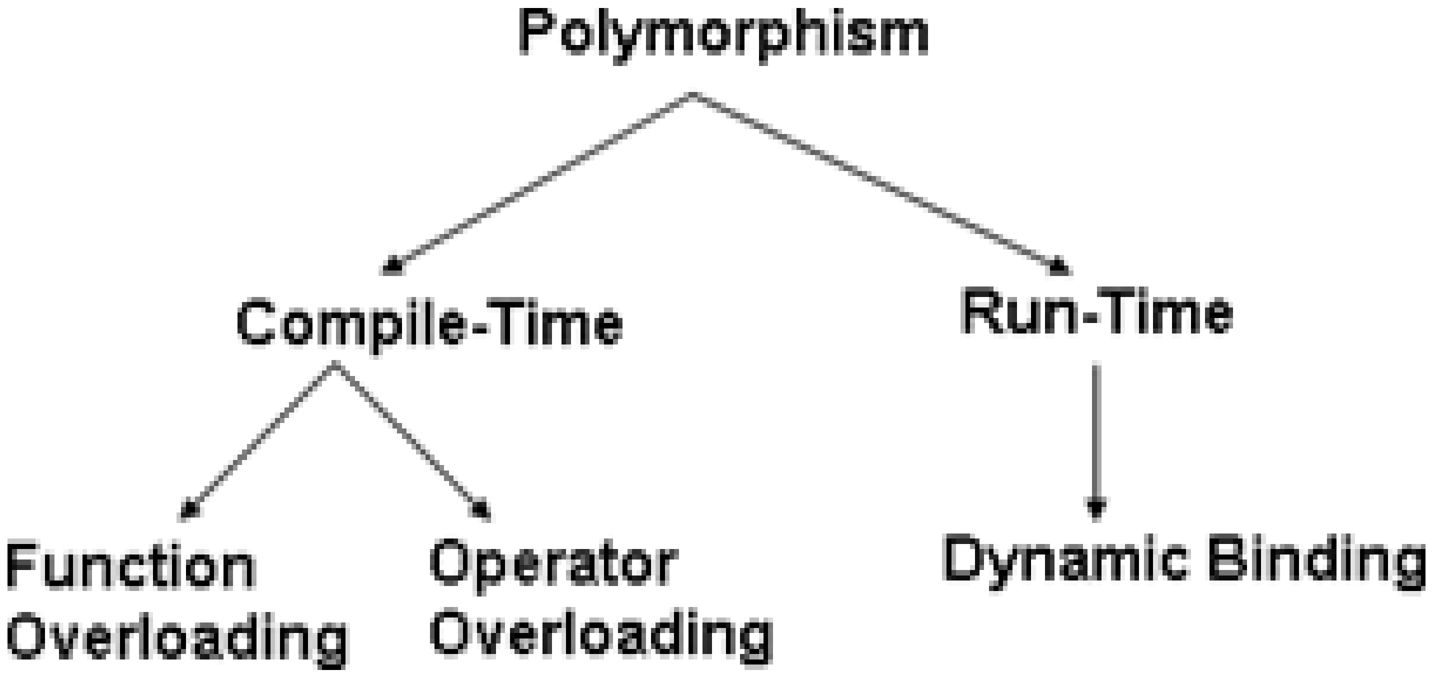
```
class Base
{
public:
Base ( )
{
cout<<”Base Constructor\n”;
}
~Base ( )
{
cout<<”Base Destructor\n”;
}
};
class Intermediate : public Base
{
public:
Intermediate ( )
{
cout<<” Intermediate Constructor\n”;
}
~ Intermediate ( )
{
cout<<” Intermediate Destructor\n”;
}
};
class Derived : public Intermediate
{
public:
Derived ( )
{
cout<<” Derived Constructor\n”;
}
~ Derived ( )
{
cout<<” Derived Destructor\n”;
}
};
void main ()
{
Derive dobj;
}
```

Output:

```
Base Constructor
Intermediate Constructor
Derived Constructor
Derived Destructor
Intermediate Destructor
Base Destructor
```

Polymorphism

It allows a single name / operator to be associated with different operations depending on the type of data passed to it.



Function Overloading (Static Binding)

Defining a function more than one time with same name and different signatures. (Signature means argument and return value type)

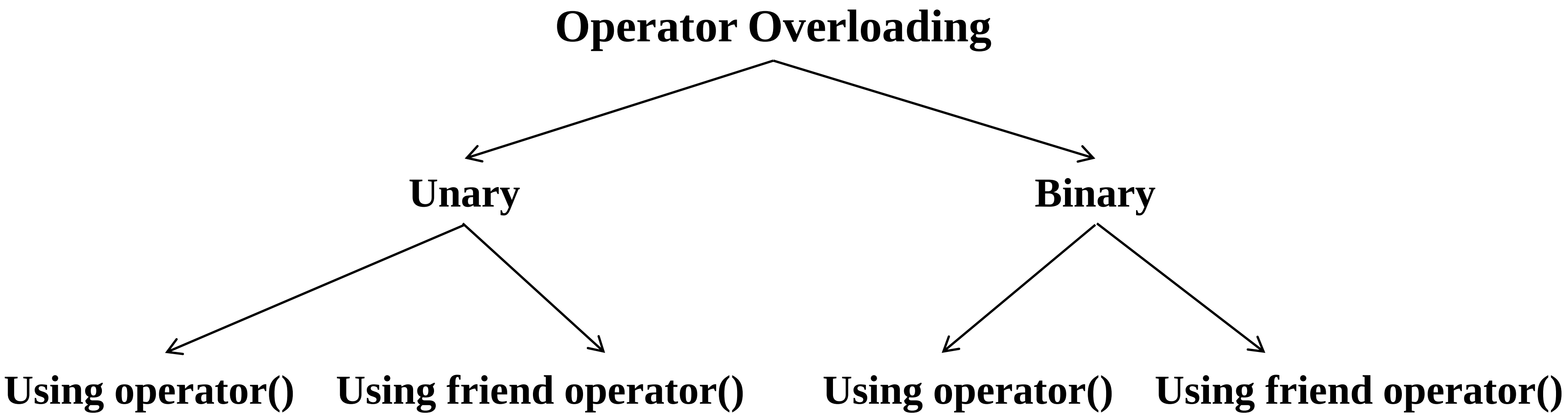
EXAMPLE-1

```
// Finding the absolute value of an integer and float.
int ABS (int a)
{
   return (abs(a));
}
float ABS (float b)
{
   return (fabs (b));
}
void main ( )
{
int a = -7, res1 = ABS (a);
float b = -66.77, res2 = ABS (b);
cout << “Absolute value of ” <<a<< “=”<<res1<< endl;
cout << “Absolute value of ”<<b<< “=” <<res2;
}
```

EXAMPLE-2

```
// Finding the area of square, rectangle, and triangle.
float Area (float s)
{
   return (s*s);
}
float Area (float a, float b)
{
   return (a*b);
}
float Area (float s1, float s2, float s3)
{
    float s = (s1+s2+s3)/2;
    float ar = sqrt(s*(s-s1)*(s-s2)*(s-s3));
    return (ar);
}
```

```
void main()
{
   float side, len, brd;
cout<<“Enter the side of a square: “;
cin>>side;
float sarea = Area(side);
cout<<“Enter the length and breadth of a rectangle: “;
cin>>len>>brd;
float rarea = Area(len, brd);
cout<<“Enter the three sides of a triangle: “;
cin>>a>>b>>c;
float tarea = Area(a, b, c);

cout<<“Area of Square = “<<sarea<<endl;
cout<<“Area of Rectangle = “<<rarea<<endl;
cout<<“Area of Triangle = “<<tarea<<endl;
}
```



Operator Overloading

Unary

Binary

Using operator()

Using friend operator()

Using operator()

Using friend operator()

Operator Overloading (Static Binding)

To use the existing operators in the member function of a class.

So that the resulting operator with objects of its class is used its operands is called operator overloading.

Operators cannot be overloaded with operator ( ) member function

1. Member or member access operator (.)
2. Pointer to member operator (. *)
3. Scope resolution operator(::)
4. Ternary operator (?:)
5. sizeof( ) operator
6. Pre-processor symbols (# and # #)

Operators cannot be overloaded with friend operator ( ) function

1) Parentheses (**( )**)
2) Square Brackets (**[ ]**)
3) Right Arrow Operator (→)
4) Assignment Operator (**=**)

I) Unary Operator Overloading Using operator ( ) Member Function

The operator ( ) has no argument and no return value.

Syntax:

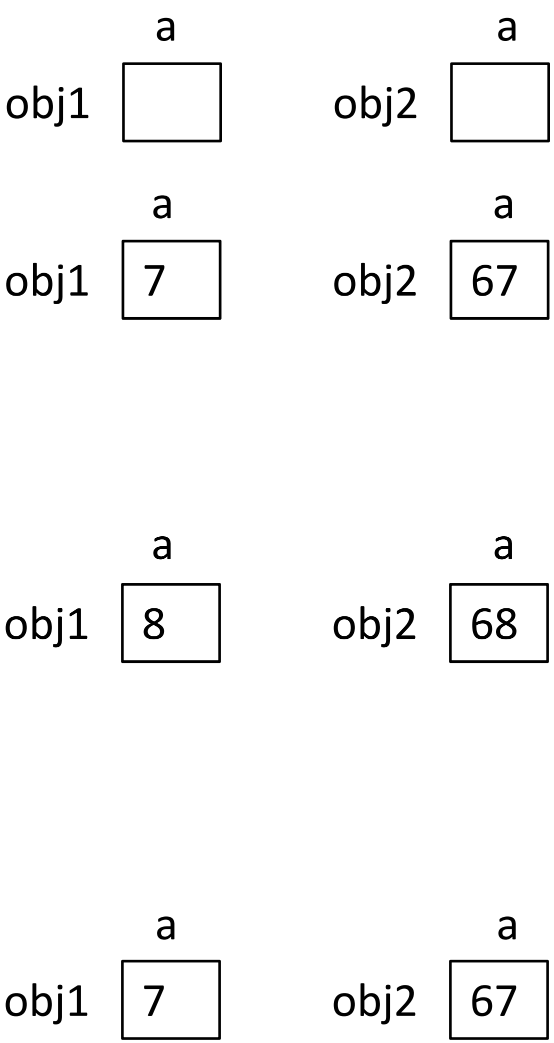
```
class class_name
{
private:
    // data member(s);
public:
void operator operator_to_be_overloaded ( )   // pre kind
{
    // statement(s);
}
void operator operator_to_be_overloaded (int)   // post kind
{
   // statement(s);
}
};
```

Example:

```
class IncDec
{
    int a;
public:
void Assign (int b)
{
a = b;
}
void Show ( )
{
cout<<"A="<<a<<endl;
}
void operator ++( ) // pre-increment
{
++a;
}
void operator ++(int) // post-increment
{
a++;
}
```

```
void operator --( ) // pre-decrement
{
   --a;
}
void operator --(int) // post-decrement
{
   a--;
}
};
```

```
void main ( )
{
IncDec obj1, obj2;

obj1.Assign(7);
obj2.Assign(67);

obj1.Show ( );
obj2.Show ( );

++obj1;
obj2++;

obj1.Show ( );
obj2.Show ( );

obj1--;
--obj2;

obj1.Show ( );
obj2.Show ( );
}
```



a a
obj1 obj2

a a
obj1 7 obj2 67

a a
obj1 8 obj2 68

a a
obj1 7 obj2 67

```
Output:
A = 7
A = 67
A=8
A = 68
A = 7
A = 67
```

II) Unary Operator Overloading Using Friend operator( ) Function

The operator ( ) has one argument of class kind and may or may not have a return value.

The operator ( ) member function needs to be declared as friend kind in the class.

Syntax:

```
class class_name
{
private:
          // data member(s);
public:
friend return_type operator operator_to_be_overloaded (class_name object_name )
{
    // statement(s);
}
};
```

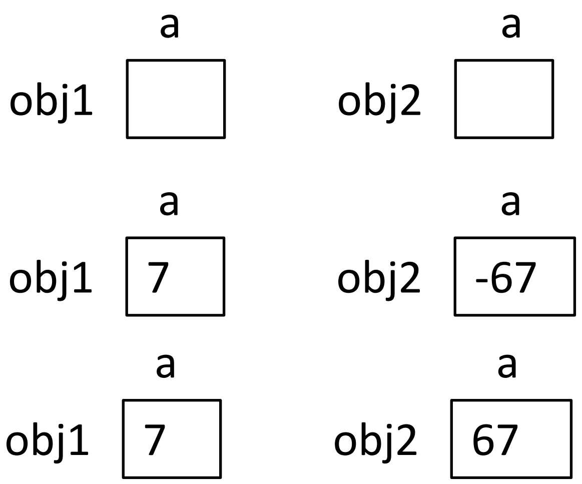
Example:

```
 class Negate
{
private:
            int a;
public:
void Assign (int b)
{
a = b;
}
void Show ( )
{
cout<<"A="<<a<<endl;
}
friend Negate operator -(Negate obj)
{
if(obj.a<0)
    obj.a=obj.a*-1;  // or obj.a = -obj.a;
return obj;
}
};
```

```
void main ( )
{
Negate obj1, obj2;
obj1.Assign (7);
obj2.Assign (-67);
obj1.Show ();
obj2.Show ( );
obj1 = -obj1;
obj2 = -obj2;
obj1.Show ( );
obj2.Show ();
}
```

Output:

```
A = 7
A = -67
A = 7
A = 67
```



```
class Negate
{
private:
          int a;
public:
void Assign (int b)
{
a = b;
}
void Show ( )
{
cout<<”A=”<<a<<endl;
}
friend void operator -(Negate &obj)
{
if(obj.a<0)
     obj.a=obj.a*-1;    // or obj.a =-obj.a;
}
};
```

```
void main ( )
{
Negate obj1, obj2;

obj1.Assign (7);
obj2.Assign (-67);

obj1.Show ();
obj2.Show ();

-obj1;
-obj2;

obj1.Show ( );
obj2.Show ();
}
```

III) Binary Operator Overloading Using operator ( ) Member Function

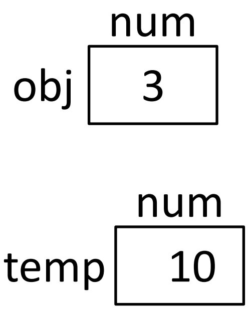
The operator ( ) function has one argument and may or may not have a return value.

Syntax:

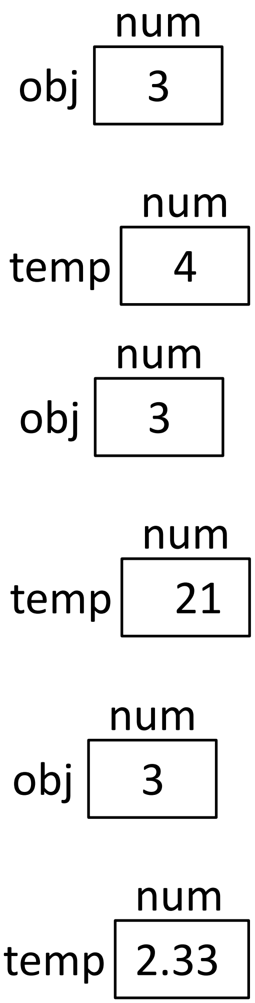
```
class class_name
{
private:
   // data member(s);

public:
return_type operator operator_to_be_overloaded (class_name object_name)
{
   // statemnt(s);
}
 };
```

Sum, Difference, Product, and Division of two numbers.

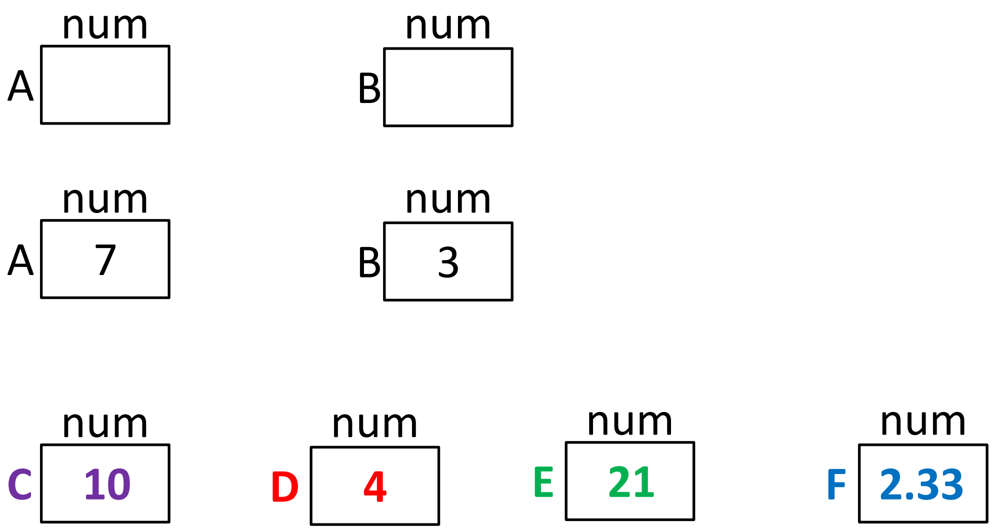
```
class OOL
{
    float num;
public:
void Assign (int a)
{
num = a;
}
void Show ( )
{
cout<<num<<endl;
}
OOL operator +(OOL obj)
{
    OOL temp;
temp.num = num + obj.num;
return temp;
}
```



```
OOL operator -(OOL obj)
{
    OOL temp;
temp.num = num - obj.num;
return temp;
}
OOL operator *(OOL obj)
{
    OOL temp;
temp.num = num * obj.num;
return temp;
}
OOL operator /(OOL obj)
{
    OOL temp;
temp.num = num / obj.num;
return temp;
}
};
```



```
void main ( )
{
OOL A, B;

A.Assign (7);
B.Assign (3);

OOL C = A + B;
OOL D = A - B;
OOL E = A * B;
OOL F = A / B;
cout<<”Two Numbers\n”;
A.Show ( );
B.Show ( );
cout<<”Sum = “;
C.Show ( );
cout<<”Difference = “;
D.Show ( );
cout<<”Product = “;
E.Show ( );
cout<<”Division = “;
F.Show ( );
}
```



Using operator overloading add and subtract two matrices of order 2X3.

```
class Matrix
{
      float mat[2][3];
public:
void Read ( );
void Display ( );
Matrix operator +(Matrix);
Matrix operator -(Matrix);
};
void Matrix::Read ( )
{
for (int i=0; i<2; i++)
{
for (int j=0; j<3; j++)
{
cout<<”Enter a Number :”;
cin>>mat[i][j];
}
}
}
```

```
void Matrix::Display ( )
{
for (int i=0; i<2; i++)
{
for (int j=0; j<3; j++)
cout<<mat[i][j]<<” “;
cout<<endl;
}
}
Matrix Matrix::operator +(Matrix obj)
{
Matrix temp;
for (int i=0; i<2; i++)
{
for (int j=0; j<3; j++)
{
temp.mat[i][j] = mat[i][j] + obj.mat[i][j];
}
}
return temp;
}
```

```
void main ( )
{
Matrix obj1, obj2;
cout<< ”Enter Numbers in 1stMatrix\n”;
obj1.Read ();
cout<< ”Enter Numbers in 2ndMatrix\n”;
obj2.Read ();
Matrix obj3 = obj1 + obj2;
Matrix obj4 = obj1 - obj2;
cout<<”1st Matrix\n”;
obj1.Display ( ) ;
cout<<”2nd Matrix\n”;
obj2.Display ( ) ;
cout<<”Matrix Addition\n”;
obj3.Display ( );
cout<<”Matrix Subtraction\n”;
obj4.Display ( );
}
```

Write a program which assigns a matrix to another. (Using operator (=) overloading)

```
class matrix
{
int mat[2][2];
public:
void read();
void operator = (matrix);
void show();
};
void matrix::read( )
{
for( int i=0, j; i<2; i++)
   for(j=0, j; j<2; j++)
{
cout<<“Enter a number…”;
cin>>mat[i][j];
}
}
```

```
void matrix::show( )
{
for( int i=0; i<2; i++)
{
for( int j=0; j<2; j++)
    cout<<mat[i][j]<<“ “;
cout<<endl;
}
}
void matrix::operator = (matrix obj)
{
for ( int i=0; i<2; i++)
{
for ( int j =0; j<2; j++)
     mat[ i ][ j ] = obj.mat[ i ][ j ];
}
}
```

```
void main()
{
matrix obj1, obj2;
obj1.read( );
obj2 = obj1;
cout<<“Actual Matrix\n”;
obj1.show( );
cout<<“Copied Matrix\n”;
obj2.show( );
}
```

IV) Binary Operator Overloading Using Friend operator( ) Function

- The **friend operator ( )** has two arguments.
- It may or may not have a return value.
- It needs to be declared as friend kind in a class.

Syntax:

```
class class_name
{
private:
        // data member(s);
public:
friend return_type operator operator_to_be_overloaded (data_type obj1, data_type obj2)
{
           // statement(s);
}
};
```

```
class Number
{  int num;
public:
void Assign (int a)
{
num = a;
}
void Show ( )
{
cout<<num<<endl;
}
friend Number operator + (Number obj1, Number obj2)
{
Number T;
T.num = obj1.num + obj2.num;
return T;
}
friend Number operator + (Number obj, int a)
{
Number T;
T.num = obj.num + a;
return T;
}
};
```

```
void main ( )
{
Number obj1, obj2;
obj1.Assign(7);
obj2.Assign(6);

Number obj3 = obj1 + obj2;
Number obj4 = obj1 + 14;

cout<<"Object1 = ";
obj1.Show ( );
cout<<"Object2 = ";
obj2.Show ( );
cout<<"Object3 = ";
obj3.Show ( );
cout<<"Object4 = ";
obj4.Show ( );
}
```

Addition and Subtraction of two complex numbers.

```
class Complex
{
        float real, img;
public:
void Assign (float a, float b)
{
real = a;
img = b;
}
void Show ( )
{
cout<<real<<” +i ”<<img<<endl;
}
friend Complex operator + (Complex obj1, Complex obj2)
{
Complex res;
res.real = obj1.real + obj2.real;
res.img = obj1.img +obj2.img;
return res;
}
friend Complex operator - (Complex obj1, Complex obj2)
{
Complex res;
res.real = obj1.real - obj2.real;
res.img = obj1.img - obj2.img;
return res;
}
};
void main ( )
{
Complex c1, c2, c3, c4;
c1.Assign (14, 37);
c2.Assign (6, 7);
c3 = c1 + c2;
c4 = c1 - c2;
cout<<”1st Complex Number =“;
c1.Show ( );
cout<<”2nd Complex Number = “;
c2.Show ( );
cout<<”Addition = “;
c3.Show ( );
cout<<”Subtraction = “;
c4.Show ( );
}
```

Pointer to Object

Pointers can be used to hold addresses of objects; just they can hold addresses of primitive and user-defined data.

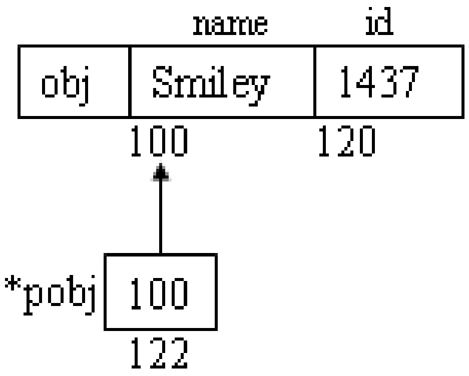
Syntax:

Declaration: class_name *pointer_to_object;

Initialization: pointer_to_object = &object;

```
class PTO
{
   char name[20];
   int id;
public:
void Get (char *a, int b)
{
strcpy (name, a); id = a;
}
void Put ( )
{
cout<<”Name : “<<name<<endl;
cout<<”ID : “<<id<<endl;
}
} obj, *pobj;
```

```
void main ()
{
obj.Get(“Smiley”, 1437);
pobj =&obj;
pobj→Put ( );
}
```



```
void main ( )
{
pobj = &obj;
pobj→Get(“Smiley”, 1437);
pobj→Put ( );
}
```

Dynamic Objects

- A class can be instantiated at runtime and objects created by such instantiation are called ***dynamic objects***.
- Two operators ***new*** and ***delete*** are used to create and release the memory allocated to the dynamic object.
- To hold the address of dynamic object, we need a ***pointer object***. These pointer objects are used to access data members or member functions of a class using → operator.

Syntax: class_name *pointer_object;

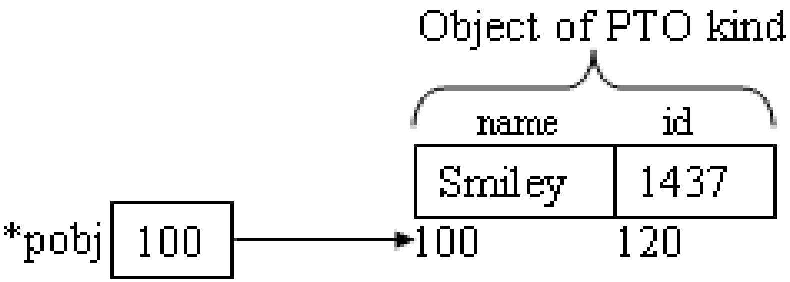
One dynamic object: pointer_object = new class_name;

Array of dynamic object: pointer_object = new class_name [size];

Syntax of deleting dynamic object: delete pointer_object;

Example: One dynamic object allocation.

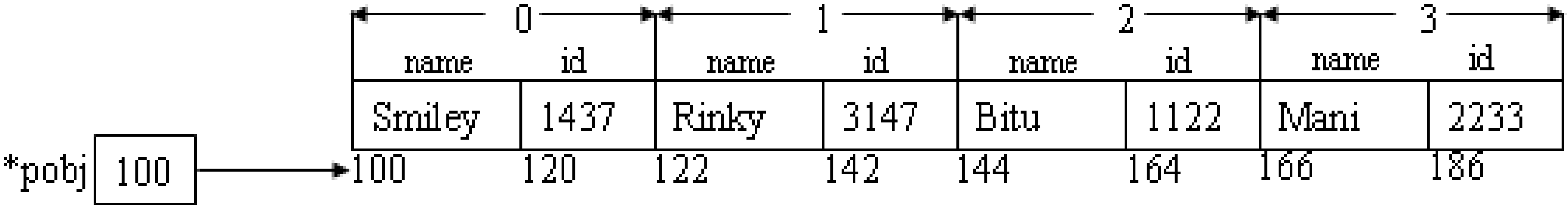
```
class PTO
{
char name[20];
intid;
public:
void Get (char *a, int b)
{
strcpy (name, a);
id = a;
}
void Put ( )
{
cout<<”Name : “<<name<<endl;
cout<<”ID : “<<id<<endl;
}
};
void main ( )
{
PTO *pobj = new PTO;
pobj→Get(“Smiley”, 1437);
pobj→Put ( );
delete pobj;
}
```



Example: Array of dynamic object allocation.

```
class PTO
{
    char name[20];
    int id;
public:
void Get (char *a, int b)
{
strcpy (name, a);
id = a;
}
void Put ( )
{
cout<<"Name : "<<name<<endl;
cout<<"ID : "<<id<<endl;
}
};
```

```
void main ( )
{
PTO *pobj = new PTO[4];
char n[20];
int id;
for (int i=0; i<4; i++)
{
cout<<"Enter Name and Id : ";
cin>>n>>id;
pobj[i].Get(n, id);
}
cout<<"Details\n";
for (i=0; i<4; i++)
    pobj[i].Put ( );
delete pobj;
}
```



0		1		2		3	
name	id	name	id	name	id	name	id
Smiley	1437	Rinky	3147	Bitu	1122	Mani	2233
100	120	122	142	144	164	166	186

Dynamic Binding or Late Binding or Dynamic Dispatch

- It is a polymorphism to be realized at the time of run-time.
- It is achieved through a special member function called **virtual function**.
- **The member function that can be changed at run-time is known as virtual function.**
- The **virtual function** must be defined in the **base class** under the public section.

Syntax of Virtual Function

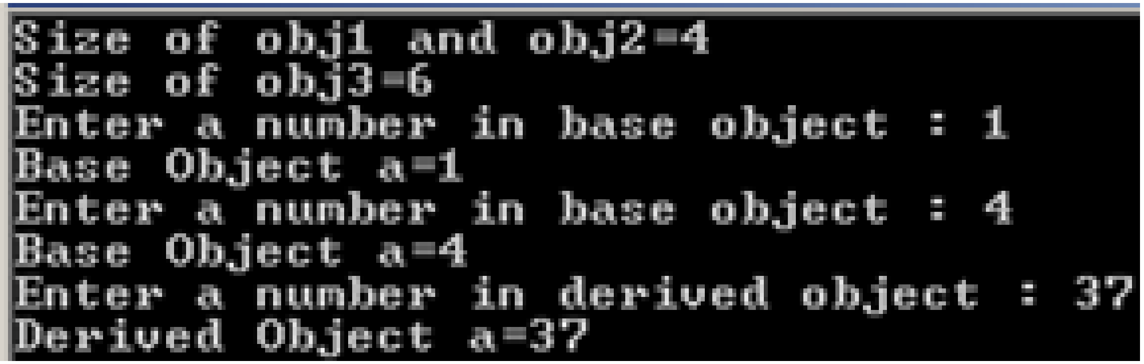
```
class base_class
{
private:
        // data member(s);
public:
   virtual return_type function_name ( Argument(s) if any )
   {
       // statement(s);
    }
};
```

Example: Virtual Function

```
class one
{
int a;
public:
virtual void accept( )
{
cout<<"Enter a number in base object : ";
cin>>a;
}
virtual void display( )
{
cout<<"Base Object a="<<a<<endl;
}
};
```

```
class two : public one
{
int a;
public:
void accept( )
{
cout<<"Enter a number in derived object : ";
cin>>a;
}
void display( )
{
cout<<"Derived Object a="<<a<<endl;
}
};
```

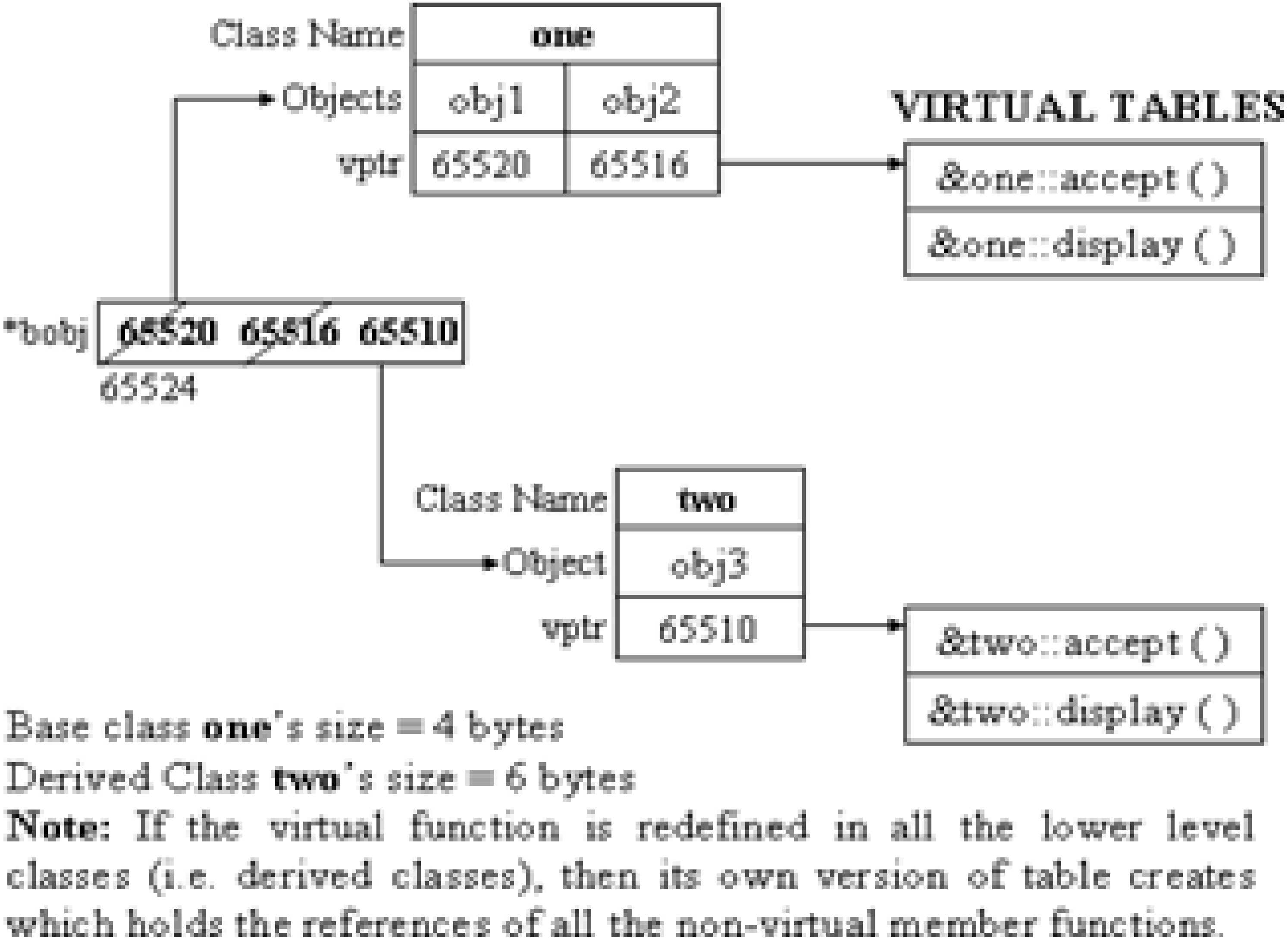
```
void main( )
{
one *bobj, obj1, obj2;
two obj3;
//4 because 2 for data member 'a' of one class and 2 for virtual function (any  number)
cout<<"Size of obj1 and obj2="<<sizeof(obj1)<<endl;

// 6 because 2 for data member 'a' of one class, 2 for virtual function, and 2 for data
// member 'a' of two class
cout<<"Size of obj3="<<sizeof(obj3)<<endl;

bobj=&obj1;
bobj → accept( );
bobj → display( );
bobj=&obj2;
bobj → accept( );
bobj → display( );
bobj=&obj3;
bobj → accept();
bobj → display();
}
```



```
Size of obj1 and obj2=4
Size of obj3=6
Enter a number in base object : 1
Base Object a=1
Enter a number in base object : 4
Base Object a=4
Enter a number in derived object : 37
Derived Object a=37
```

VTABLE and VPTR

VTABLE (virtual table) is formed for each class having virtual function and for the derived class which implements the virtual function. TABLE holds address of virtual function. The compiler places a VPTR (virtual pointer) of every object that points to the particular VTABLE.



Base class **one**'s size = 4 bytes

Derived Class **two**'s size = 6 bytes

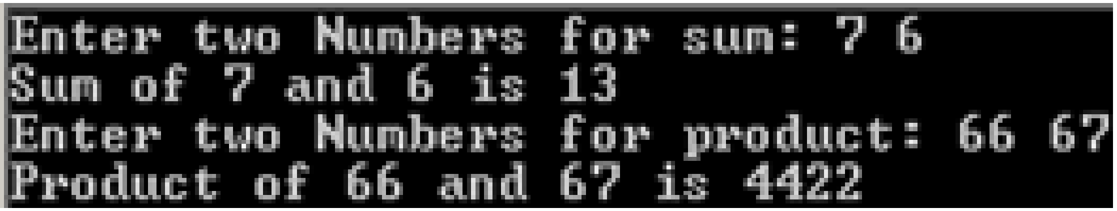
Note: If the virtual function is redefined in all the lower level classes (i.e. derived classes), then its own version of table creates which holds the references of all the non-virtual member functions.

Find sum and product of two numbers using virtual function.

```
class Sum
{
   float a, b;
public:
virtual void Read ( )
{
cout<<"Enter two Numbers for sum: ";
cin>>a>>b;
}
virtual void Result ( )
{
cout<<"Sum of "<<a<<" and "<<b<<" is "<<a+b<<endl;
}
};
```

```
class Product : public Sum
{
float a, b;
public:
void Read ()
{
cout<<"Enter two Numbers for product:";
cin>>a>>b;
}
void Result ( )
{
cout<<"Product of "<<a<<" and "<<b<<" is "<<a*b<<endl;
}
};
```

```
void main ( )
{
Sum *p, obj1;
Product obj2;
p = &obj1;
p → Read ( );
p → Result ( );
p =&obj2;
p → Read ( );
p → Result ();
}
```



```
Enter two Numbers for sum: 7 6
Sum of 7 and 6 is 13
Enter two Numbers for product: 66 67
Product of 66 and 67 is 4422
```

Pure Virtual Function

- Virtual functions are defined inside base class normally serve as a framework for future design of the class hierarchy.

- These functions can be overridden by the member functions of derived class.

- A class containing pure virtual function cannot be used to define any objects of its own.

- Such classes are called **pure abstract class** **or** **abstract class**.

- The classes without pure virtual function which can be instantiated and called **concrete classes**.

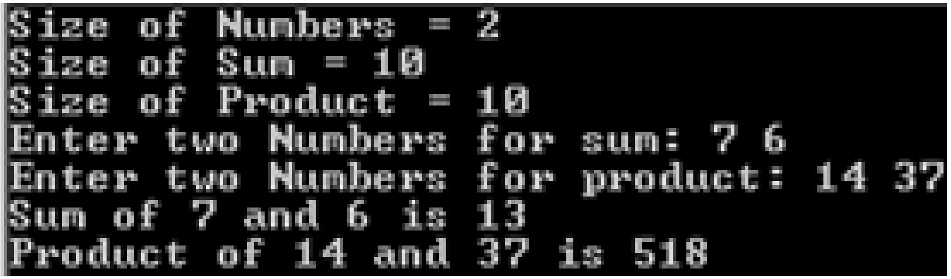
Syntax:

```
class Base_Class_Name    // pure abstract class or
{                        // abstract class
public:
   virtual return_type functionName(Argument(s), if any) = 0;
};
```

```
// Find sum and product of two numbers using pure virtual function.
class Numbers       // abstract class
{
public:
        virtual void Read() = 0;      // pure virtual function
        virtual void Result() = 0;    // pure virtual function
};
```

```
class Sum : public Numbers         // concrete class
{
        float a, b;
public:
void Read ()
{
  cout<<"Enter two Numbers for sum: ";
  cin>>a>>b;
}
void Result ( )
{
  cout<<"Sum of "<<a<<" and "<<b<<" is "<<a+b<<endl;
}
};
```

```
class Product : public Numbers      // concrete class
{   float a, b;
public:
void Read ()
{
cout<<"Enter two Numbers for product:";
cin>>a>>b;
}
void Result ( )
{
cout<<"Product of "<<a<<" and "<<b<<" is "<<a*b<<endl;
}
};
```



```
Size of Numbers = 2
Size of Sum = 10
Size of Product = 10
Enter two Numbers for sum: 7 6
Enter two Numbers for product: 14 37
Sum of 7 and 6 is 13
Product of 14 and 37 is 518
```

```
void main ( )
{
cout<<"Size of Numbers = "<<sizeof(Numbers)<<endl;
cout<<"Size of Sum = "<<sizeof(Sum)<<endl;
cout<<"Size of Product = "<<sizeof(Product)<<endl;
Sum obj1;            Product obj2;
obj1.Read ();        obj2.Read ( );
obj1.Result ( );     obj2.Result ( );
}
```

Note:

- The **abstract class** with at least **one pure virtual function** needs 2 bytes.

- Every derived class must define at least **one pure virtual function** from the base class (abstract class).

- So the derived class size is always **2 bytes** (for defining at least a pure virtual function) **+ size of all the data member(s)**.

Exception Handling

Detection of errors systematically is known as exception handling. Exceptions are generally two types.

Synchronous Exception:

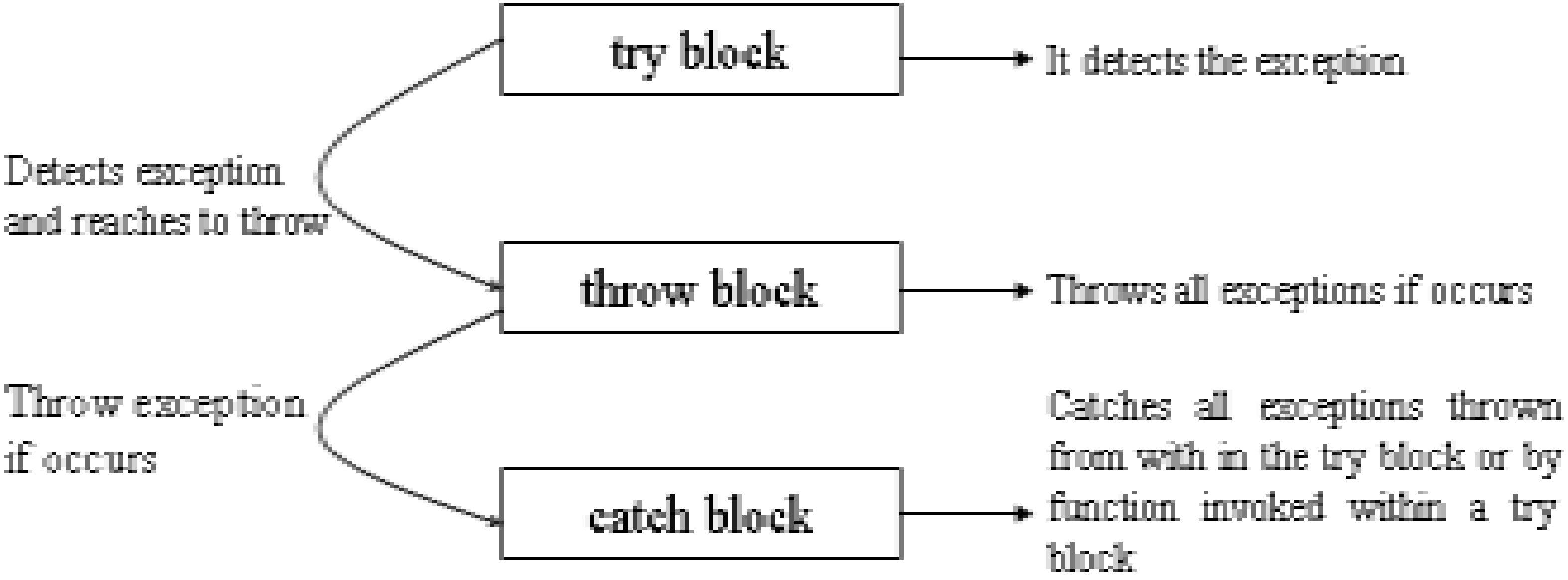
- It occurs during the program execution due to some fault in the input data or technique.
- These kinds of exceptions are detected and controlled through the program.
- It is completely related to logical or run-time error. So it is related to software coding.
- **Examples: Out-of-range, Overflow, Division by zero etc.**

Asynchronous Exception:

- It occurs by events or faults unrelated to the program and cannot be controlled through program.
- **Examples: Keyboard interrupts Hardware malfunctions, Disk failure, etc.**

Handling Model

- It uses three blocks: try, throw, and catch.
- The relationship of these three exception handling constructs called the exception handling model which is shown diagrammatically as:



try block → It detects the exception

Detects exception and reaches to throw

throw block → Throws all exceptions if occurs

Throw exception if occurs

catch block → Catches all exceptions thrown from with in the try block or by function invoked within a try block

Syntax of try block:

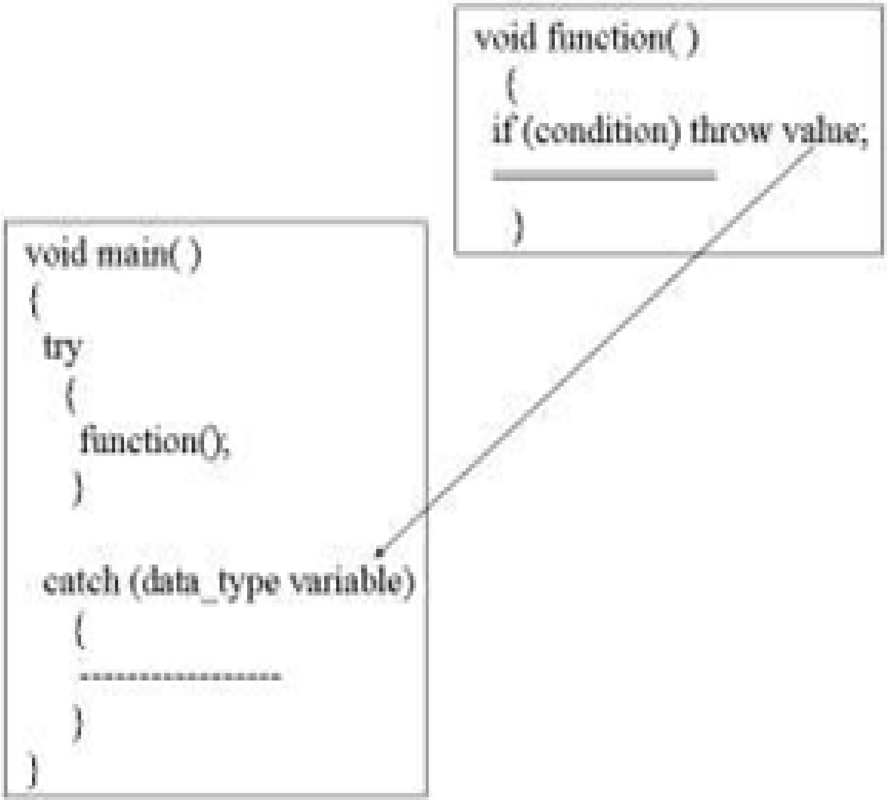
```
try
{
Statement(s);
}
```

Syntax of catch block:

```
catch (dataType var)
{
Statement(s);
}
```

Syntax of throw statement:

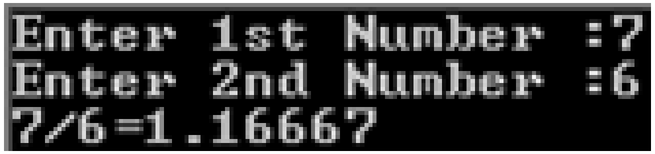
```
throw constant/variable/expression;
```



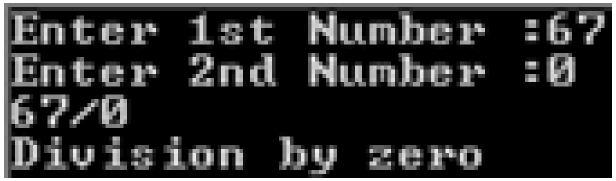
```
void function( )
  {
  if (condition) throw value;
  ___________________
  }
```

```
void main( )
{
 try
  {
   function();
  }

 catch (data_type variable)
  {
   ----------------
  }
}
```

```
// Read two numbers. Find the division.
int main()
{
float num1, num2;
try
{
cout<<"Enter 1st Number :";
cin>>num1;
cout<<"Enter 2nd Number :";
cin>>num2;
if(num2==0) throw num2;
float res = num1/num2;
cout<<num1<<"/"<<num2<<"="<<res;
}
catch(float a)
{
cout<<num1<<"/"<<num2<<endl;
cout<<"Division by zero";
}
return 0;
}
```



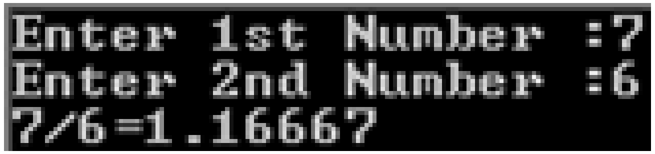
```
Enter 1st Number :7
Enter 2nd Number :6
7/6=1.16667
```



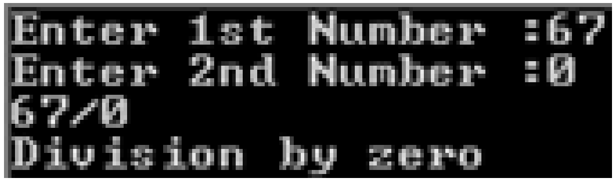
```
Enter 1st Number :67
Enter 2nd Number :0
67/0
Division by zero
```

```
// Read two numbers. Find the division
// using function.
float Divide (float a, float b)
{
if (b==0) throw b;
return a/b;
}
int main()
{
float num1, num2;
try
{
cout<<"Enter 1st Number :";
cin>>num1;
cout<<"Enter 2nd Number :";
cin>>num2;
float res = Divide(num1,num2);
cout<<num1<<"/"<<num2<<"="<<res;
}
```

```
catch (float a)
{
cout<<num1<<"/"<<num2<<endl;
cout<<"Division by zero";
}
return 0;
}
```



```
Enter 1st Number :7
Enter 2nd Number :6
7/6=1.16667
```



```
Enter 1st Number :67
Enter 2nd Number :0
67/0
Division by zero
```

Catch All Exceptions [catch(…)]

- To catch any kind of exception by the exception handler.

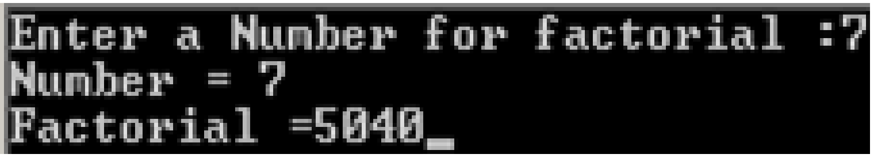
- The exception handler has one argument i.e. ellipsis(...) which can receive any kind of data type.

Can receive any kind of data thrown by the throw statement.

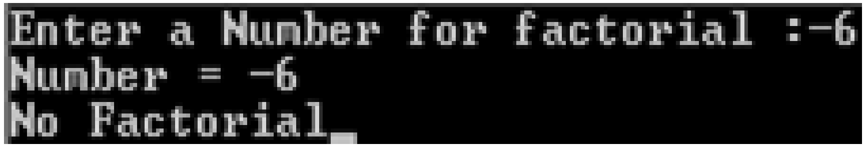
Syntax:

```
catch ( … )
{
statement (s);
}
```

```
// Finding factorial of a number using catch all exception method.
long Fact(int a)
{
if(a<0) throw a;
long res =1;
for(int i=1; i<=a; i++)
res = res*i;
return res;
}
int main()
{
int num;
try
{
cout<<"Enter a Number for factorial :";
cin>>num;
long res = Fact(num);
cout<<"Number = "<<num<<endl<<"Factorial ="<<res;
}
catch(...)
{
cout<<"Number = "<<num<<endl;
cout<<"No Factorial";
}
return 0;
}
```



```
Enter a Number for factorial :7
Number = 7
Factorial =5040_
```



```
Enter a Number for factorial :-6
Number = -6
No Factorial_
```

Multiple Exception

- If a program has more than one exception then it is necessary to have that many exception handlers.
- One exception handler is used for solving one kind of exception.
- These many exception handlers are defined after the try block.

Syntax:

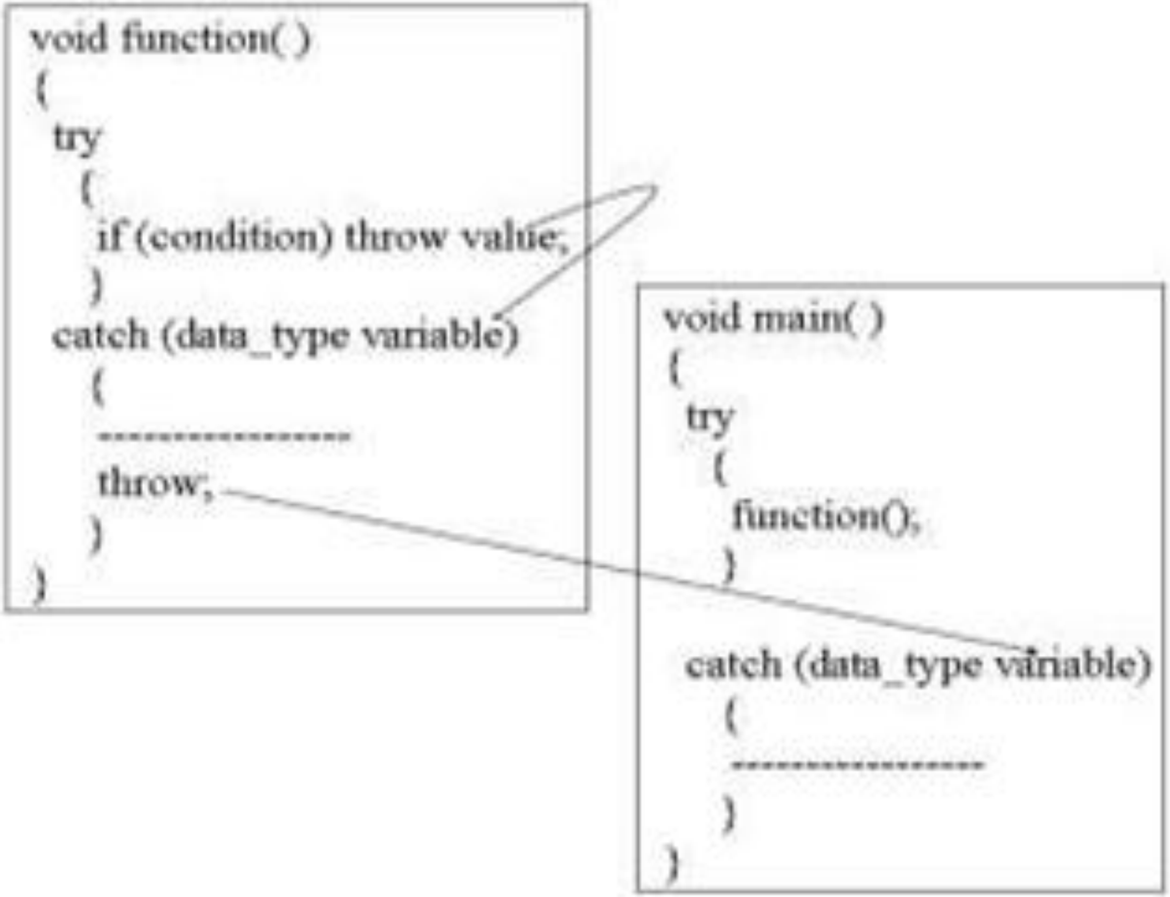
```
try
{
statement(s);
}
catch (data_type1 variable)
{
statement(s);
}
catch (data_type2 variable)
{
statement(s);
}
……………………………………
catch (data_typeN variable)
{
statement(s);
}
```

```
// Check a number is positive or negative or
// zero using multiple exception.

void Check(float num)
{
if(num>0) throw 'a';
else if(num<0) throw 67;
else throw num;
}

int main()
{
float num;
try
{
cout<<"Enter a Number :";
cin>>num;
Check(num);
}
```

```
catch(char a)
{
cout<<"Positive";
}

catch(int a)
{
cout<<"Negative";
}

catch(float a)
{
cout<<"Zero";
}
return 0;
}
```

Rethrowing Exception

- To rethrow an exception, the statement is throw.
- It allows an exception without any argument.
- It causes the current exception to be thrown to the next enclosing **try/catch block**.
- Syntax: **throw;**

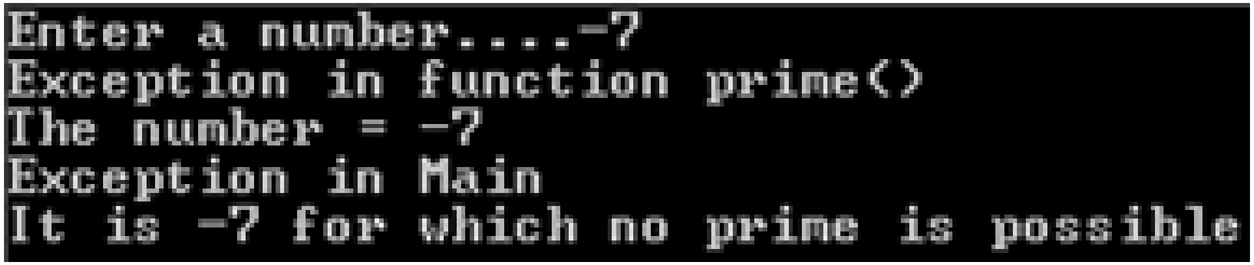


```
void function( )
{
  try
   {
    if (condition) throw value;
   }
  catch (data_type variable)
   {
    ----------------
    throw;
   }
}
```

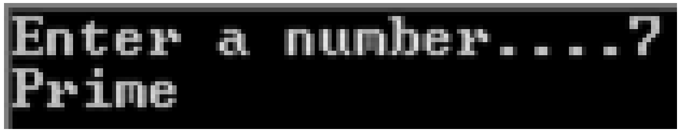
```
void main( )
{
  try
   {
    function();
   }

  catch (data_type variable)
   {
    ----------------
   }
}
```

```
// Check a number is prime or not using rethrow exception method.
int prime(int n)
{
int c=0;
try
{
if(n<=0) throw n;
for(int i=1;i<=n;i++)
    if(n%i == 0) c++;
return c;
}
catch(int e)
{
cout<<"Exception in function prime()\n";
cout<<"The number = "<<e<<endl;
throw;
}
}
```

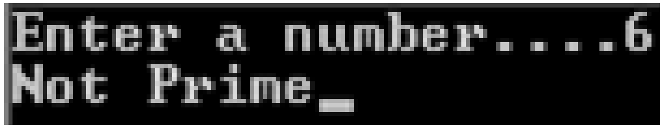
```
int main()
{
int num;
try
{
cout<<"Enter a number ";
cin>>num;
int r=prime(num);
if(r==2) cout<<"Prime";
else cout<<"Not Prime";
}
catch(int a)
{
cout<<"Exception in Main\n";
cout<<"It is "<<a<<" for which no prime is possible";
}
return 0;
}
```



```
Enter a number....-7
Exception in function prime()
The number = -7
Exception in Main
It is -7 for which no prime is possible
```



```
Enter a number....7
Prime
```



```
Enter a number....6
Not Prime_
```

Object Slicing

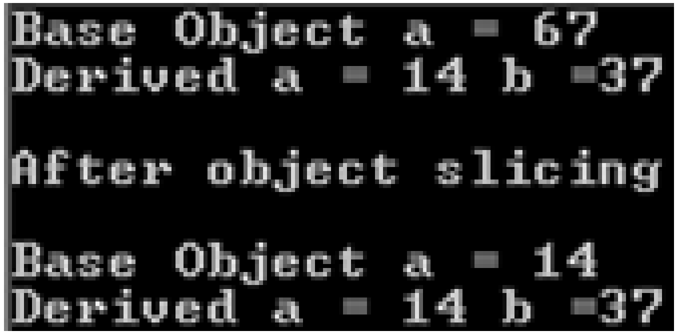
To assign the derived object to the base object, which the compiler allows it. Only the data members of base object are assigned.

Example-1

```
class one
{
protected:
        int a;
public:
void Assign(int x)
{
a = x;
}
void Display ()
{
cout<<"Base Object a = "<<a<<endl;
}
};
```

```
class two : public one
{
int b;
public:
void Assign(int x, int y)
{
one::Assign(x);
b=y;
}
void Display ()
{
cout<<"Derived a = "<<a;
cout<<" b ="<<b<<endl;
}
};
```

```
void main()
{
one obj1;
two obj2;

obj1.Assign(67);
obj2.Assign(14,37);

obj1.Display();
obj2.Display();

obj1 = obj2;         // object slicing takes place

cout<<"\nAfter object slicing\n\n";
obj1.Display();
obj2.Display();
}
```



```
Base Object a = 67
Derived a = 14 b =37

After object slicing

Base Object a = 14
Derived a = 14 b =37
```

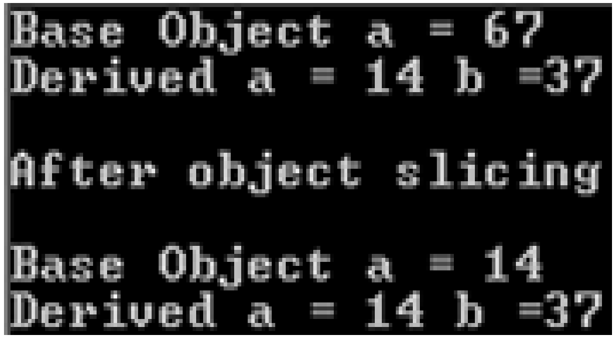
Note:
obj1 = obj2; In this expression the assignment operator (=) is used for assignment of data members of derived object to the data members of base object.

Example-2

```
class one
{
protected:
            int a;
public:
one(int x)
{
a = x;
}
void display ()
{
cout<<"Base Object a = "<<a<<endl;
}
};
```

```
class two : public one
{
    int b;
public:
two(int x, int y):one(x)
{
b=y;
}
void display ()
{
cout<<"Derived Object a = "<<a;
cout<<" b = "<<b<<endl;
}
};
```

```
void main()
{
one obj1(67);
two obj2(14, 37);

obj1.display();
obj2.display();

obj1 = obj2;    // object slicing takes place

cout<<"\nAfter object slicing\n\n";
obj1.display();
obj2.display();
}
```



```
Base Object a = 67
Derived a = 14 b =37

After object slicing

Base Object a = 14
Derived a = 14 b =37
```

Generic Programming With Templates

Template supports generic programming, which allows developing reusable software components such as functions, classes etc, and supporting different data types in a single framework.

The templates declared for functions are called **function templates** and those declared for classes are called **class templates**.

They perform appropriate operations depending on the data type of the parameters passed to them.

It allows a single template to deal with a generic data type.

Function Template

A function Template or generic function specifies how an individual function can be constructed.

Syntax:

```
template <class/typename template_data_type_name,…..>
return_type function_name(template_data_type_name variable1,…)
{
    // statement(s)
}
```

Note:

At least one argument of function template must be template type.

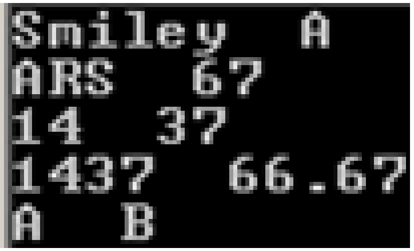
```
template <class T>
//template <typename T>
void Show (T a)
{
cout<<a<<endl;
}

void main ( )
{
Show(67);
Show(14.37);
Show(‘A’);
Show(“Smiley”);
}
```

```
Output:
67
14.37
A
Smiley
```

```
template<class T>
// one argument primitive and another one template kind
void Show (char *p, T a)
{
cout<<p<<a<<endl;
}
void main ( )
{
Show(“The Integer = ”, 67);
Show(“The Float = “, 14.37);
Show(“The Character =”, ‘A’);
Show(“The String = “, “Smiley”);
}
```

```
Output:
The Integer = 67
The Float = 14.37
The Character = A
The String = Smiley
```

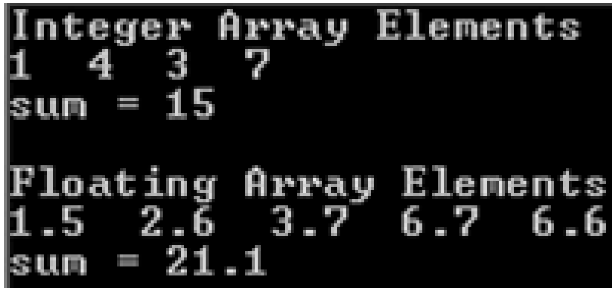
```
template <typename A, typename B>
void Show(A a, B b)
{
cout<<a<<""<<b<<endl;
}

int main()
{
Show("Smiley", 'A');
Show("ARS", 67);
Show(14, 37);
Show(1437, 66.67);
Show('A', 'B');
return 0;
}
```



```
Smiley   A
ARS   67
14   37
1437   66.67
A   B
```

```
// Sum of array of integers and array of
// floating numbers.
template<class T>
// T sum(T a[ ], int n)
T sum(T *a, int n)
{
T s = 0;
for (int i = 0; i < n; i ++)
s = s + a [i];
return s;
}

template<class T>
// void show(T a[], int n)
void show(T *a,int n)
{
for (int i = 0; i < n; i ++)
cout<<a [i]<<" ";
cout<<endl;
}
```

```
int main()
{
int a[4] = {1, 4, 3, 7}, res1;
float b[5] = {1.5, 2.6, 3.7, 6.7, 6.6}, res2;

res1 = sum (a, 4);
res2 = sum (b, 5);

cout<<"Integer Array Elements\n";
show(a, 4);
cout <<"sum = "<< res1 << endl;

cout<<"\nFloating Array Elements\n";
show(b, 5);
cout <<"sum = "<<res2;
return 0;
}
```



```
Integer Array Elements
1   4   3   7
sum = 15

Floating Array Elements
1.5    2.6    3.7    6.7    6.6
sum = 21.1
```

Class Templates

Similar to functions, classes can also be declared to operate on different data types.

Such classes are called class templates.

Definition-1:
A class template specifies how individual classes can be constructed.

Defintion-2:
A class template provides a specification for generating classes based on parameters.

Class Template Syntax:

```
template<class/typename template_data_type_name,……..>
class class_name
{
private:
          template_data_type_name var1, var2,……;

public:
// prototype or declaration
return_type function_name(template_data_type_name, template_data_type_name);

};

template <class/typename template_data_type_name,…..>
return_type class_name<template_data_type_name,……> ::
                            function_name (template_data_type_name variable1, …….)
{
// statement(s);
}
```

Template Class

A class template is instantiated by passing a given set of types to it as template arguments.

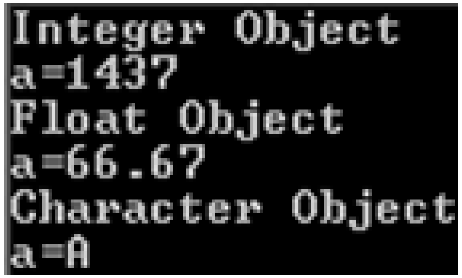
```
void main ( )
{
class_name<template_data_type_name,…> object_name1,…;
}
```

```
template<class T>
class Values // class template
{   T a;
public:
void assign(T b)
{
a = b;
}
void show( )
{
cout<<" Value = "<<a<<endl;
}
};
```

```
Output
Size of Class Template of Character = 1
Size of Class Template of Integer = 2
Size of Class Template of Float = 4
```

```
void main()
{
// template class of int kind is created and available in the memory
Values<int> iobj;
// template class of char kind is created and available in the memory
Values<char> cobj;
// template class of float kind is created and available in the memory
Values<float> fobj;
cout<<"Size of Class Template of Character = "<<sizeof(Values<char>)<<endl;
cout<<"Size of Class Template of Integer = "<<sizeof(Values<int>)<<endl;
cout<<"Size of Class Template of Float = "<<sizeof(Values<float>)<<endl;
}
```

```
template <class T>
class ARS      // class template
{
private :
T a;
public :
void Assign (T b)
{
a = b;
}
void Show ( )
{
cout<<”a=”<<a<<endl;
}
};
```

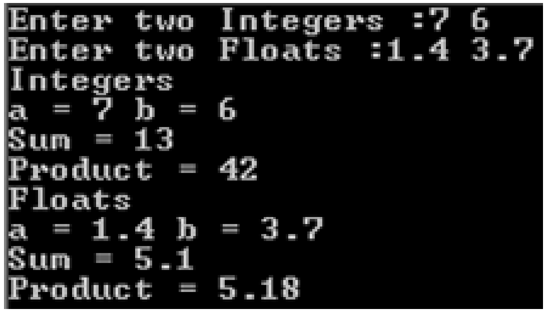


```
Integer Object
a=1437
Float Object
a=66.67
Character Object
a=A
```

```
int main ( )
{
ARS<int>IObj;          // template class
ARS<float> FObj;     // template class
ARS<char> CObj;     // template class

IObj.Assign(1437);
FObj.Assign(66.67);
CObj.Assign(‘A’);

cout << “Integer\n”;
IObj.Show( );

cout <<“Float\n”;
FObj.Show();

cout <<"Character\n";
CObj.Show();
return 0;
}
```

```
// Sum and product of two numbers.
template <class T>
class SP
{        T a, b;
public :
void Get()
{
cin>>a>>b;
}
void Show ( )
{
cout<<"a = "<<a<<" b = "<<b<<endl;
}
void Sum ( )
{
T r = a+b;
cout<<"Sum = "<<r<<endl;
}
void Product ( )
{
T r = a*b;
cout<<"Product = "<<r<<endl;
}
};
```

```
void main ( )
{
SP<int> obj1;
SP<float> obj2;
cout<<"Enter two Integers :";
obj1.Get();
cout<<"Enter two Floats :";
obj2.Get();
cout <<"Integers\n";
obj1.Show();
obj1.Sum();
obj1.Product();
cout <<"Floats\n";
obj2.Show();
obj2.Sum();
obj2.Product();
}
```



```
Enter two Integers :7 6
Enter two Floats :1.4 3.7
Integers
a = 7 b = 6
Sum = 13
Product = 42
Floats
a = 1.4 b = 3.7
Sum = 5.1
Product = 5.18
```

```
template<class T>
class Smiley
{
T a;
public:
void Assign(T b)
{
a = b;
}
void Display(int);
};

template<class M>
void Smiley<M>::Display(int n)
{
for(int i=1; i<=n; i++)
      cout<<a<<endl;
}
```

```
void main( )
{
Smiley<int> A;
Smiley<char> B;
Smiley<float> C;
A.Assign(67);
B.Assign('@');
C.Assign(6.7);

A.Display(5);
B.Display(7);
C.Display(2);
}
```

Output

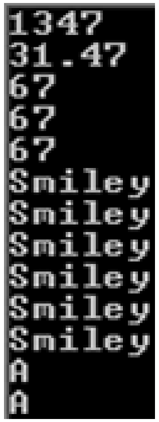
```
67
67
67
67
67
@
@
@
@
@
@
@
6.7
6.7
```

Overloaded function Templates

It may be overloaded either by (other) functions of its name or by (other) templates function of the same name.

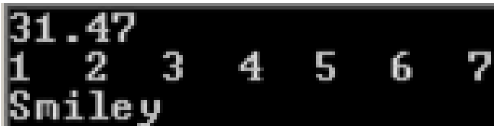
```
template < class T>
// function template with template argument
void print(T a)
{
cout <<a <<endl;
}

template <class T>
// function template with template and primitive argument
void print(T a, int n)
{
for (int i=1; i<=n; i++)
     cout << a << endl;
}
```

```
int main( )
{
print (1347);
print (31.47);
print (67,3);
print ("Smiley", 6);
print('A', 2);
return 0;
}
```



```
1347
31.47
67
67
67
Smiley
Smiley
Smiley
Smiley
Smiley
Smiley
A
A
```

```
template < class T>
// function template with template argument
void print (T a)
{
cout <<a <<endl;
}

 // normal function with primitive argument
void print(int n)
{
for (int i=1; i<=n; i++)
  cout << i <<" ";
cout<<endl;
}

void main()
{
   print(31.47);         // invokes function template
// invokes normal function rather than invoking function template
// because normal function has priority than the function template
    print(7);

    print ("Smiley");    // invokes function template
}
```



```
31.47
1   2   3   4   5   6   7
Smiley
```