

1. Introduction to OOPs

Dr.T.Logeswari

Introduction

- C++ is an object oriented programming languages
- Initially named 'C with classes', C++ was developed by Bjarne Stroustrup at AT&T Bell laboratories in early eighties
- C++ is the combination of object oriented feature of a languages called Simula 67 and power and elegance of C

Programming Paradigm

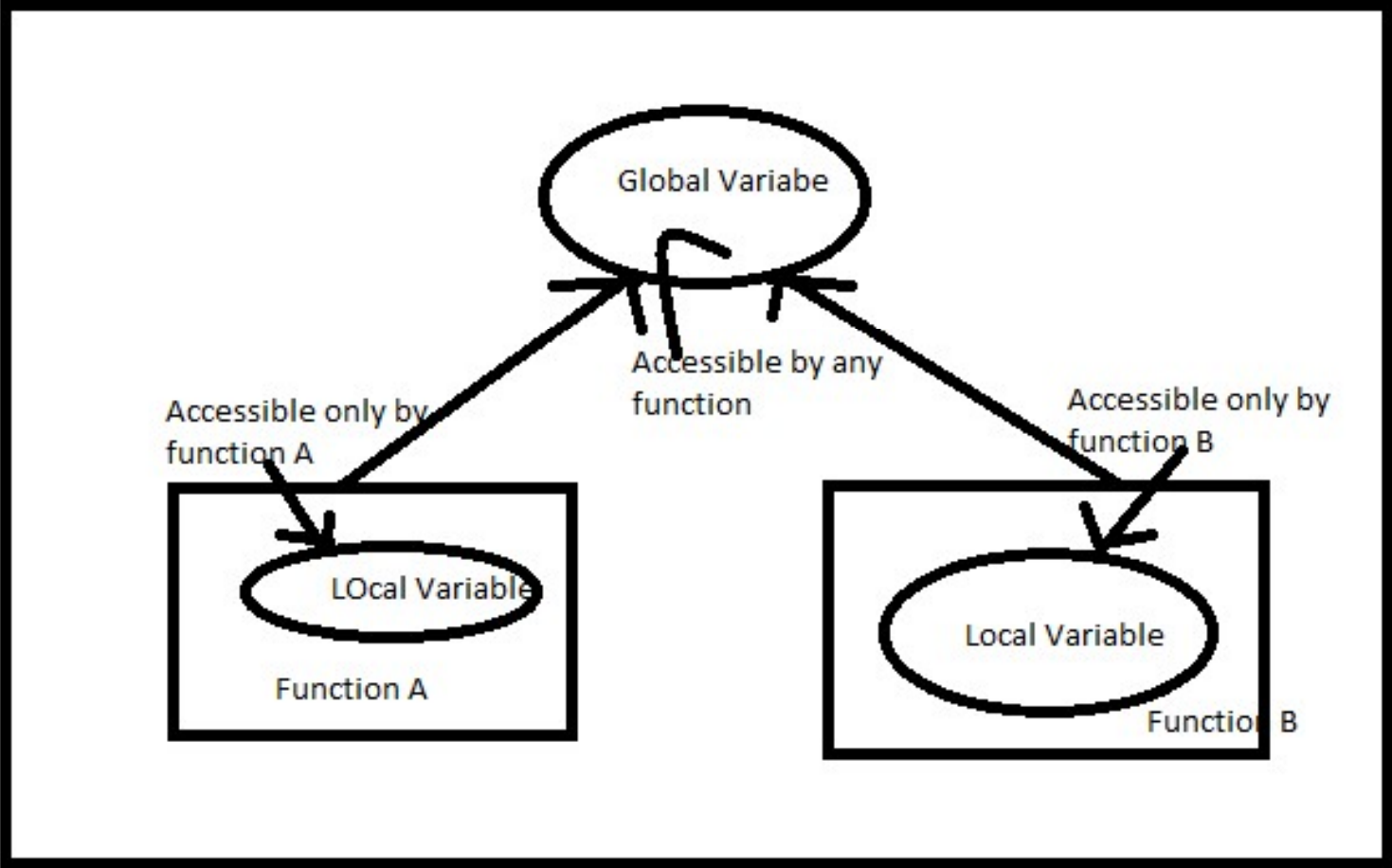
- Evolution of Programming Paradigm
 - Procedure Oriented Programming
 - Structured Oriented Programming
 - Object Oriented Programming

Procedure Oriented Programming

- Procedural language
 - The construction of program that are the collections of interacting function or procedure
 - Ex
 - Step 1: get two number
 - Step 2: add these number
 - Step 3: divide by 100
 - Step 4: display the output
- Each statement tell the computer to do something
It consist of series of steps or procedure take place

- Programmer write procedural program in many programming languages such as COBOL, Pascal, and Fortran are procedural languages, since they use procedure oriented programming approach
- A typical program consist of list of instruction to accomplish a task
- larger program are divided into number of function and then data
- In large complex multi function program , the data item may be global as well as local

Global & Local variable in procedural programming



- **Features**

- Emphasis is laid on the algorithm
- Large and complex program divided into function
- Function share global data
- Data is passed from function to function
- Top down approach is used in program design

Drawback

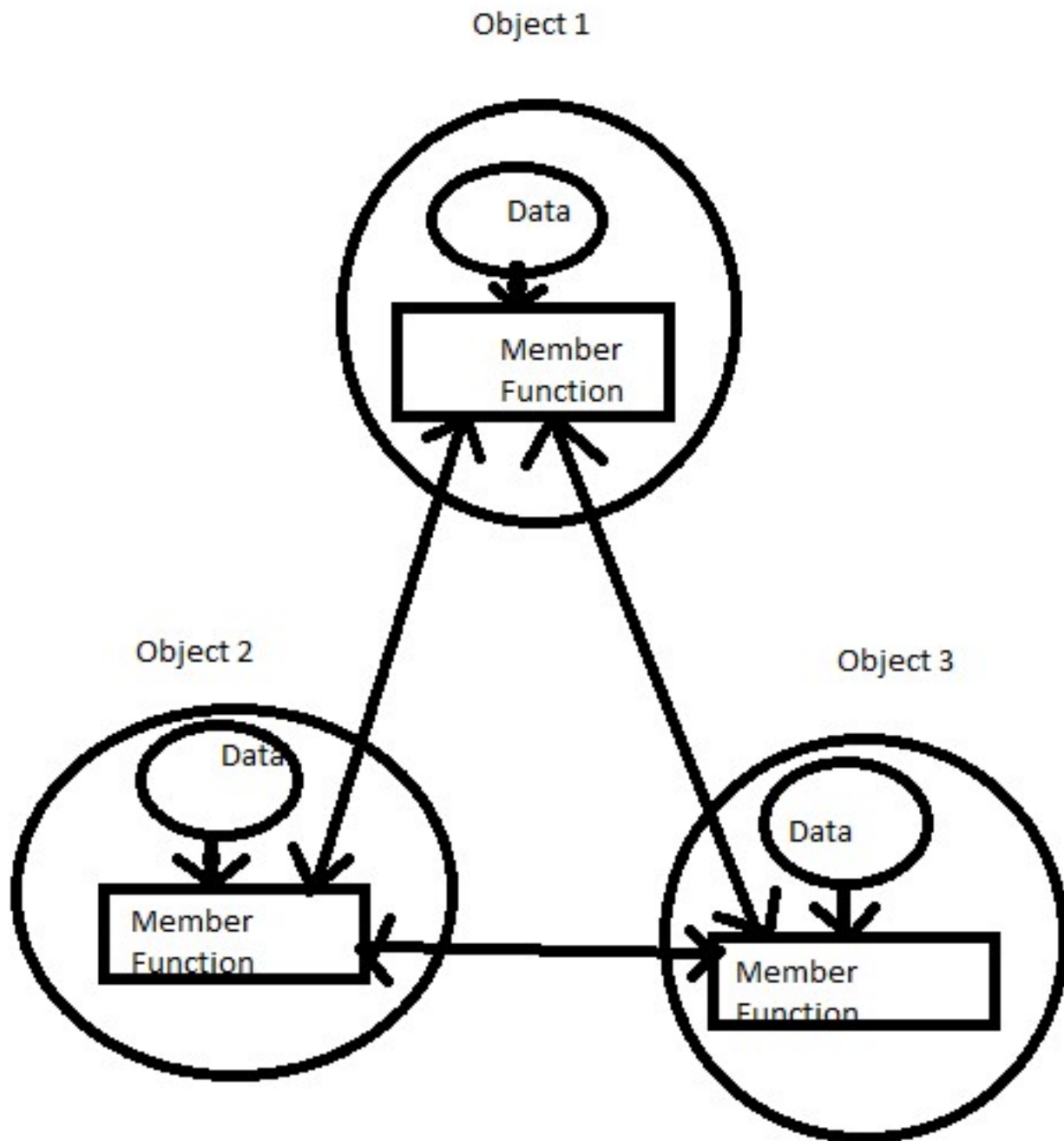
- If complex increase as the program grow larger and complex
- It does not model real world problems as close to user perspective as possible

Structured Programming

- It deals only with logic and code
- It suggest that well structured or organized program can be written using modularity, sequence, selection and iteration
- The term function used in C & C++. The other language use subroutine, subprogram or method.

Object oriented programming

- OOPS took basic idea of structured programming and combined them with several new concepts.
- OOPS does not allow data to move freely between the function.
- It combines both data and function that operate on that data into a single unit called an object.
- C++ program generally consists of a number of such objects which can communicate with each other through member functions.
- i.e. a function belonging to one object can access the function of another object.



Definition of OOPS

“Object oriented programming is a programming methodology that associates **data structures** with a set of **operators** which act upon it.”

OR

An approach resulting in modular programs by creating partitioned memory area for both data and function, which can be used as templates for creating copies of such modules

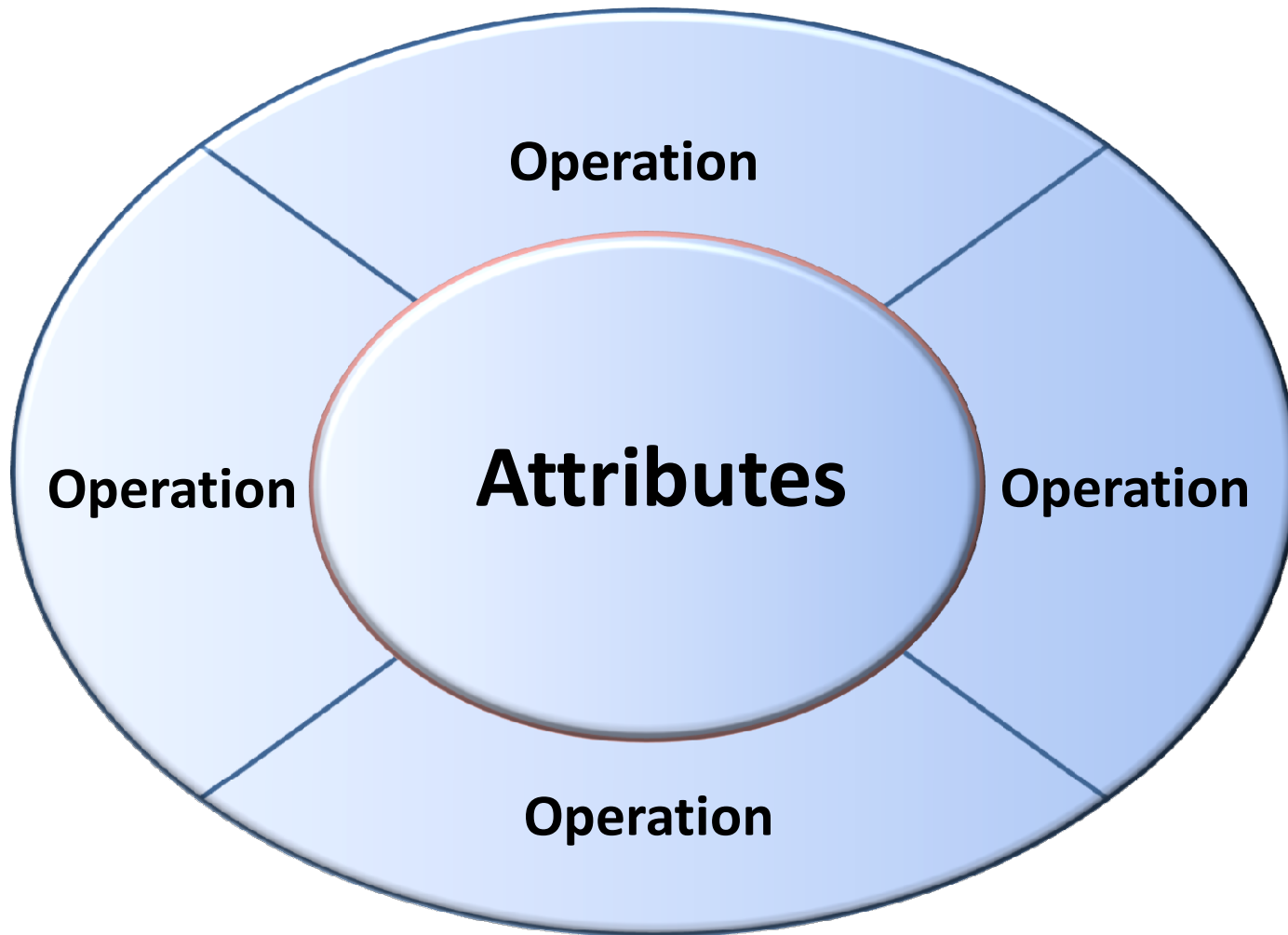
Elements of OOP

- Objects
- Classes
- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

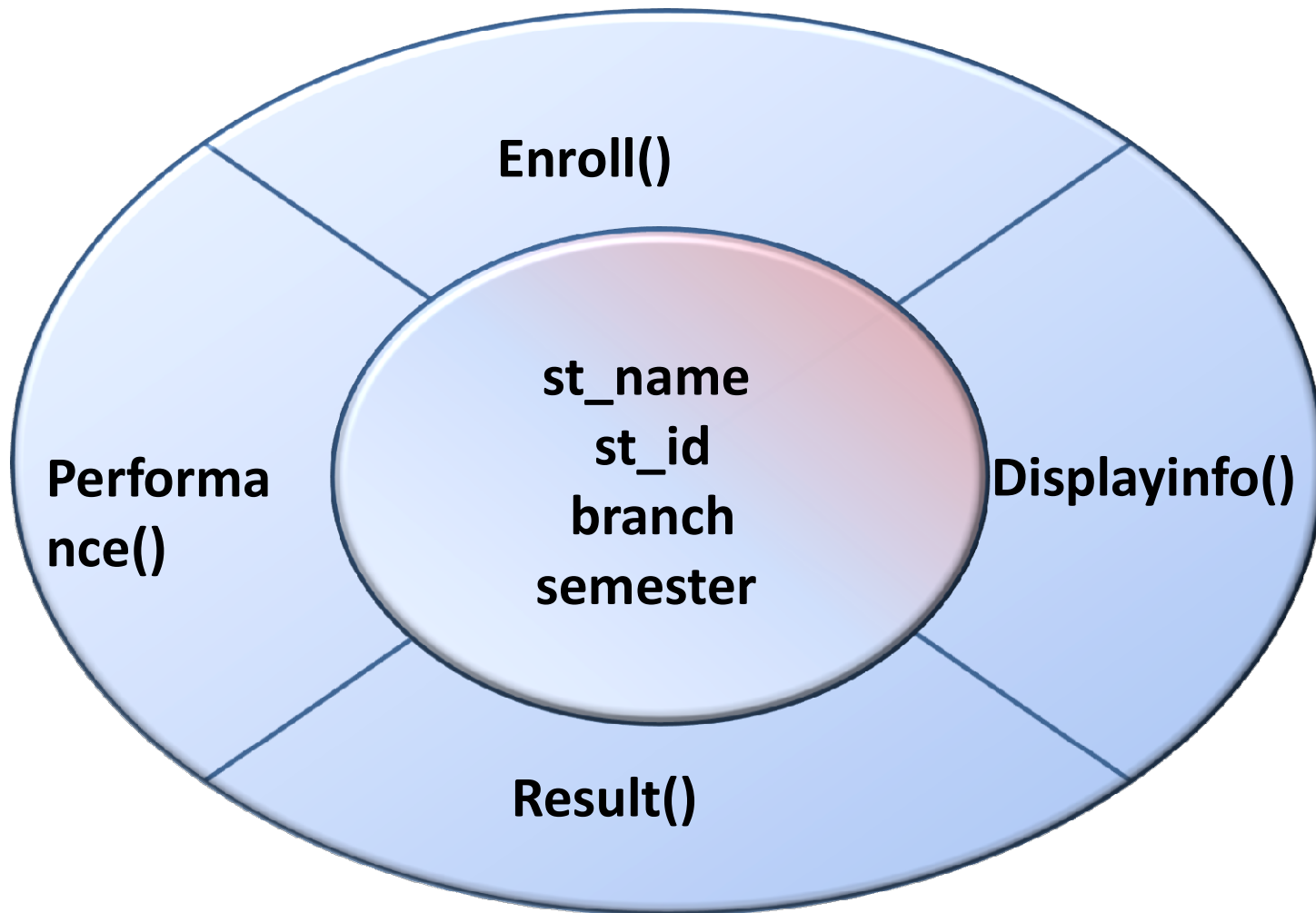
Objects

- OOP uses objects as its **fundamental building blocks**.
- Objects are the **basic run-time entities** in an object-oriented system.
- Every object is associated with **data** and **functions** which define meaningful operations on that object.
- Object is a real world **existing entity**.
- Object are **state** or **behavior**
- Object is an **Instance** of a particular class.

Object

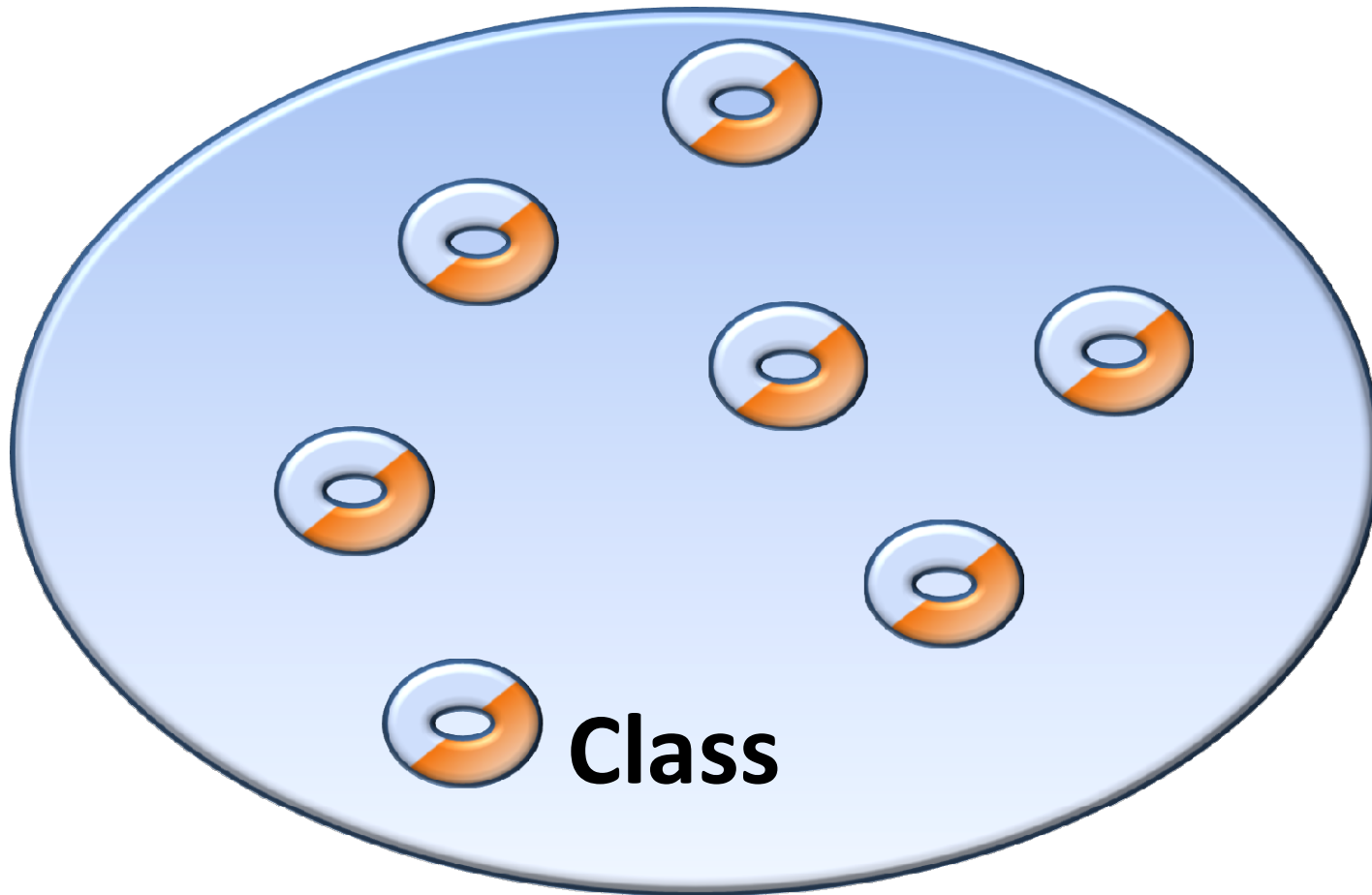


Example: StudentObject



Class

- Class is a collection of **similar objects**.



- Class = template or blueprint that describe Behavior
 - A class is created **with data members and member function**
 - Once class is defined any number of object belonging to this class can be created

Eg

Class : Shape Class : Automobile

Object : ?

Class

```
class class_name  
{  
Attributes;//Properties  
Operations;//Behaviours  
};
```

Class example

```
class student
{
char st_name[30];
char st_id[10];
char branch[10];
char semester[10];
Void Enroll( );
Void Displayinfo( );
Voide Result( );
Void Performance( );
};
```

Encapsulation

“Mechanism that associates the **code** and the **data** it manipulates into a single unit and keeps them safe from external interference and misuse.”

The data can only be accessible only through the function otherwise it is hidden from user

This insulation of data from direct access by the program is called data hiding

Encapsulation = **Data Hiding + Abstraction**

Encapsulation

Class: student

Attributes: st_name, st_id,
branch, semester

Functions: Enroll()
Displayinfo()
Result()
Performance()

Data Abstraction

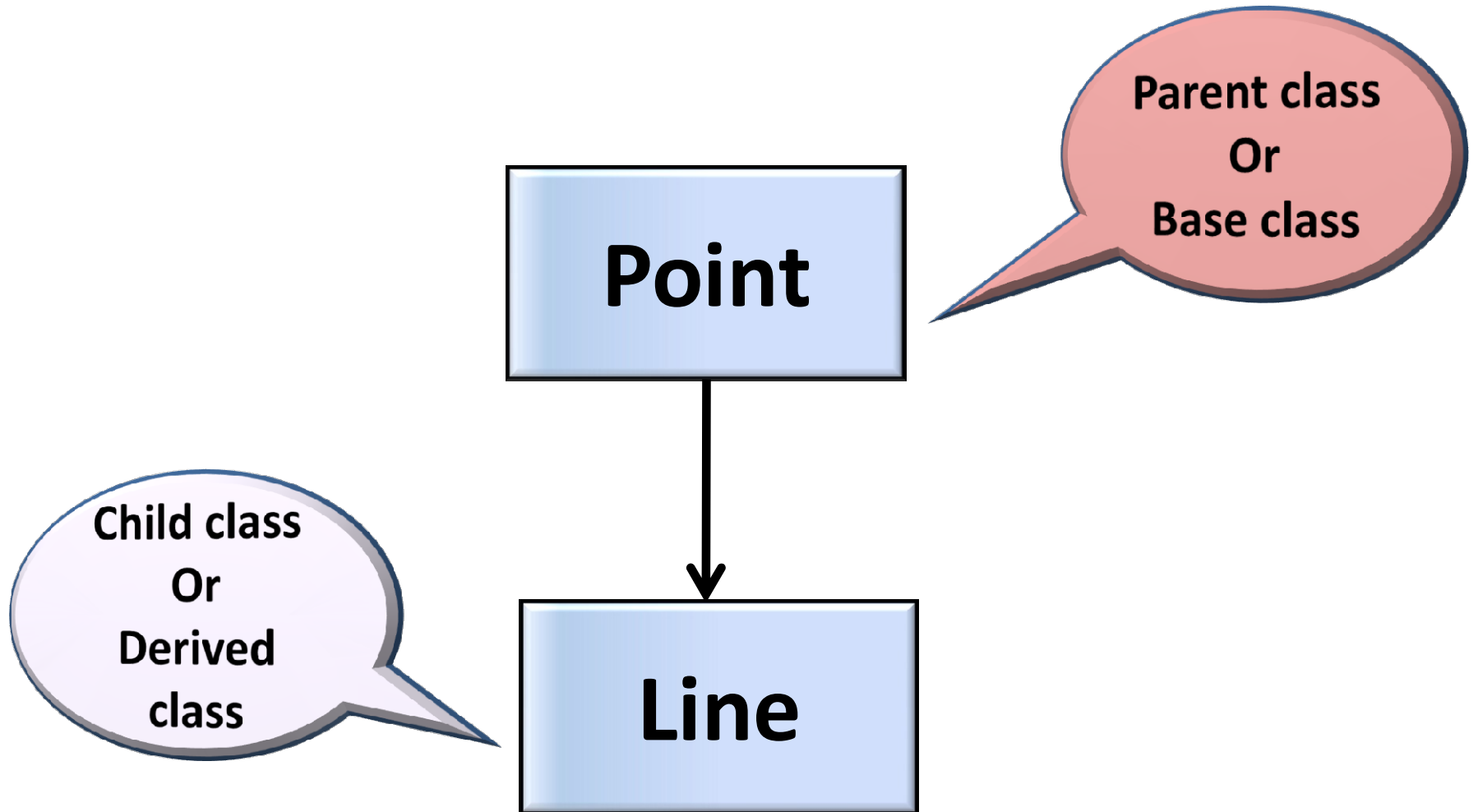
“A data abstraction is a **simplified view** of an object that includes only features one is **interested** in while **hides** away the **unnecessary** details.”

“Data abstraction becomes an **abstract data type** (ADT) or a user-defined type.”

Inheritance

- “Inheritance is the mechanism to provides the power of **reusability** and **extendibility**.”
- “Inheritance is the process by which one **object can acquire the properties of another object.**”

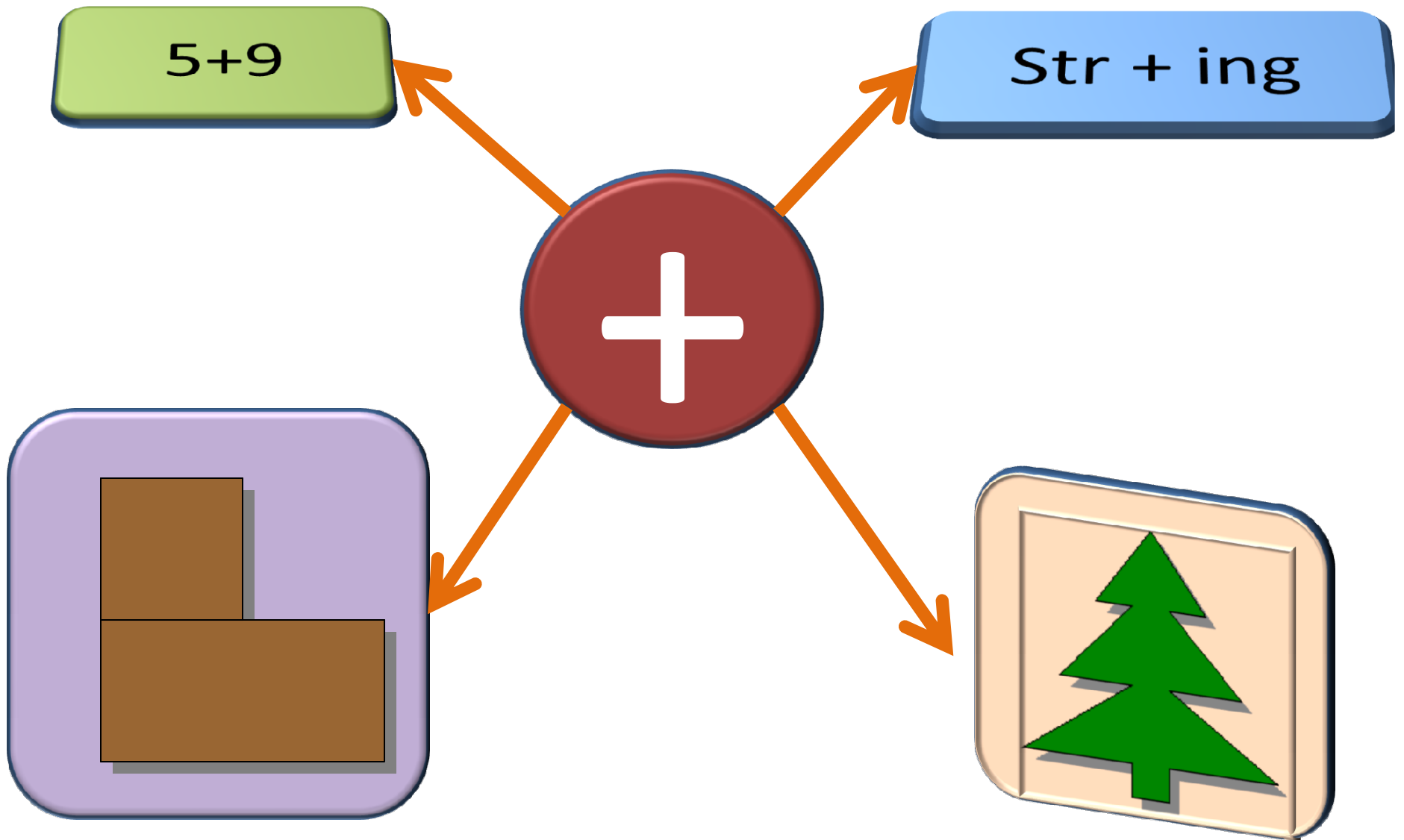
Inheritance



Polymorphism

- Polymorphism means that the **same thing** can exist in **two forms**.
- “Polymorphism is in short the ability to call **different functions** by just using **one** type of **function call**.”

Polymorphism



Dynamic Binding

“ Dynamic Binding is the process of **linking** of the **code** associated with a **procedure call** at the **run-time**”.

Two types

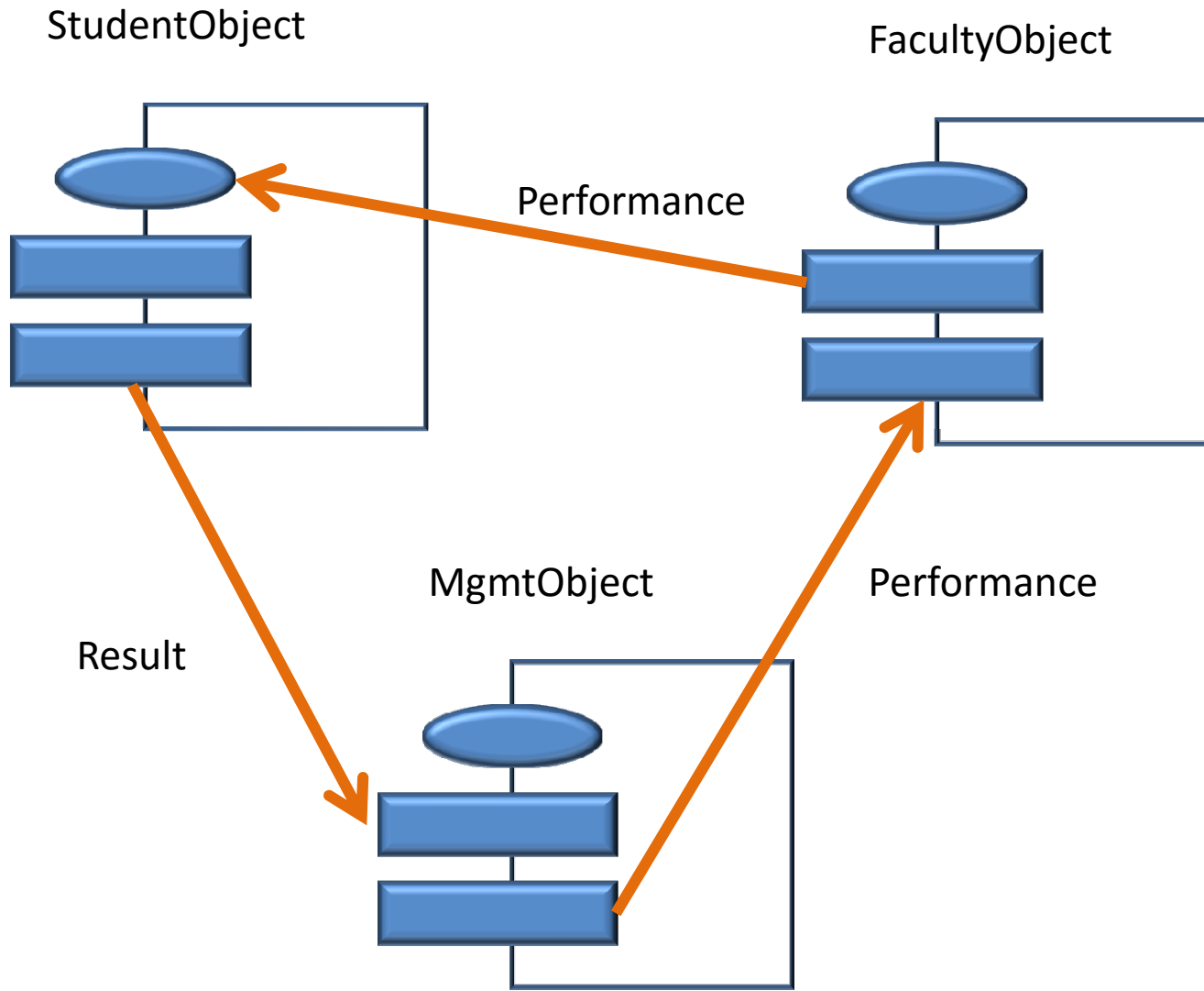
static binding – it will take place during compilation

dynamic binding – the code associated with function call is not known until runtime

Message Passing

“The process of **invoking** an operation on an object. In response to a message the **corresponding** method is executed in the object”.

Message Passing



Characteristic of OOP

- Emphasis is laid on data
- Program are divided into object
- Object contain data and function that operate on the data
- Data is hidden and cannot be accessed by outside function
- Object are allowed to communicate with each other through function
- Bottom up approach is used in program design

Benefits of OOP

- Software complexity can be managed easily
- Exception checking and error handling can be implemented
- Multiple instance of an object is possible
- Message passing techniques makes it easier to communicate with external system
- Data hiding helps programmer with data security

Object Oriented Languages

- The feature of object oriented programming languages
 - Data encapsulation
 - Data hiding and access mechanism
 - Automatic initialization of object
 - Operator overloading
 - Inheritance
 - Dynamic binding

Object Oriented Languages

1. Object-Oriented programming Languages

Examples: Simula, Smalltalk80, Objective C, java, C++, etc.,

Application of OOP

- Real time systems
- Object oriented databases
- AI and Expert system
- Simulation and modeling
- Neural network
- Office automation system

Comparison of POP and OOP

POP	OOP
Focus is on the function	Focus is on the data
Data is not secure and can be corrupted	Data is secure
Use top down programming design	Use bottom up programming design
Does not model real world problem	Models real world problem
Programs are divided into function	Program are divided into object

Structured Vs Object Oriented Programming

Function Oriented

Procedure Abstraction

Does not support
External Interface

Free flow of Data

Also called FOP

Object Oriented

Procedure & Data abstraction

Supports External Interface

Secured Data & not freely
flows

Also called OOP

2. C++ Feature

Dr.T.Logeswari

C++ Basic Elements

- Programming language is a set of rules, symbols, and special words used to construct programs.
There are certain elements that are common to all programming languages.

C++ Character Set

- Character set is a set of valid characters that a language can recognize.

Letters A-Z, a-z

Digits 0-9

Special Characters Space + - * / ^ \ () [] { }
= != <> ' " \$, ; : % ! & ? _ # <= >=

@

Formatting characters backspace, horizontal tab, vertical tab, form feed, and carriage return

Tokens

- A token is a group of characters that logically belong together. The programmer can write a program by using tokens.
- C++ uses the following types of tokens.
- Keywords, Identifiers, Literals, Punctuators, Operators.

1. Keywords

- These are some reserved words in C++ which have predefined meaning to compiler called keywords.
- Has predefined functionality
- C++ has 48 keywords
- Written in only in lower case

RESERVED KEYWORDS

delete	boolean	Break	Enum
case	volatile	Catch	Char
const	continue	Default	Do
else	asm	Extern	Union
float	for	Auto	Unsigned
if	inline	Register	Class
int	template	Long	Double
virtual	operator	Signed	goto
Protected	public	Sizeof	Return
Static	Struct	this	new
Friend	Throw	Typedef	private
try	Switch	while	short

2. Identifiers

- Symbolic names can be used in C++ for various data items used by a programmer in his program.
- A symbolic name is generally known as an identifier. The identifier is a sequence of characters taken from C++ character set.
- The rule for the formation of an identifier are:

- An identifier can consist of alphabets followed by letter and/or underscores.
- It must not start with a digit
- C++ is case sensitive that is upper case and lower case letters are considered different from each other.
- It should not be a reserved word.
- Sum, avg_ht ----- valid identifier
- 1oth std-no -----invalid identifier

3. Literals

- Literals (often referred to as constants) are data items that never change their value during the execution of the program. The following types of literals are available in C++.
 - Integer-Constants
 - Character-constants
 - Floating-constants
 - Strings-constants

Integer Constants

- Integer constants are whole number without any fractional part. C++ allows three types of integer constants.
- **Decimal integer constants** : It consists of sequence of digits and should not begin with 0 (zero). For example 124, - 179, +108.
- **Octal integer constants**: It consists of sequence of digits starting with 0 (zero).
- For example. 014, 012.
- **Hexadecimal integer constant**: It consists of sequence of digits preceded by ox or OX.

Character constants

- A character constant in C++ must contain one or more characters and must be enclosed in single quotation marks. For example 'A', '9', etc.
- C++ allows non graphic characters which cannot be typed directly from keyboard, e.g., backspace, tab etc. These characters can be represented by using an escape sequence.
- An escape sequence represents a single character.

Floating constants

- They are also called real constants. They are numbers having fractional parts.
- They may be written in fractional form or exponent form.
- A real constant in fractional form consists of signed or unsigned digits including a decimal point between digits.
- For example 3.0, -17.0, -0.627 etc.

String Literals

- A sequence of character enclosed within double quotes is called a string literal.
- String literal is by default (automatically) added with a special character '\0'
- which denotes the end of the string. Therefore the size of the string is increased by one character. For example "COMPUTER" will be represented as "COMPUTER\0" in the memory and its size is 9 characters.

LITERALS (Symbolic Constants)

- Using **const** qualifier
ex: `const int size=10;`
- Using **enum** keyword
ex: `enum{X,Y,Z};`
defines `const X=0;`
defines `const Y=0;`
defines `const Z=0;`

4. Punctuators

- The following characters are used as punctuators in C++.

Brackets [] - Opening and closing brackets indicate single and multidimensional array subscript.

Parentheses() - Opening and closing brackets indicate functions calls,; function parameters for grouping expressions etc.

Braces { } - Opening and closing braces indicate the start and end of a compound statement.

Comma , - It is used as a separator in a function argument list.

Semicolon ; - It is used as a statement terminator.

Colon : - It indicates a labeled statement or conditional operator symbol.

Equal sign = It is used as an assignment operator.

Pound sign # It is used as pre-processor directive.

5. Operators

- Operators are special symbols used for specific purposes. C++ provides six types of operators.
- Arithmetical operators,
- Relational operators,
- Logical operators,
- Unary operators,
- Assignment operators,
- Conditional operators,
- Comma operator

Structure of C++ Program



Include Files

Class Definition

Class Function Definition

Main Function Program

Simple C++ Program

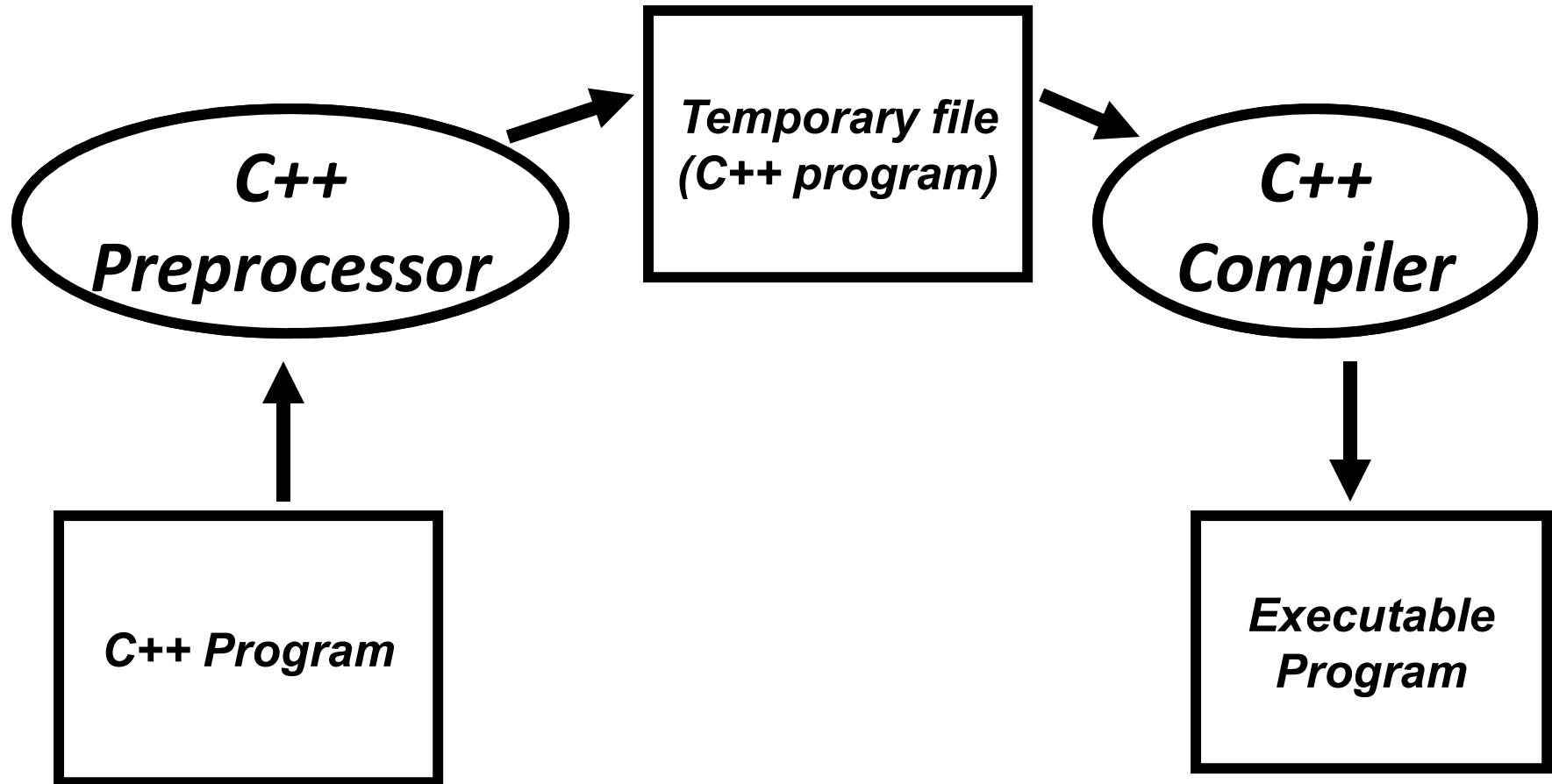
```
// Hello World program ← comment

#include <iostream.h> ← Allows access to an I/O
                       library

int main() { ← Starts definition of special function
              main()
    cout << "Hello World\n"; ← output (print) a
                              string

    return 0; ← Program returns a status
              code (0 means OK)
}
```

Preprocessing

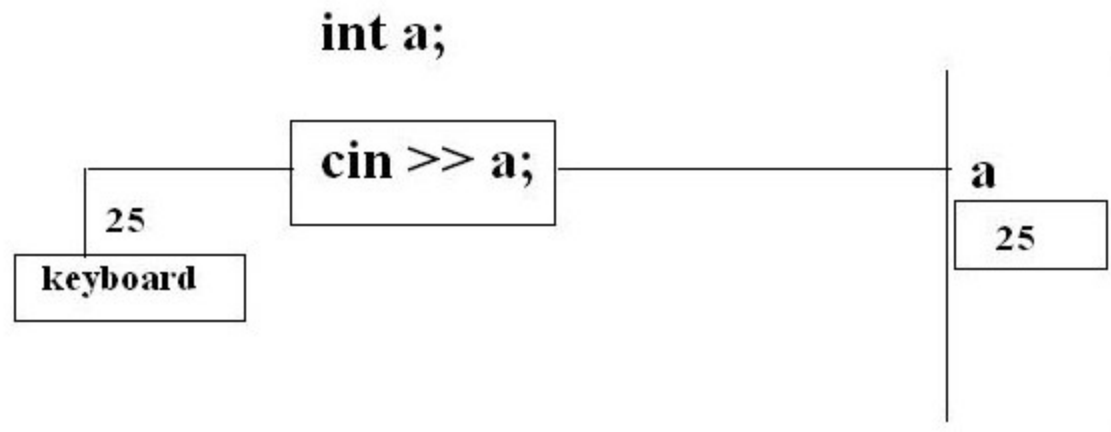


C++

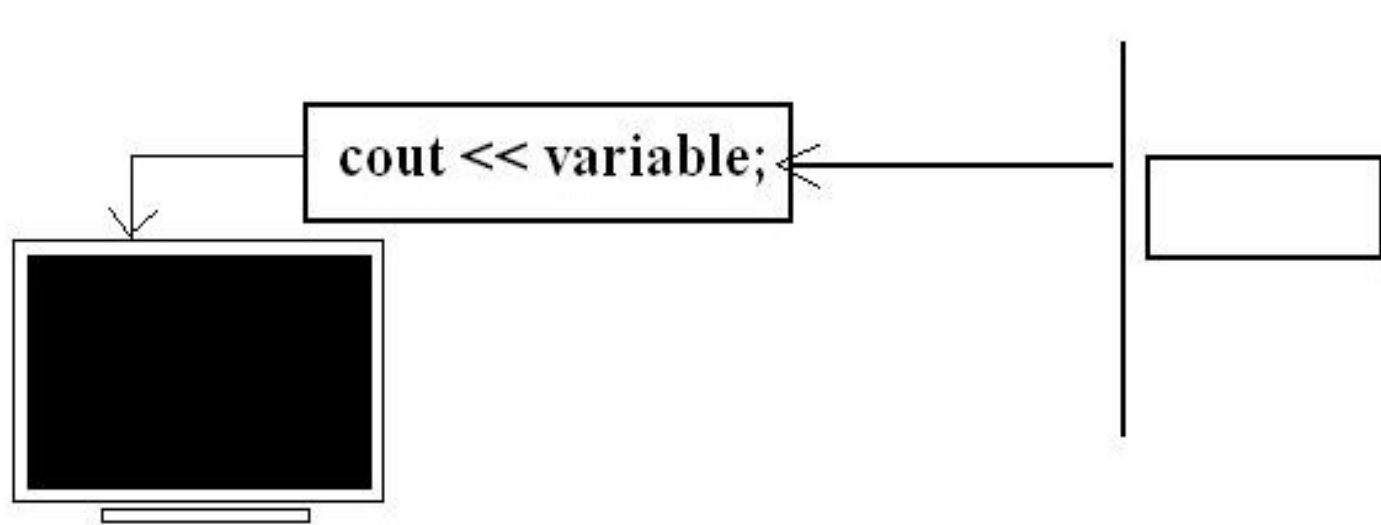
Input and output of c++

`cin >> variable ;`

example:



Output statement



```
cout << "Welcome to c++";
```



```
a = 100;  
cout << a;
```



```
a = 10;  
cout << "Value of a is " << a;
```



```
a = 10;  
b = 20;  
c = a + b;  
cout << "sum of " << a << " and " << b << "is" << c;
```

Sum of 10 and 20 is 30

```
a = 10;  
b = 20;  
cout << "Sum of 2 numbers = " << a + b;
```

expression

Operators

Types of operator

- Three types

- Unary – it operate on single operand

- example

- ++ increment -- decrement :: scope

- Binary – it operate on two operand

- Example

- Arithmetic, relational, logical, assignment, bitwise

- Ternary and Special Operator

Operators in C++

New operators in C++ :

- o << Insertion Operator
- o >> Extraction Operator
- o :: Scope Resolution Operator
- o :: * Pointer-to-member declaration
- o ->* Pointer-to-member selection
- o .* Pointer-to-member Operator
- o delete Memory Release Operator
- o endl Line Feed Operator
- o new Memory Allocation Operator
- o setw Field Width Operator

**All C operators are
valid in C++**

Scope Resolution Operator

C++ is a block structured language. The scope of a variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to be local to that block.

```
.....  
.....  
{  
    int x = 10;  
    .....  
    .....  
}  
.....  
.....  
{  
    int x = 1;  
    .....  
    .....  
}
```

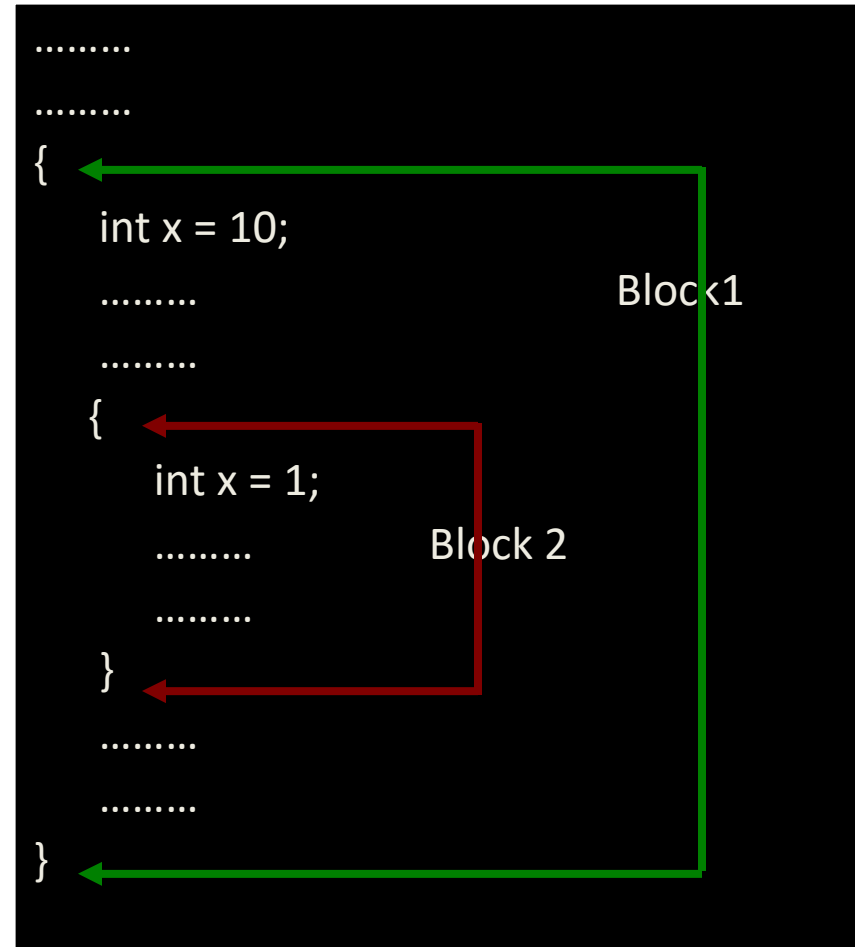

Scope Resolution Operator

continue...

Blocks in C++ are often nested.

In C, the global version of a variable can not be accessed from within the inner block.

C++ resolved this problem with the use of the scope resolution operator (`::`).



Scope Resolution Operator continue...

The scope resolution operator (`::`) can be used to uncover a hidden variable.

`::` variable-name

This operator allows access to the global version of a variable.

Scope Resolution Operator continue...

```
#include<iostream.h>
#include<conio.h>
int m = 10;
void main( )
{
    int m = 20;
    clrscr( );
    cout << "m_local      = " << m << "\n";
    cout << "m_global    = " << ::m << "\n";
    getch( );
}
```

m_local = 20

m_global = 10

Memory Dereferencing Operator

- C++ allow access of class member through pointer. This is achieved by following operator
 - :: * is used to declare a pointer to a member of a class
 - >* is used to declare a member using object name and a pointer to that member
 - . * is used to access a member using pointer to the object and to the member

Memory Management Operators

malloc() and **calloc()** functions are used to allocate memory dynamically at run time.

The function **free()** to free dynamically the allocated memory.

} C
&
C++

The unary operators **new** and **delete** perform the task of allocating and freeing the memory.

} C++

Memory Management Operators

continue...

- **new** to create an object
- **delete** to destroy an object

A data object created inside a block with **new**, will remain in existence until it is explicitly destroyed by using **delete**.

Thus the life time of an object is directly under our control and is unrelated to the block structure of the program.

Memory Management Operators

continue...

- **new** to create an object

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

The *data-type* may be any valid data type

pointer-variable is a pointer of type *data-type*

The **new** operator allocates sufficient memory to hold a data object of type *data-type* and returns the address of the object

The pointer-variable holds the address of the memory space allocated

Memory Management Operators

continue...

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

```
p = new int;    // p is a pointer of type int
```

```
q = new float; // q is a pointer of type float
```

Here p and q must have already been declared as pointers of appropriate types.

Alternatively, we can combine the declaration of pointers and their assignments as:

```
int *p = new int;
```

```
float *q = new float;
```


Memory Management Operators

continue...

```
int *p = new int;
```

```
float *q = new float;
```

```
*p = 25; // assign 25 to the newly created int object
```

```
*q = 7.5; // assign 7.5 to the newly created float object
```

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

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

```
float *q = new float ( 7.5 );
```

Memory Management Operators

continue...

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

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

```
int *p = new int [10];
```

When creating multi-dimensional arrays with new, all the array sizes must be supplied.

```
array_ptr = new int[3][5][4]; //legal
```

```
array_ptr = new int[m][5][4]; //legal
```

```
array_ptr = new int[3][5][ ]; //illegal
```

```
array_ptr = new int[ ][5][4]; //illegal
```

Memory Management Operators

continue...

- o **delete** to destroy an object

```
delete pointer-variable;
```

When a data object is no longer needed, it is destroyed to release the memory space for reuse.

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

```
delete [ size ] pointer-variable;
```

The size specifies the number of elements in the array to be freed.

```
delete [ ]p; // delete the entire array pointed to by p
```

Memory Management Operators

continue...

If sufficient memory is not available for allocation, `malloc()` and `new` returns a null pointer

```
.....  
.....  
p = new int;  
if (!p)  
{  
    cout << "Allocation failed \n";  
}  
.....  
.....
```

Memory Management Operators

continue...

Advantages of `new` over `malloc()`:

- It automatically computes the size of the data object. No need to use `sizeof` operator.
- It automatically returns the correct pointer type. No need to use type cast.
- It is possible to initialize the object while creating the memory space.
- Like any operator, `new` and `delete` can be overloaded.

Manipulators

- C++ provide a set of function called manipulator that can be used to control the appearance of the output
- To use these manipulator the file `iomanip.h` must be included in program
- Manipulator can be classified into two categories
- Parameterized and non parameterized

- Parameterized manipulator contain a single argument where as non parameterized does not contain any argument
- Manipulator are always used with cout statement
- A combination of ios function and manipulator can be used in a program

Manipulators

Manipulators are operators that are used to format the data display.

Commonly used manipulators are:

- o `endl` // causes a line feed when used in an
Non P // output statement
- o `setw` // to specify field width and force the
P // data to be printed right-justified

Manipulators

continue...

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main( )
{
    int m, n, p;
    m = 2597;
    n = 14;
    p = 175;
    clrscr( );

    cout <<setw(10) << "First = " <<setw(10) << m << endl
         <<setw(10) << "Second = " << setw(10) << n << endl
         <<setw(10) << "Third = " << setw(10) << p << endl;
    getch( );
}
```

First = 2597

Second = 14

Third = 175

Manipulators

continue...

We can create our own manipulators as follows:

```
# include < ostream.h>
ostream & symbol (ostream & output)
{
    output << "\tRs. ";
    return output;
}
```

Manipulators

continue...

```
#include<iostream.h>
#include<conio.h>

ostream & symbol (ostream & output)
{
    output << "Rs. ";
    return output;
}

void main( )
{
    clrscr( );

    cout << symbol << "5,000/-" <<endl;
    getch( );
}
```

Type Cast Operator

C++ permit explicit type conversion of variables or expressions using the type cast operator.

- (type-name) expression // C notation
- type-name (expression) // C++ notation
// like a function call
// notation

eg:- average = sum /(float) i; // C notation

average = sum / float(i); // C++ notation

Type Cast Operator

continue...

```
p = int * ( q ); // is illegal
```



The type name should be an identifier

```
p = ( int * ) q; // is legal
```

Alternatively, we can use typedef to create an identifier of the required type.

```
typedef int * int_pt;
```

```
p = int_pt ( q );
```

Introduction

- A string is a sequence of character.
- We have used null terminated `<char>` arrays (C-strings or C-style strings) to store and manipulate strings.
- ANSI C++ provides a class called **string**.
- We must include `<string>` in our program.

C++ Strings

- Supports **C-String**
- C++ defines a string class called **string**
- The string class supports several constructors. The prototypes of commonly used one is
 - `string();`
 - ex: `string str("Alpha");`

C++ Strings

Operator	Meaning
=	Assignment
+	Concatenation
+=	Concatenation assignment
==	Equality
!=	Inequality
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
>>	Reads
<<	Prints

C++ Strings

- `str2 = str1; // assigning a string`
- `str3 = str1 + str2; //concatenating strings`
- `if(str2 > str1) cout<<" str2 is bin";//compares`
- `str1 = "This is a null-terminated string.\n";`
- `cin>>str1; //reads`
- `Cout<<str1; //prints`

Initializing string variables

- In C, a string can be a specially terminated char array or char pointer
 - a char array, such as `char str[]="high";`
 - a char pointer, such as `char *p="high";`
- If a char array, the last element of the array must be equal to `'\0'`, signaling the end
- For example, the above `str[]` is really of length 5:
`str[0]='h' str[1]='i' str[2]='g' str[3]='h' str[4]='\0'`
- The same array could've been declared as:
 - `char str[5] = {'h','i','g','h','\0'};`
- If you write `char str[4] = {'h','i','g','h'};`, then `str` is an array of chars but not a string.
- In `char *p="high";` the system allocates memory of 5 characters long, stores "high" in the first 4, and `'\0'` in the 5th.

Declaration of strings

- The following instructions are all equivalent. They declare `x` to be an object of type `string`, and assign the string “high school” to it:
 - **string** `x`(“high school”);
 - **string** `x`= “high school”;
 - **string** `x`; `x`=“high school”;

Operations on strings (Concatenation)

- Let x and y be two **strings**
- To concatenate x and y , write: $x+y$

```
string x= "high";  
string y= "school";  
string z;  
z=x+y;  
cout<<"z="<<z<<endl;  
z =z+" was fun";  
cout<<"z="<<z<<endl;
```

Output:
z=highschool

z= highschool was fun

C ++ Strings

- **Formatted Input:** Stream extraction operator
 - `cin >> stringObject;`
 - the extraction operator `>>` formats the data that it receives through its input stream; it skips over whitespace
- **Unformatted Input:** `getline` function for a **string**
 - `getline(cin, s)`
 - does not skip over whitespace
 - delimited by newline
 - reads an entire line of characters into `s`

```
string s = "ABCDEFGH";
```

```
getline(cin, s); //reads entire line of characters into s
```

```
char c = s[2]; //assigns 'C' to c
```

```
s[4] = '*'; //changes s to "ABCD*FGH"
```

Getline() function

- C++ provides the function `getline()` to efficiently read a line of text

```
cin.getline(text, size)
```

Here `text` is the string variable

`Size` is the number of character in the string

Write() function

- It is used to display an entire line of text
`cout.write(text, size)`

get and getline Member Functions

- **cin.get()** : inputs a character from stream (even white spaces) and returns it
- **cin.get(c)** : inputs a character from stream and stores it in **c**

Get() and put() function

- It is used to get() and put() single character of input and output respectively
- Char ch;

```
cin=get(ch);
```

 get a character from the keyboard and assign it to variable ch
- Char ch;

```
ch=cin.get();
```

 it perform the same task of assigning the input character to ch

- The function `put()` is used to output a character

```
cout.put(ch);
```

Example

```
Char ch;
```

```
Ch=cin.get();
```

```
Cout.put(ch);
```

Function in C++

Dr.T.Logeswari

INTRODUCTION

- Functions are the building blocks of C++ programs where all the program activity occurs.
- Function is a collection of declarations and statements.

C++ Functions

- “**Set of program statements** that can be processed independently.”
- Like in other languages, called **subroutines** or **procedures**.

Advantages ...?

- Elimination of redundant code
- Easier debugging
- Reduction in the Size of the code
- Leads to reusability of the code

Function

- The following program illustrates the use of a function :

```
//to display general message using function
```

```
#include<iostream.h>
```

```
include<conio.h>
```

```
void main()
```

```
{
```

```
void disp(); //function prototype
```

```
clrscr(); //clears the screen
```

```
disp(); //function call
```

```
getch(); //freeze the monitor
```

```
}
```

```
//function definition
void disp()
{
cout<<"Welcome to II BCA";
cout<<"Programming is nothing but logic
  implementation";
}
```


FUNCTION DEFINITION AND DECLARATION

- The general syntax of a function definition in C++
data type name_of_the_function (argument list)
{
 //body of the function
}

Here, the data type specifies the type of the value to be returned by the function. It may be any valid C++ data type.

When no type is given, then the compiler returns an integer value from the function.

- Name_of_the_function is a valid C++ identifier (no reserved word allowed) defined by the user and it can be used by other functions for calling this function.
- Argument list is a comma separated list of variables of a function through which the function may receive data or send data when called from other function.
- When no parameters, the argument list is empty

```
//function definition add()
void add()
{
int a,b,sum;
cout<<"Enter two integers"<<endl;
cin>>a>>b;
sum=a+b;
cout<<"\nThe sum of two numbers is
"<<sum<<endl;
}
```

- The above function add () can also be coded with the help of arguments of parameters as shown below:

```
//function definition add()
void add(int a, int b) //variable names are must
                        in definition
{
    int sum;
    sum=a+b;
    cout<<"\nThe sum of two numbers is
"<<sum<<endl;
}
```

Calling a function

- A function can be called by specifying its name, followed by a list of arguments enclosed in parentheses
- The argument may be separated by commas
- Syntax

```
variable = function_name(argument list);
```

or

```
function_name(argument-list);
```

```
float cubeRoot(float); // Function prototype
void main()
{
    float a,b;
    clrscr();
    cout<<"/n type a number to find cube root:";
    cin>>a;
    cout<<"/n cube root of"<<a<<"is"<<cubeRoot(a);
}
Float cubeRoot(float x)
{
    return(pow(x,(1.0/3.0))); //return(exp(1.0/3.0*log(x)))
}
```

ARGUMENTS TO A FUNCTION

- Arguments(s) of a function is (are) the data that the function receives when called/invoked from another function.

Passing Constant

- Consider a function that will print any character any number of times
- The calling program supplies constant argument, such as '*' and 50 to the function
- The variable used within the function to hold the argument values are called parameter
- In line() they are ch and len
- When the function is called its parameter are automatically initialized to the value passed by calling function


```
Void line(char, int); // function declaration
```

```
Void main()
```

```
{
```

```
Clrscr();
```

```
line('*',50);
```

```
Line('!',50);
```

```
Line('-',40);
```

```
Getch();
```

```
}
```

```
//function to draw a line
```

```
Void line(char ch, int len)
```

```
{
```

```
For(int i=1;i<=len;i++)
```

```
    cout<<ch;
```

```
Cout<<endl;
```

```
}
```

Program to illustrate function with constant argument

Output

```
*****
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
-----
```

Passing variables

- The previous program passing constant now here passing variables as arguments

```
Void line(char, int); // function declaration
```

```
Void main()
```

```
{
```

```
Char ch1;
```

```
Int len;
```

```
Clrscr();
```

```
Cout<<"enter a character:";
```

```
Cin>>ch1;
```

```
Cout<<"enter the number of times to repeat:";
```

```
Cin>>len;
```

```
Getch();
```

```
}
```

```
// function to draw line
Void line(char ch, int n)
{
for(int i=1;i<=n;i++)
{
Cout<<ch;
Cout<<endl;
}
```

Output

Enter a character:@

Enter no of times to repeat:10

@ @ @ @ @ @ @ @ @ @

Ch1 and len in main() are used
as arguments to line()

Actual Arguments

- The arguments appearing in the function call are known as actual arguments
- It may be constant, variable or an expression
- In function call:

Result = mul(a,b); a and b are the actual arguments

The actual arguments must have some value stored in function call

Ex : a=10; b=5;

result = mul(a,b); or

Cin>>a>>b;

Result = mul(a,b);

Formal parameter

- The argument that appear in the function header are referred to as formal parameter
- It get their values from the calling function

CALLING FUNCTIONS

- In C++ programs, functions with arguments can be invoked by :
 - (a) Value
 - (b) Reference

Call by Value:

In this method the values of the actual parameters (appearing in the function call) are copied into the formal parameters (appearing in the function definition), i.e., the function creates its own copy of argument values and operates on them

Difference

Pass by value	Pass by reference
Calling function send copies of data	Calling
The formal parameter are ordinary variable	The formal parameter are pointer variables
Result are send calling function through return mechanism(one)	Several result can sent back to calling function
Actual parameter are unaffected by changes made within the function	Direct changes made to the actual parameter

Pass by reference with reference argument

- When parameter passed by reference, the formal parameter become aliases to the actual parameter in the calling function

```
Void swap(int &, Int &);
```

```
Void main()
```

```
{  
}
```

```
Void swap(int &a, int &b)
```

```
{  
}
```

This mechanism only available in c
It permit the manipulation of obje
By reference and eliminate copyin
Of parameter from calling function
To called function

Comparison

Reference argument	Pointer Arguments
A reference variable is always be initialized. Once initialized it cannot be made refer to another object	A pointer can point to different object
A reference argument is prefixed with an ampersand(&) symbol	A pointer argument is prefixed with asterisk(*) symbol
It can be used to implement overloaded operator	It is not used to implement overloaded operator

Passing array to function

- Constant and variable are passed to the function as argument
- Sometime the entire array of function should pass
- It is not possible to pass a block of memory to the function. Instead a reference(address) of the first element of the array can be passed to the function
- The array element are stored in contiguous memory location, reference to the first element gives access to all the element of the array

- C++ convert array declaration in the formal parameter of a function, into an array pointer, the size of the array need not be specified
- Program –: rev - string passes two character array str[] and rev[]

```
void reverse(char s[], char r[]);
```

```
Void main()
```

```
{
```

```
Char str[50], rev[50];
```

```
Clrscr();
```

```
Cout<<"enter a string";
```

```
Cin.getline(str,50);
```

```
Cout<<"the original string"<<str
```

```
Reverse(str, rev);
```

Getline() function is used to read a line of text

Syntax:

```
Cin.getline(text,size);
```

```
Cout<<reverse string<<rev;
}
Void reverse(char s[], char[])
{
Int len = strlen(s);
For(int i=0; i<len;i++)
r[i]=s[len-i-1];
r[i]='\0';
}
```

The array are passed to function using pass by value

Passing & returning structure variable

- Passing structure to function works like pass by value ie the function works with copy of the structure.
- The function can also return the structure. To pass the address of the structure. The address operator(&) must be used

Struct clock

```
{  
Int hr;  
Int min;  
};  
Clock sumtime(clock t1, clock t2);  
Void showsum(clock t);  
Void main()  
{  
Clock t3;  
Clock t1={5,50};  
Clock t2={3,45};  
Clrscr();  
t3=sumtime(t1,t2);  
Cout<<first clock time;  
Showsum(t1)
```

```
Cout<<second clock;
Showsum(t2);
Cout<<third clock;
Showsum(t3);
getch();
}
Clock sumtime(clock t1, clock t2)
{
Clock total;
total.min= (t1.min+t2.min)%60;
total.hr=(t1.hr+t2.hr)+(t1.min+t2.min)/60;
return total;
}
```



```
Void showsum(clock t)
```

```
{
```

```
Cout<<t.hr<<"hour"<<t.min<<"minutes";
```

```
}
```

Output

First clock time : 5 hours, 50 minutes

Second clock time: 3 hours, 45 minutes

Sum of two clock: 9 hours, 35 minutes

DEFAULT ARGUMENTS

- C++ allows a function to assign a parameter the default value in case no argument for that parameter is specified in the function call.
- Default values are assigned in function declaration(prototype). Example

```
float si(float amount, int period, float rate= 1.10)
```

```
#include<iostream.h>
int calc(int U)
{
    If (U % 2 == 0)
        return U+10;
    Else
        return U+2
}
Void pattern (char M, int B=2)
{
    for (int CNT=0;CNT<B; CNT++)
        cout<calc(CNT) <<M;
        cout<<endl;
}
Void main ()
{
    Pattern('*');
    Pattern ('#,4)
    Pattern (;@,,3);
}
```

CONSTANT ARGUMENTS

- A C++ function may have constant arguments(s). These arguments(s) is/are treated as constant(s). These values cannot be modified by the function.
- For making the arguments(s) constant to a function, we should use the keyword `const` as given below in the function prototype :
- `Void max(const float x, const float y, const float z);`

- Here, the qualifier `const` informs the compiler that the argument(s) having `const` should not be modified by the function `max()`. These are quite useful when call by reference method is used for passing arguments.

Return statement

- This statement is normally used to send back values from the sub program or function to the calling function or main program
- A function can return only one value or none, for every function call
- General form

Return 0;

Or

Return(expression)

- It is not compulsory in a function most function use it, either to stop execution or return a value to the calling function
- A function can have several return statement

Return by reference

- A function can be called by reference or return a reference
- When a function return a reference to a data object, the object must be existing even after the function terminates.
- To implement this function return a reference to an argument that passed to it

```
float& maxi(float, float, float float&);  
Void main()  
{  
float main()  
{  
float x,y,z large;  
clrscr();  
Cout<<"type 3 no";  
Cin>>x>>y>>z;  
Maxi(x,y,z,large);  
Cout<<"given value of x y and z";  
Cout<<"x=x"<<x;  
Cout<<"y=y"<<y;  
Cout<<"z=z"<<z;  
Cout<<"largest is"<<large;  
getch();  
}
```



```
float& maxi(float a, float b, float c, float& big)
{
big=a;
If (b>big) big =b;
If(c>big) big = c;
return big;
}
```

output

Recursive function

- Recursive in a process by which a function call itself repeatedly until some condition is satisfied
- The process is used for repeated calculation, where each action in terms of a previous result

```
    fun(int x)
    {
    ....
    ....
    y=fun(x);
    ...
    }
```

- To write a recursive function, two conditions must be satisfied
 - The program must be in recursive form
 - It must include a terminating condition

Eg

$$n! = n * (n - 1) !$$

Function overloading

- In c++ polymorphism (poly means many and morp means form) is implemented using
 - Operator overloading
 - Function overloading
- Overloading refer to the capability of using a single name for several different task
- The function overloading the same name is used for many function which vary in their list of argument and data types and perform difference action

Inline Functions

“ Inline functions are those whose **function body** is inserted **in place** of the **function call** statement during the compilation process.”

- **Syntax:**

```
inline return_dt func_name(formal parameters)
{
    function body
}
```

Inline Functions

- **Frequently** executed interface functions.
- Expanding **function calls** inline can produce **faster** run times.
- Like the **register** specifier, **inline** is actually just a *request, not a command*, to the compiler.

- The inlining does not work for the following situations :
 1. For functions returning values and having a loop or a switch or a goto statement.
 2. For functions that do not return value and having a return statement.
 3. For functions having static variable(s).
 4. If the inline functions are recursive (i.e. a function defined in terms of itself).

4. Classes & Objects

Introduction

- **The New C++ Headers(New style)**

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

- **The old style Headers**

```
#include<iostream.h>
```

The New C++ Headers

- A *namespace* is simply a declarative region.
- The purpose of a namespace is to localize the names of **identifiers** to avoid name collisions.
- `iostream`, `math`, `string`, `fstream` etc., forms the **contents** of the namespace called **std**.

Class Specification

- **Syntax:**

```
class class_name
```

```
{
```



Data members



Members functions

```
};
```

Class Specification

- **class Student**

{

int st_id;

char st_name[];

void read_data();

void print_data();

};



Data Members or Properties of Student Class



Members Functions or Behaviours of Student Class

Class Specification

- **Visibility of Data members & Member functions**
 - public** - accessed by member functions and all other non-member functions in the program.
 - private** - accessed by only member functions of the class.
 - protected** - similar to private, but accessed by all the member functions of immediate derived class
 - default** - all items defined in the class are private.

Class specification

- **class** Student

{

int st_id;

char st_name[];

void read_data();

void print_data();

};



**private / default
visibility**

Class specification

- **class Student**

{

public:

int st_id;

char st_name[];

public:

void read_data();

void print_data();

};



public visibility

Class Objects

- **Object Instantiation:**

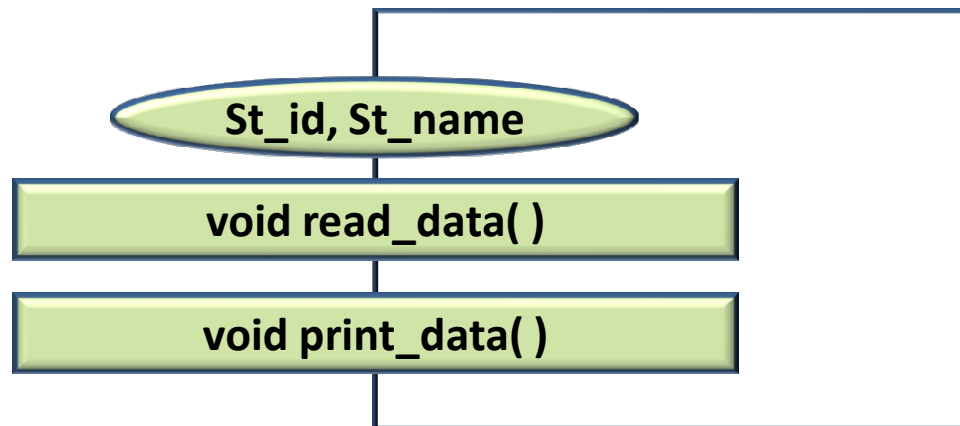
The process of creating object of the type class

- **Syntax:**

class_name obj_name;

ex: Student st;

← Creates a single object of the type Student!



Class Object

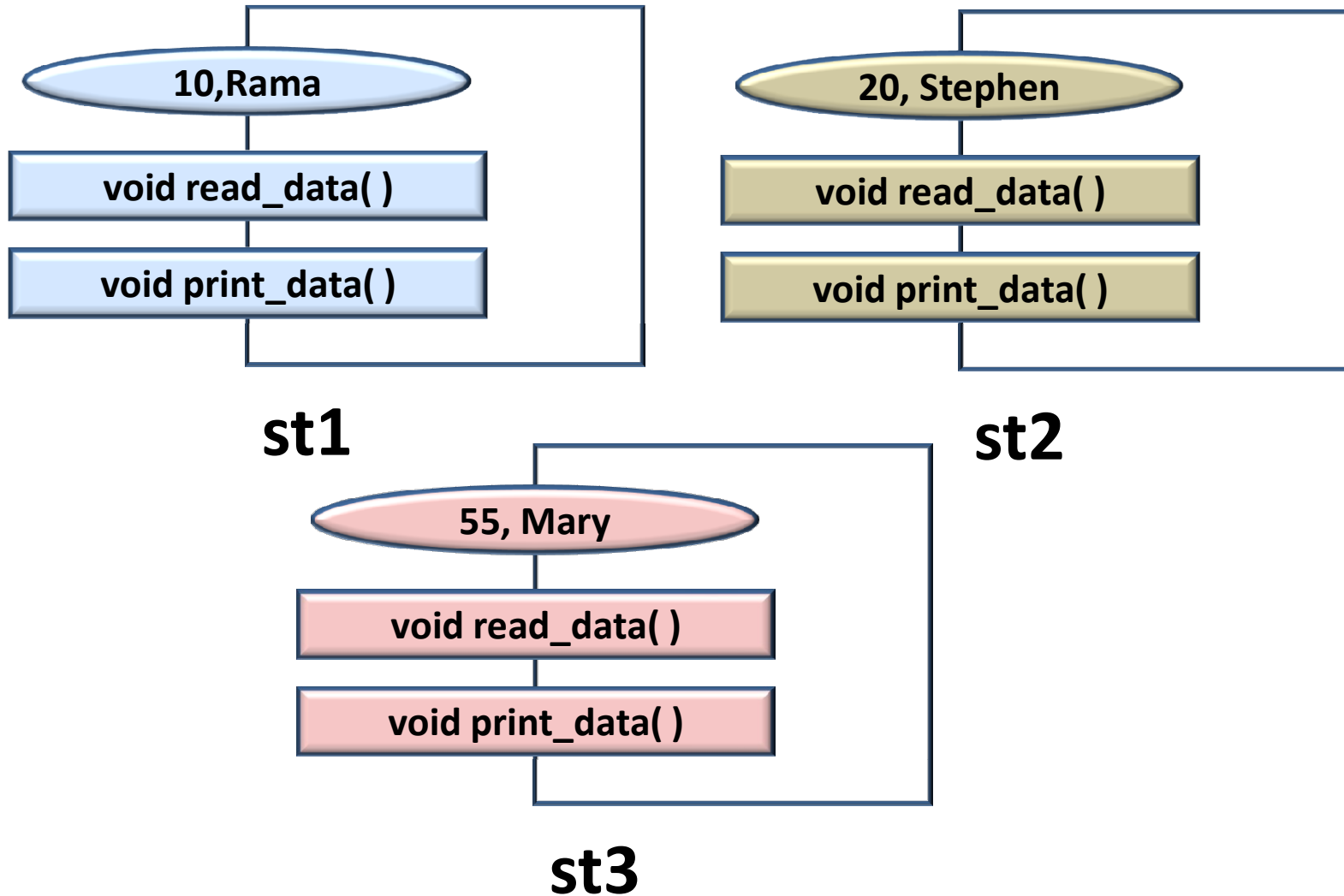
- **More of Objects**

ex: Student st1;

Student st2;

Student st3;

Class Objects



Accessing Data Members

(outside the class)

- **Syntax: (single object)**

obj_name . datamember;

ex: st.st_id;

- **Syntax:(array of objects)**

obj_name[i] . datamember;

ex: st[i].st_id;

Accessing Data Members

(inside the class member function)

- **Syntax: (single object)**

data_member;

ex: st_id;

- **Syntax:(array of objects)**

data_member;

ex: st_id;

Defining Member Functions

- **Syntax :(Inside the class definition)**

```
ret_type fun_name(formal parameters)
{
    function body
}
```

Defining Member Functions

- **Syntax:(Outside the class definition)**

```
ret_type class_name::fun_name(formal parameters)
{
    function body
}
```

Nesting of member Function

- A member function of class can be called in two ways
 - Called by an object of that class using the dot operator
 - Called inside of the another member function of the class
- when a member function Called inside of the another member function of the same class it is known as nesting of member function

Class large

{

Private:

int m,n;

Public:

int getdata();

void putdata();

Void findlarge();

};

Void large::getdata()

{

cout<<"enter two integer";

cin>>m>>n;

}


```
Int large :: findlarge()
```

```
{
```

```
    If(m>n)
```

```
    return (m);
```

```
    else
```

```
    return(n);
```

```
}
```

```
Void large :: putdata()
```

```
{
```

```
    Cout<<"given number"<<"m"<<"n"<<endl;
```

```
    Cout<<"largest"<<"findlarge());
```

```
Void main()
```

```
{
```

```
clrscr();
```

```
large x;
```

```
x.getdata();
```

```
x.putdata();
```

```
getch();
```

```
}
```

Output

Enter two integer : 50 100

Largest :100

Arrays as class members

- The data member of a class can be array.
- Example to store marks obtained by a student in 5 different subject

```
const int size=10;
```

```
class student
```

```
{
```

```
int roll; int marks [m]; char name[20];
```

```
public:
```

```
void getdata(); void calsum();
```

```
void displayAll(void);
```

```
};
```

```
const int size=5;
class student
{
private:
int roll; int marks [m]; char name[20];
public:
void getdata(); void calsum();
void displayAll(void);
};
```

```
Void student :: getdata()
{
Cout<<"enter roll number";
Cin>>roll;
Cout<<"enter the name:";
Cin>>name
```

```
Cout<<"enter marks"<<m<<"subject";  
for(int i=1;i<=m;i++)  
Cin>>marks[i];  
}  
Int student ::calsum()  
{  
Int total =0;  
for(int i=1;i<=m;i++)  
total = total +marks[i];  
return(total);  
}
```

```
Void student :: dispalyAll()
{
Cout<<"roll no"<<roll;
Cout<<"name"<<name;
Cout<<"marks";
for(int i=1;i<=m;i++)
Cout<<marks[i]<<setw(5);
Cout<<"total"<<calsum();
}
Void main()
{
Student s;
s.getdata();
s.displayAll();
}
```

```
Enter roll no : 1
Enter name : banu
Enter marks in 5 subject : 20 20 40 50 10

Roll no :1
Name : banu
Marks: 20 20 40 50 10
Total : 140
```

Array of object

- Array can be any data type, it is legal to have arrays of variable that are of type class
- Such arrays are called arrays of object

Class item

{

Private:

Int code; char name[20]; float rate;

Public:

Void getdata(); void print data();

};

- Item is the user defined type. It is used to create array of object such as
item pc[3];
- Item printer[10];
- Individual element of the array can be accessed by the usual array accessing methods and member function can be accessed by using dot member operator
 - Eg pc[1].printdata();

Class item

{

Private:

Int code;

Char name[25];

Float rate;

Public:

Void getdata()

{

Cout<<"enter item name",

Cin>>name;

Cout<<"enter item code";

Cin>>code;

Cout<<"enter item price";

Cin>>rate;

```
Void print data();
{
Cout<<"item code"<<code;
Cout<<"item name"<<name;
Cout<<"item rate"<<rate;
}
};
Const int size =3;
Void main()
{
Item pc[size];
```

```
for(int i=0;i<size;i++)
pc[i].getdata();
Cout<<"details of pc";
for(i=0;i<size;i++)
pc[i].printdata();
getch();
}
```

Output

Passing Objects as Arguments

- Passing of parameter as object can be done in two ways
- Objects are passed to functions through the use of the standard **call-by-value** mechanism.
- Means that a **copy of an object** is **made** when it is passed to a function.

- Call by value
- A copy of the entire object is passed to the function.
- This method is known as pass by value.
- Any changes made to the object inside the function, do not affect the object used to call the function

- **Pass by reference**

- It is similar to call by reference can be used.
- In this method only address of the object is passed to the called function, so any changes made to the object inside the function are reflected in the actual object.

Passing Objects as Arguments

```
class complex
{
    .....
    .....
    void Add(int x, complex c);
    .....
    .....
};
```

```
void main()
{
    complex obj, s1;
    .....
    .....
    .....
    obj.Add(6, s1);
    .....
    .....
}
```




Returning Objects

- A function may return an object to the caller.

```
class complex
{
    .....
    .....
    complex Add(int x, complex c);
    .....
    .....
};
```

```
void main()
{
    complex obj, s1;
    .....
    .....
    .....
    obj=obj.Add(6, s1);
    .....
    .....
}
```



Returning object from function

- This is like normal function, a function can return object too.

Class complex

```
{  
    float real;  
    float imag;  
Public:  
Void readdata()  
{  
    Cout<<"enter the real part";  
    Cin>>real;  
    Cout<<"enter the imaginary part";  
    Cin>>img;  
}
```

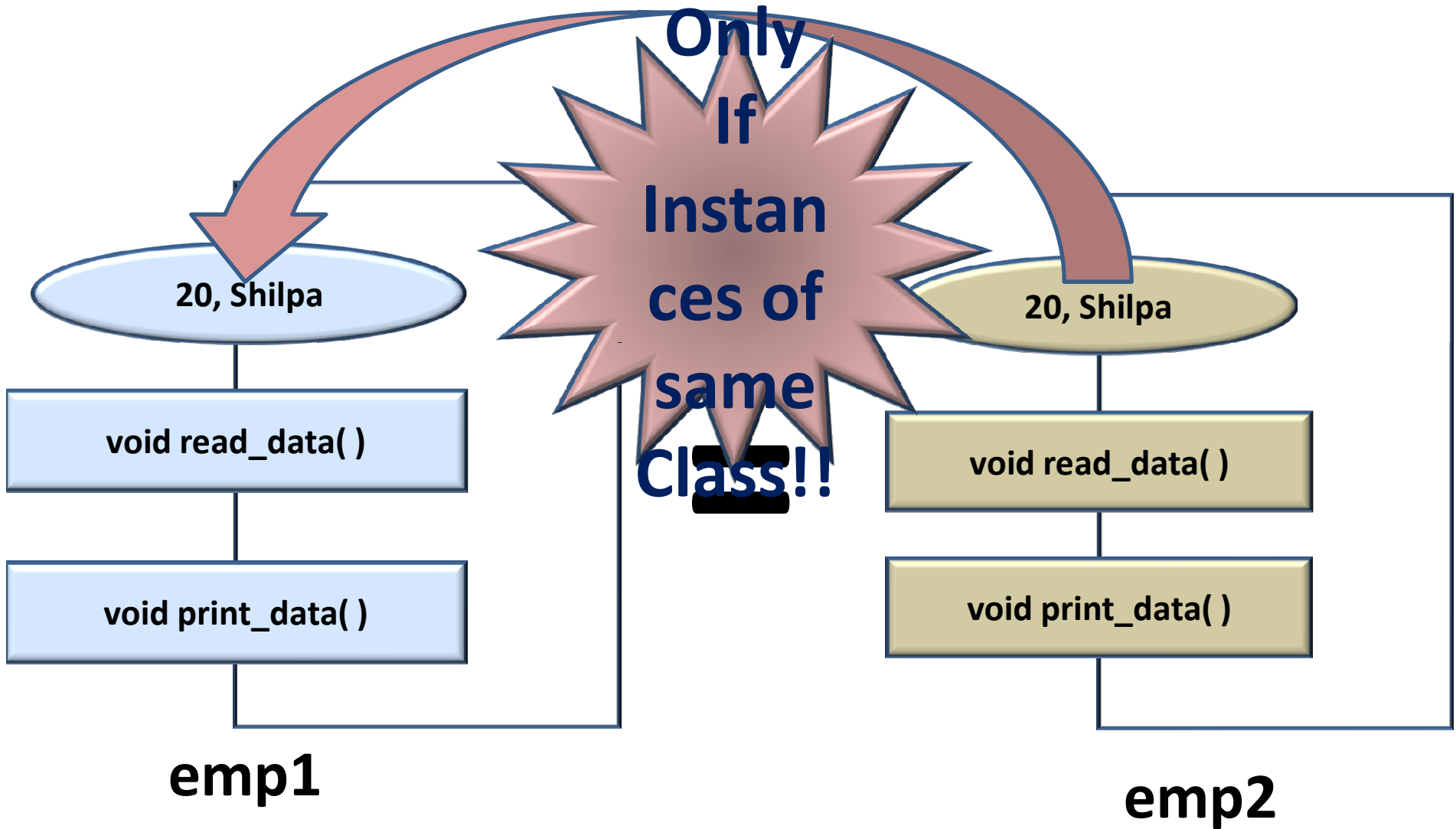
```
Void printdata()
{
Cout<<"real part"<<real<<endl;
Cout<<"imaginary part"<<imag<<endl;
}
Complex sum(Complex c2)           //temp variable to hold sum
                                   of two number
{
Complex temp;
temp.real = real +c2.real;
temp.imag= imag+c2.imag;
return(temp);
}
};
```

```
Void main()
{
Clrscr():
Complex c1, c2, c3;
C1.readdata();
C2.readdata();
c3=c1.sum(c2);
Cout<<"the first complex number";c1.printdata();
Cout<<"the secocnd complex number";c2.printdata();
Cout<<" Sum of two complex number";c3.printdata();
}
```

Assigning object

- An object can be assigned to another object, only if both object are of same type
- On assignment, the data member of the object will be copied into the corresponding member of the other object

Object Assignment



Class complex

{

float real;

float imag;

Public:

Void input complex()

{

Cout<<"type real part";

Cin>>real;

Cout<<"type the imaginary part";

Cin>>img;

}

```
Void outputcomplex()
{
Cout<<"real part"<<real<<endl;
Cout<<"imaginary part"<<imag<<endl;
}
};
Void main()
{
Clrscr();
Complex c1,c2;
Cout<<"input first complex number"<<endl
C1.inputcomplex();
Cout<<"first complex number";c1.outputcomplex();
C2=c1;
Cout<<"second complex number";c2.outputcomplex();
}
```

Memory allocation for object

- When member function is defined, it is created and placed in memory.
- When an object is created only the space is allocated for the member variable
- So object of the class use same member function , no separate space is allocated for the member function during creation of object
- So data is therefore placed in memory whenever each object is defined

Static class member

- In a class, both member variable and function can be declared as static
- A static data member of a class is similar to static variable in c but with following characteristics
 - It is like a global variable in for its class, available to all object of that class
 - When the first object of its class is created it is initialized to zero automatically

- When a static data member is declared as private, the non member function cannot access it.

- Declaration

Class info

{

Static int count; // declaration within class

.....

.....

};

- Defined outside the class

```
Int info::count; // definition outside the class  
                 named info
```

- A static variable can be initialized with any value at the time of its definition outside the class

```
Int info :: count = 15;
```

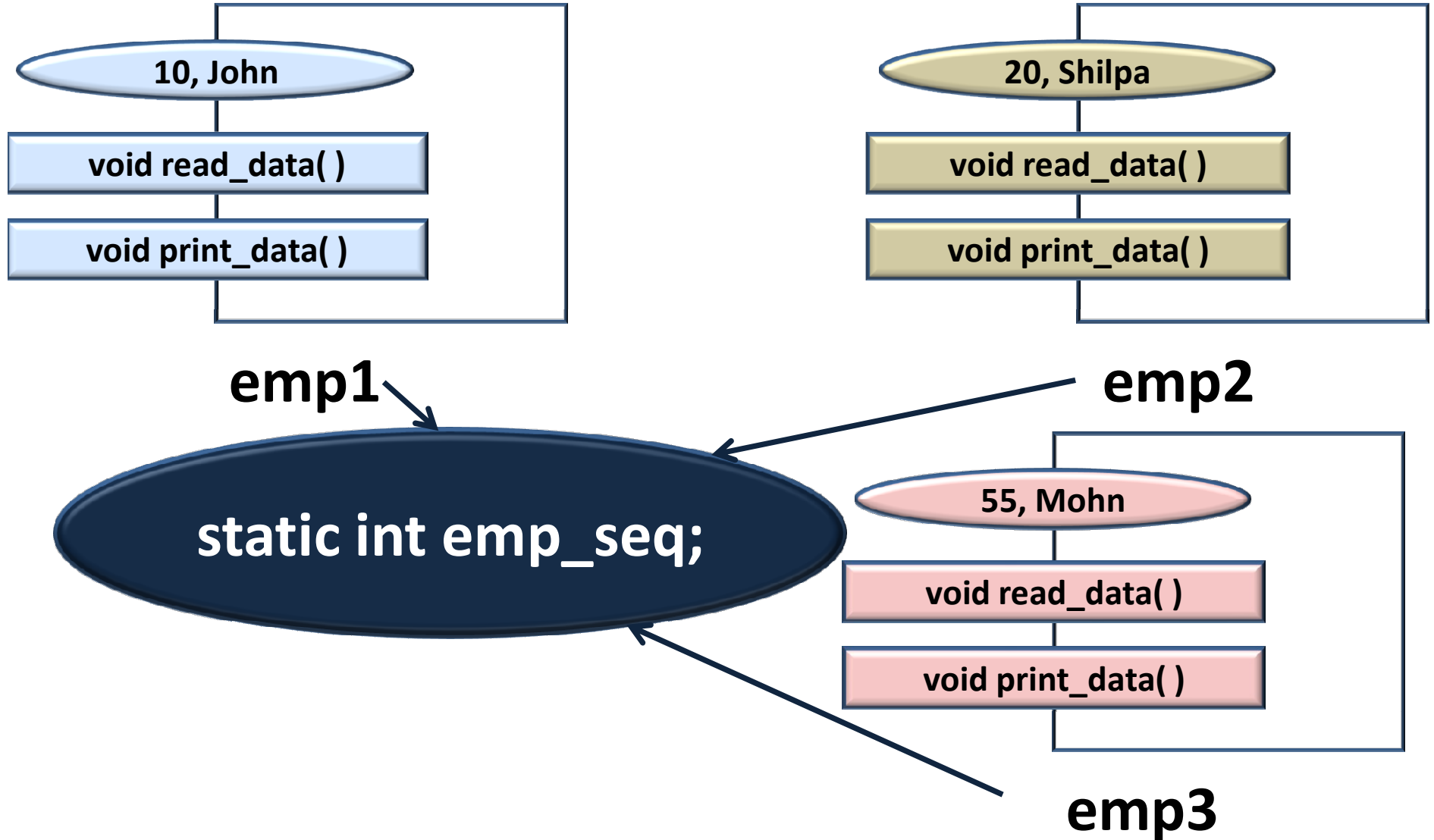
Static Data Members

- **Static data members** of a class are also known as "**class variables**".
- Because their **content** does **not depend** on **any object**.
- They have only **one unique** value for **all** the objects of that same class.

Static Data Members

- Tells the compiler that **only one copy** of the variable will exist and **all objects** of the class will **share** that variable.
- Static variables are **initialized to zero** before the **first object** is created.
- Static members have the **same properties** as **global variables** but they **enjoy class scope**.

Static Data Member



Static Member Functions

- Member functions that are declared with **static** specifier.

Syntax:

```
class class_name
{
public:
static ret_dt fun_name(formal parameters);
};
```

Static Member Functions

Special features:

- They can directly refer to **static members** of the class.
- They does not have **this pointer**.
- They cannot be a static and a non-static version of the **same** function.
- The may not be **virtual**.
- Finally, they cannot be declared as **const** or **volatile**.

Local Classes in C++

A class declared inside a function becomes local to that function and is called Local Class in C++.

For example, in the following program, Test is a local class in fun().

```
void fun()
{
    class Test // local to fun
    {
        /* members of Test class */
    };
}
```

Local Classes

“A class defined **within a function** is called Local Class.”

Syntax:

```
void function()
{
    class class_name
    {
        // class definition
    } obj;
    //function body
}
```

```
void fun()
{
    class myclass {
        int i;
        public:
        void put_i(int n) { i=n; }
        int get_i() { return i; }
    } ob;
    ob.put_i(10);
    cout << ob.get_i();
}
```

Characteristic of local classes

- When a class is declared within a function, it is known only to the function and therefore becomes a local class
- Global variables must be accessed using the scope resolution operator (::)
- Static variable must not be declared within a local class

Multiple Classes

Syntax:

```
class class_name1
{
//class definition
};

class class_name2
{
//class definition
};
```

Example:

```
class test
{
public:
int t[3];
};
```

Example:

```
class student
{
int st_id;
test m;
public:
void init_test()
{
m.t[0]=25;
m.t[1]=22;
m.t[2]=24;
}
};
```

Nested Class

- One class can be defined another class. This created a nested class.
- It is valid only within the scope of the enclosing class

Nested Classes

Syntax:

```
class outer_class
{
    //class definition
    class inner_class
    {
        //class definition
    };
};
```

Example:

```
class student
{
    int st_id;
    public:
    class dob
        { public:
            int dd,mm,yy;
        }dt;
    void read()
    {
        dt.dd=25;
        dt.mm=2;
        dt.yy=1988;}
};
```

Global Classes in C++

A class declared outside the body of all the function in a program is called Global Class in C++.

For example

```
class sample
```

```
{ int x; public: void readx() { cin>>x;} };
```

```
void main()
```

```
{
```

```
sample s1;
```

```
s1.readx();
```

```
}
```

Local and Global Objects in C++

A class declared outside the body of all the function in a program is called Global Class in C++.

For example

```
class sample
```

```
{ int x; public: void readx() { cin>>x;} };
```

```
sample s1; // Global Object
```

```
void main()
```

```
{
```

```
sample s2; // Local Object
```

```
s1.readx();
```

```
}
```


Chapter 5

Constructors & Destructors

Class

- A class consists of:
 - member data (**usually private**)
 - member functions (**usually public**)
 - the simplest class shall have at least two member functions:
 - set data
 - show data
 - member data can be public, and member functions can be private

Member data initialization

- After an object is created:
 - Standard ways for data initialization:
 - Setting default constants
 - Reading input from user (using `cin >>`)
 - ...

Member data initialization

- example

```
class circle
{
private:
int xC, yC, radius ;
public:
void set (int x, int y, int r)
    {
        xC = x;  yC = y;  radius = r;
    }
    void get ()
    { cout << "\nEnter x coordinate: " ;
      cin >> xC;
      cout << "\nEnter y coordinate: " ; cin
      >> yC;
      cout << "\nEnter radius : " ; cin >>
      radius;
    }
    void draw ()
    {
      draw_circle(xC, yC, radius);
    }
}
```

```
int main()  
{  
  ...  
  circle c1, c2; c1.set (15, 17, 8); c2.get (); c1.draw();  
  c2.draw()  
  ...  
}
```

Constructors

- “A **constructor** function is a special function that is a **member of a class** and has the **same name** as that **class**, used to **create**, and **initialize** objects of the **class**.”
- Constructor function do **not** have **return type**.
- Should be declared in **public** section.

Constructors

Syntax:

```
class class_name
{
public:
class_name();
};
```

Example:

```
class student
{ int st_id;
public:
    student()
    {
        st_id=0;
    }
};
```

Constructors

- How to call this special function...?

```
int main()
{
    student st;
    .....
    .....
};
```



```
class student
{
    int st_id;
    public:
    student()
    {
        st_id=0;
    }
};
```


Constructors

- Pgm to create a class **Addition** to add two integer values. Use constructor to initialize values.
- Pgm to create a class **Circle** to compute its area. Use constructor to **initialize** the data members.

Constructor - example

```
class circle
{
    private:
    int xC, yC, radius ;
    public:
    circle()
    {
        xC = 25 ; yC = 25 ; radius = 5;
    }
    void set (int x, int y, int r)
    {
        xC = x; yC = y; radius = r;
    }
    void get ()
    { cout << "\nEnter x coordinate: " ; cin
      >> xC;
      cout << "\nEnter y coordinate: " ; cin >>
      yC;
      cout << "\nEnter radius : " ; cin >>
      radius;
    }
    void draw ()
    {
        draw_circle(xC, yC, radius);
    }
}
```

```
int main() {
    ...
    circle c1, c2; c1.set (15, 17, 8);
    c1.draw(); c2.draw()
    ...
}
```

Types of Constructors

- Parameterized constructors
- Constructor without default argument
- Overloaded constructors
- Constructors with default argument
- Copy constructors
- Dynamic constructors

- C++ allows, passing of argument/ parameter to the constructor at the time of creating the object.
- A constructor with one or more arguments is known as parameterized constructor

Parameterized Constructors

```
class Addition
```

```
{
```

```
    int num1;
```

```
    int num2;
```

```
    int res;
```

```
    public:
```

```
    Addition(int a, int b); // constructor
```

```
    void add( );
```

```
    void print();
```

```
};
```

*Constructor with parameters
B'Coz it's also a function!*



Default Constructor

- A constructor which does not accept parameter is called default constructor

```
Class sample
{
    ....
    public:
    sample()
    {} // Constructor with no argument
    .....
};
```

Calling constructor implicitly or explicitly

- In a class where a constructor has been parameterized, the object declaration must pass initial values as arguments to the constructor function. This can be done by
 - Calling the constructor implicitly
 - Calling the function explicitly

Class point

{

Int x,y;

Public:

point(int a, int b);

....

.....

};

Point ::point(int a, int b)

{

x=a; y=b;

}

In the main program object can be created by calling the constructor implicitly as

```
point p1 (10,20); // implicit call
```

Where x is initialized to 10 and y =20

The second method of creating object by calling the constructor explicitly as

```
point p1=p1(10,20);
```

Constructor overloading

When more than one constructor function is defined in a Class is known as Constructor Overloading

Overloaded constructors shall have *different* parameter Lists

- **circle();**
- **circle (int x, int y, int r);**

The reason for using overloading constructor

To get flexibility

to define copy constructor

To support array

Overloaded Constructors

```
class Addition
```

```
{
```

```
    int num1,num2,res;
```

```
    float num3, num4, f_res;
```

```
    public:
```

```
    Addition(int a, int b); // int constructor
```

```
    Addition(float m, float n); //float constructor
```

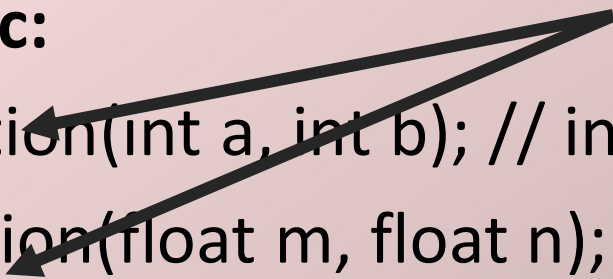
```
    void add_int( );
```

```
    void add_float();
```

```
    void print();
```

```
};
```

Overloaded Constructor with parameters B'Coz they are also functions!




Constructor with default arguments

- Similar to function with default arguments constructor too can be defined with default arguments.
- In fact overloading of a constructor can be avoided in many cases, by using default arguments
- The same rules for function with default arguments apply to constructor i.e default values must be added from right to left
- Default values cannot be provided to an argument in the middle of an arguments

Constructors with Default Argument

```
class Addition
{
    int num1;
    int num2;
    int res;
    public:
    Addition(int a, int b=0); // constructor
    void add( );
    void print();
};
```



Constructor with default parameter.

Copy Constructor

- It is a special case of constructor, used to make a copy of one class object and initialize it by using another class object of the same class type

- Format

```
Classname ( classname & obj)
```

```
{
```

```
//body of constructor
```

```
}
```

Obj is a reference to an existing object. The copy constructor normally takes one argument which is an object type class name

Copy Constructor

```
class code
{
    int id;
    public:
    code() //constructor
    { id=100;}
    code(code &obj) // constructor
    {
    id=obj.id;
    }
};
```


Dynamic constructor

- When memory is allocated to object at the time of their creation, it is known as dynamic construction of the object
- Such object can be created with the help of the new operator
- Dynamic constructor are used to allocate memory for creating such object
- This result in allocation of right amount of memory and thus result in saving of memory

Dynamic Constructors

```
class Sum_Array
{
    int *p;
    public:
    Sum_Array(int sz) // constructor
    {
        p=new int[sz];
    }
};
```

String Manipulation using Constructor

- Operation on strings can be manipulated in C++ by defining a string class and creating string object
- The operation such as concatenation, comparison, length of a string etc can be performed on the object

Destructors

- A destructor is a special member function of a class which is called automatically by the compiler to destroy the object created by the constructor

Destructors characteristic

- “A **destructor** function is a special function that is a **member of a class** and has the **same name** as that **class** used to **destroy** the **objects.**”
- Must be declared in **public** section.
- Destructor do **not** have **arguments & return type.**

NOTE:

A class can have **ONLY ONE** destructor

Syntax

- The general format is

```
~sample() { } // destructor for the  
                constructor sample()
```

Destructors

Syntax:

```
class class_name
{
public:
~class_name();
};
```

Example:

```
class student
{
    public:
    ~student()
    {
        cout<<"Destructor";
    }
};
```

Template

Dr.T.Logeswari

Templates ?



- **Templates** are a feature of the C++ programming language that allow functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

Types of templates ?



- C++ provides two kinds of templates:
 - Class templates and
 - Function templates.

Function Template?



- Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type. In C++ this can be achieved using template parameters.

What is template parameter ?



- A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass values and also types to a function.

Template Instantiation



- When the compiler generates a class, function or static data members from a template, it is referred to as template instantiation.
 - A function generated from a function template is called a generated function.

From Compiler's point of view...



- Templates are not normal functions or classes. At that moment, when an instantiation is required, the compiler generates a function specifically for those arguments from the template.



```
template <class myType>
myType GetMax (myType a, myType b)
{
    return (a>b?a:b);
}
```

Template function with two arguments of same type.



```
template <class T, class U>  
T GetMin (T a, U b)  
{  
    return (a<b?a:b);  
}
```

Template function with two arguments of different type or same type. It depends on the argument passed.

More...



We can also overload a Function Template as well as Override a Function Template.

Overloading and Overriding can be achieved through Functions as well as Template Functions.

Polymorphism in C++

Dr.T.Logeswari

Polymorphism in C++

- The process of representing one Form in multiple forms is known as **Polymorphism**. Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.
- Polymorphism is derived from 2 greek words: **poly** and **morphs**. The word "poly" means many and **morphs** means forms. So polymorphism means many forms.

Real life example of Polymorphism in C++

- Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person have different-different behaviors.

Type of polymorphism

- Compile time polymorphism
- Run time polymorphism

Compile time polymorphism

- In C++ programming you can achieve compile time polymorphism in two way, which is given below;
 - Method overloading
 - Method overriding
- **Method Overloading in C++**
- Whenever same method name is existing multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**

- In below example method "sum()" is present in Addition class with same name but with different signature or arguments.

Example of Method Overloading in C++

```
#include<iostream.h>
#include<conio.h>
class Addition
{
public:
void sum(int a, int b)
{
cout<<a+b;
}
void sum(int a, int b, int c)
{ cout<<a+b+c;
}
};
```



```
void main()
{
  clrscr();
  Addition obj;
  obj.sum(10, 20);
  cout<<endl;
  obj.sum(10, 20, 30);
}
```

Output

30

60

Method Overriding in C++

- Define any method in both base class and derived class with same name, same parameters or signature, this concept is known as **method overriding**.
- In below example same method "show()" is present in both base and derived class with same name and signature.

Example of Method Overriding in C++

```
#include<iostream.h>
#include<conio.h>
class Base
{
public: void show()
{
cout<<"Base class";
}
};
class Derived:public Base
{
public: void show()
{
cout<<"Derived Class";
}}}
```

```
int main()
{
Base b; //Base class object
Derived d; //Derived class object
b.show(); //Early Binding Occurs
d.show();
getch();
}
```

Output

Base class

Derived Class

Run time polymorphism

- In C++ Run time polymorphism can be achieved by using [virtual function](#).
- A **virtual function** is a member function of a class that is declared within a base class and re-defined in a derived class.
- When you want to use the same function name in both the base and derived class, then the function in the base class is declared as virtual by using the **virtual** keyword and again re-defined this function in the derived class without using the virtual keyword.

Syntax

```
Virtual return_type function_name()  
{  
.....  
.....  
}
```

Virtual Function Example

```
#include<iostream.h>
#include<conio.h>

class A
{
public:
virtual void show()
{
cout<<"Hello base class";
}
};
```

```
class B : public A
{
public: void show()
{
cout<<"Hello derive class";
}
};

void main()
{
clrscr();
A aobj;
B bobj;
```



```
A *bptr;  
bptr=&aobj;  
bptr->show(); // call base class function  
    bptr=&bobj;  
bptr->show(); // call derive class function  
    getch();  
}
```

Output

Hello base class

Hello derive class

Pure Virtual Functions

- Pure virtual Functions are virtual functions with no definition.
- They start with **virtual** keyword and ends with = 0.
- Here is the syntax for a pure virtual function
Virtual return_type function_name(arg_list)=0

Eg:

```
virtual void f() = 0;
```

Abstract Class

Abstract Class is a class which contains at least one Pure Virtual function in it.

Abstract classes are used to provide an Interface for its sub classes.

Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Example of Abstract Class

```
class Base //Abstract base class
{
public:
virtual void show() = 0; //Pure Virtual Function
};
class Derived:public Base
{
public: void show()
{
cout << "Implementation of Virtual Function in
Derived class"; } };
```

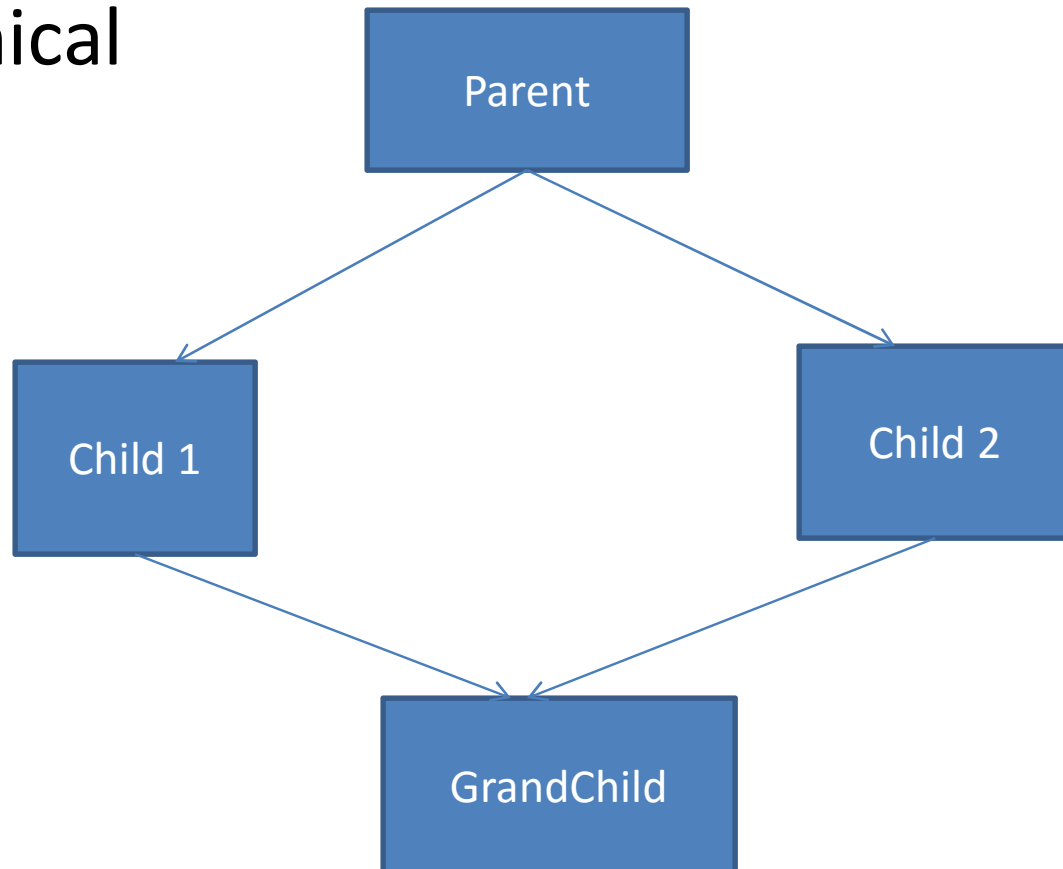
```
int main()
{
Base obj; //Compile Time Error
Base *b;
Derived d;
  b = &d;
  b->show();
}
```

Output :

Implementation of Virtual Function in Derived
class

Virtual Base Class

- Some situation we need to use all three kinds of inheritance: multilevel, multiple and hierarchical



- The grand child inherit the qualities of parent through two separate paths
 - Parent, child1, grandchild
 - Parent, child2, grandchild

Streams

Dr.T.Logeswari

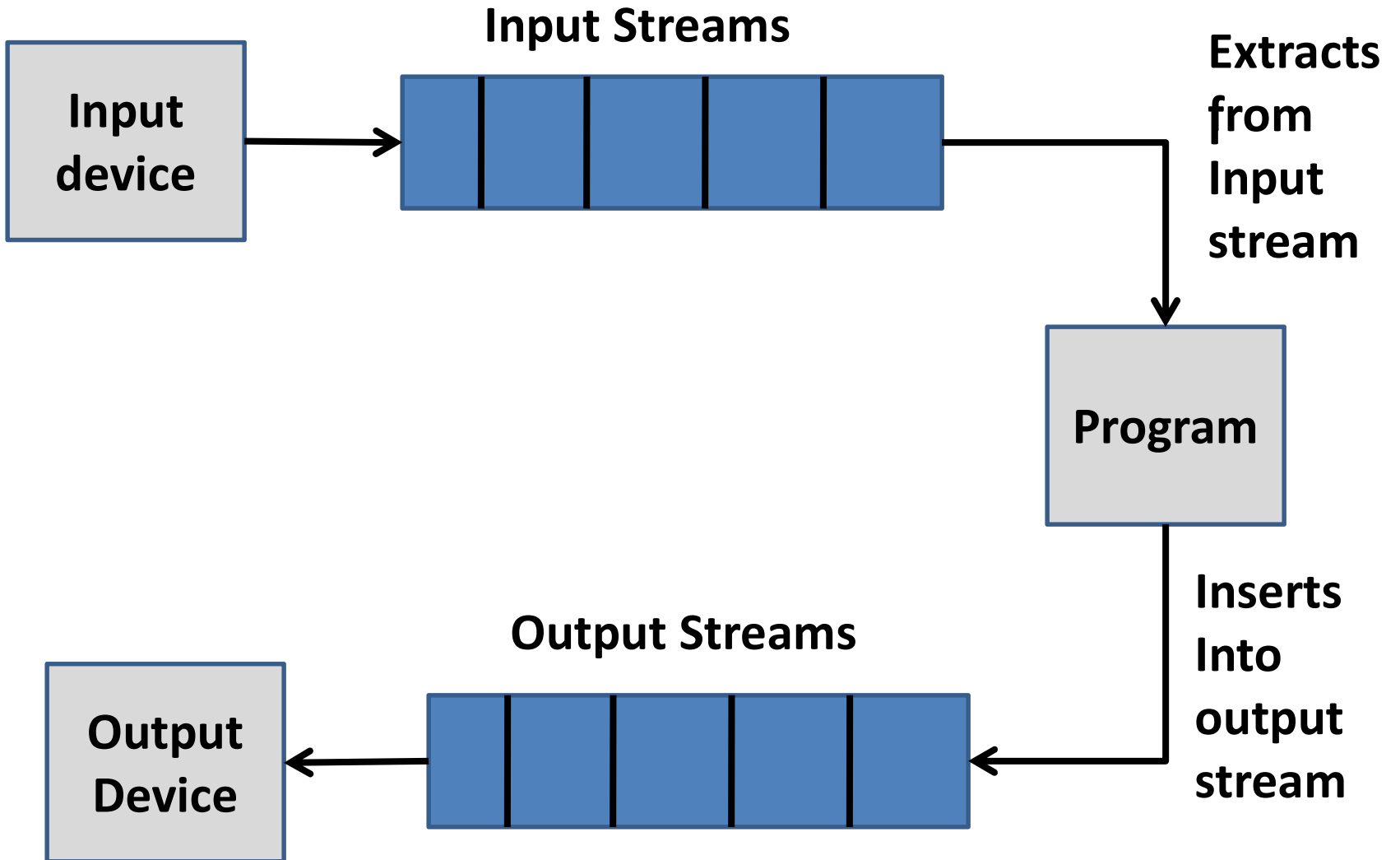
Introduction

- In C++ I/O system operates through **streams**.
- I/O system provides a level of **abstraction** between the **programmer** and the **device**.
- This **abstraction** is called a *stream* and the *actual device* is called a *file*.

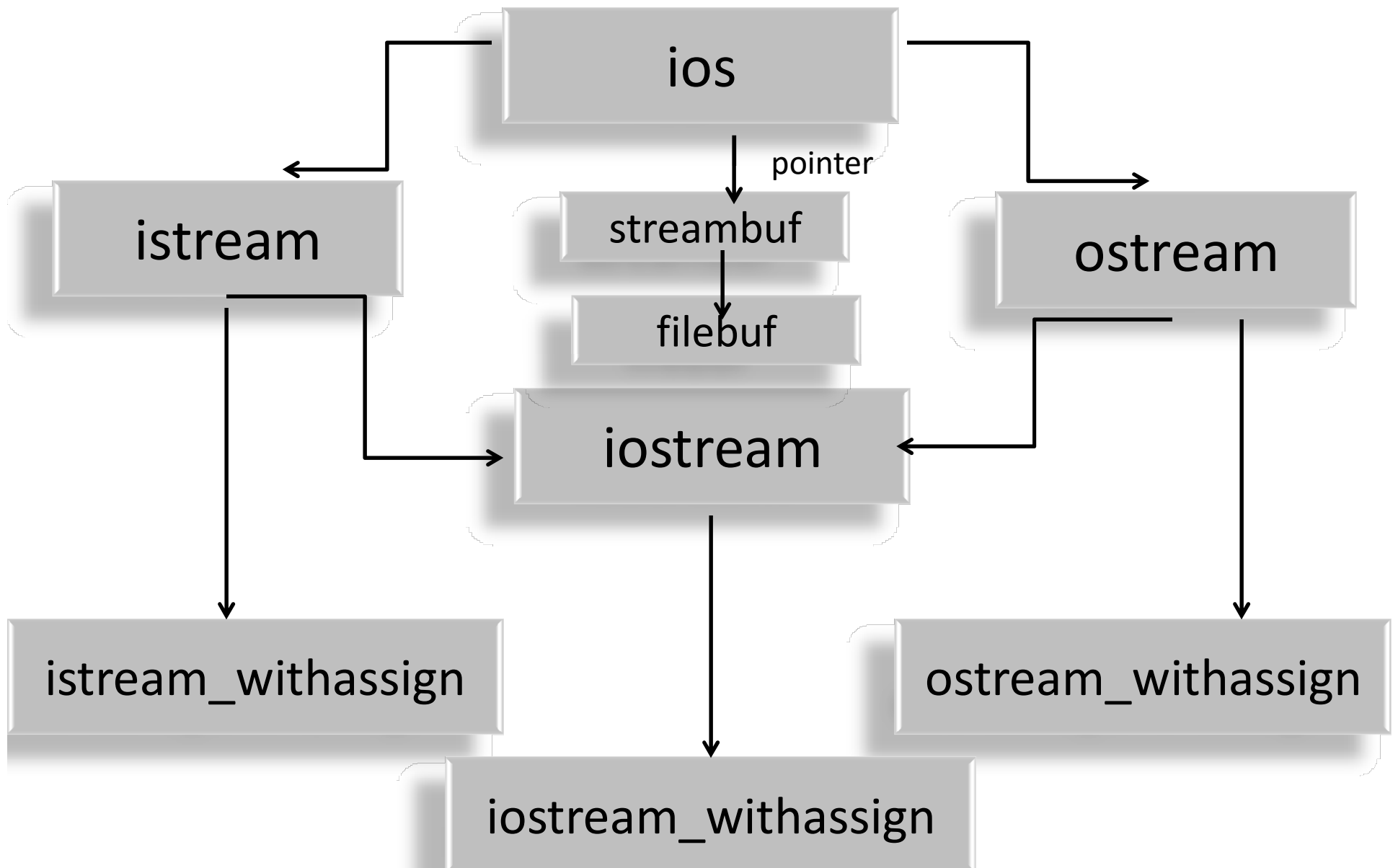
Introduction

- A stream is a **logical device** that either produces or consumes information.
- A stream is **linked** to a **physical device** by the I/O system.
- Standard C++ provides support for its I/O system in **<iostream.h>**

C++ Streams



I/O Stream Classes for console Operations



C++'s Predefined Streams

When a C++ program begins execution, Two built-in streams are automatically opened.

Stream	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output	Screen

Unformatted I/O

- Input operator
- Output operator
- Overloading I/O Operator

Input Operator

- Extraction operator:(>>)
- float var;
cin >>var;
char line[20];
cin>>line;
- get(), getline(),read()

Output Operator

- Insertion Operator:(<<)
- float var;
char line[20];
cout<< var<<line;
- put(),putline(),write()

Formatted I/O

- There are three related but conceptually different ways that we can format data.
 - directly accessing members of the **ios** class.
 - using special functions called **manipulators**.
 - user defined output functions

Formatting Using the ios Members

- The **ios** class declares a bitmask enumeration called **fmtflags** in which the following set of format flags are defined.
- To set a flag, the `setf()` function is used. This function is a member of `ios`.
- **Syntax:** `fmtflags setf(fmtflags flags);`
example: `stream.setf(ios::showpos);`

Flag	Meaning
skipws	leading white-space characters are discarded when performing input on a stream
left	output is left justified.
right	output is right justified. Default is right justified.
internal	a numeric value is padded to fill a field by inserting spaces between any sign or base character.
oct	flag causes output to be displayed in octal.
hex	flag causes output to be displayed in hexadecimal.
dec	flag causes output to be displayed in decimal. Default is decimal output.
showbase	Shows the base of numeric values

Flag	Meaning
showpos	causes a leading plus sign to be displayed before positive values.
scientific	floating-point numeric values are displayed using scientific notation. By default, when scientific notation is displayed, the e is in lowercase.
uppercase	characters are displayed in uppercase.
showpoint	causes a decimal point and trailing zeros to be displayed for all floating-point output
fixed	floating-point values are displayed using normal notation.
unitbuf	the buffer is flushed after each insertion operation.
boolalpha	Booleans can be input or output using the keywords true and false.

Function	Meaning
width()	To specify required field size for displaying an output value.
precision()	To specify the number of digits to displayed after the decimal point of a float value value.
fill()	To specify a character to used to fill the unused portion of a field.
setf()	Sets the format flags
unsetf()	Un-Sets the format flags

Using Manipulators to Format I/O

Manipulators	Meaning
boolalpha	Turns on boolalpha flag.
dec	Turns on dec flag.
endl	Output a newline character and flush the stream.
ends	Output a null.
fixed	Turns on fixed flag.
flush	Flush a stream.
hex	Turns on hex flag.
internal	Turns on internal flag.
left	Turns on left flag.
noboolalpha	Turns off boolalpha flag.

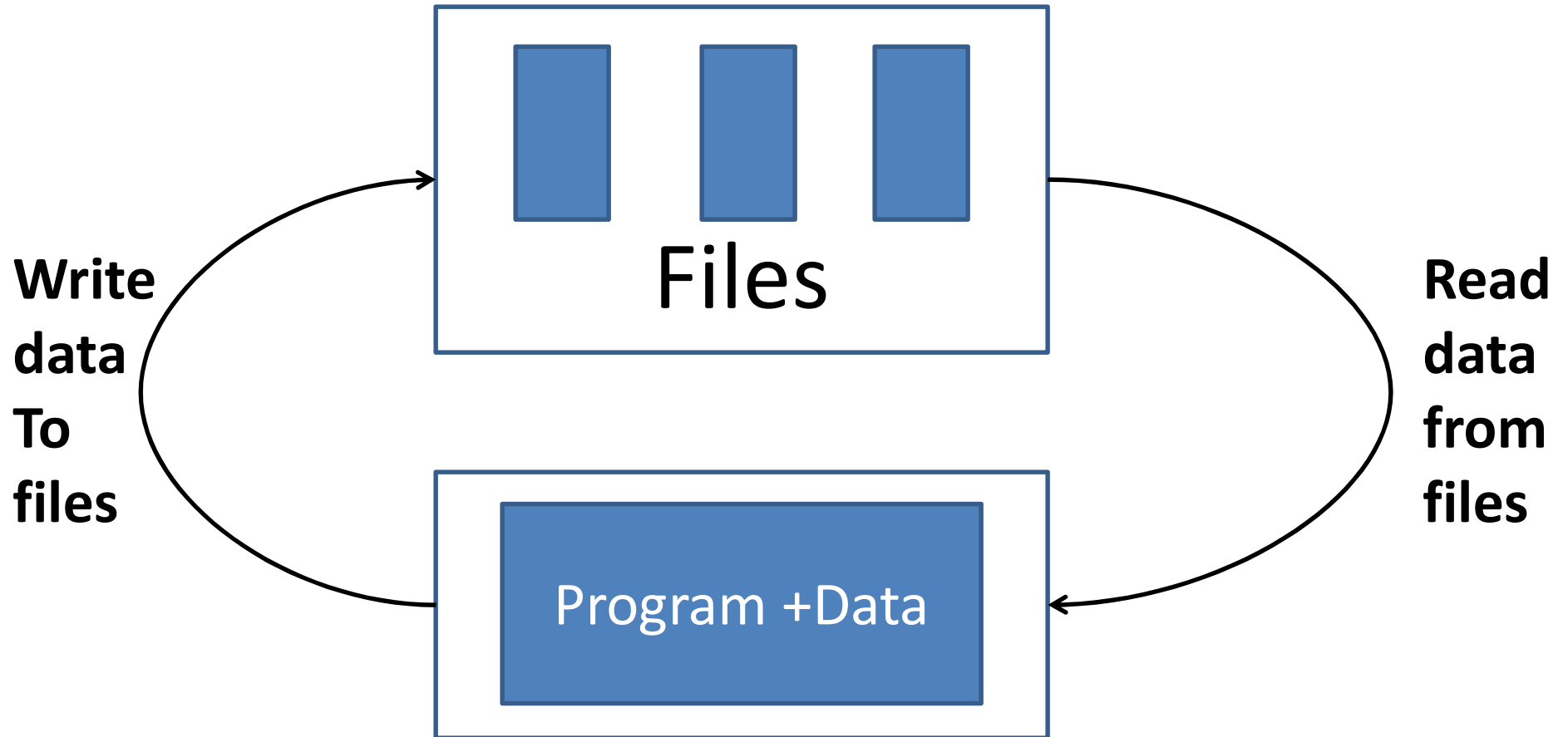
Manipulators	Meaning
noshowbase	Turns off showbase flag.
noshowpoint	Turns off showpoint flag.
no showpos	Turns off showpos flag.
noskipws	Turns off skipws flag.
nounitbuf	Turns off unitbuf flag.
noupper	Turns off uppercase flag.
oct	Turns on oct flag.
right	Turns on right flag.
scientific	Turns on scientific flag.
setbase(int base)	Set the number base to <i>base</i>.

Manipulators	Meaning
setfill(int ch)	Set the fill character to <i>ch</i> .
setiosflags(fmtflags f)	Turn on the flags specified in <i>f</i> .
setprecision(int p)	Set the number of digits of precision.
setw(int w)	Set the field width to <i>w</i> .
showbase	Turns on showbase flag .
showpoint	Turns on showpoint flag .
showpos	Turns on showpos flag .
skipws	Turns on skipws flag .
unitbuf	Turns on unitbuf flag .
uppercase	Turns on uppercase flag .
ws	Skip leading white space.

INTRODUCTION

- ✓ Computer programs are associated to work with files as it helps in storing data & information permanently.
- ✓ File - itself a bunch of bytes stored on some storage devices.
- ✓ In C++ this is achieved through a component header file called *fstream.h*
- ✓ The I/O library manages two aspects- as interface and for transfer of data.
- ✓ The library predefine a set of operations for all file related handling through certain classes.

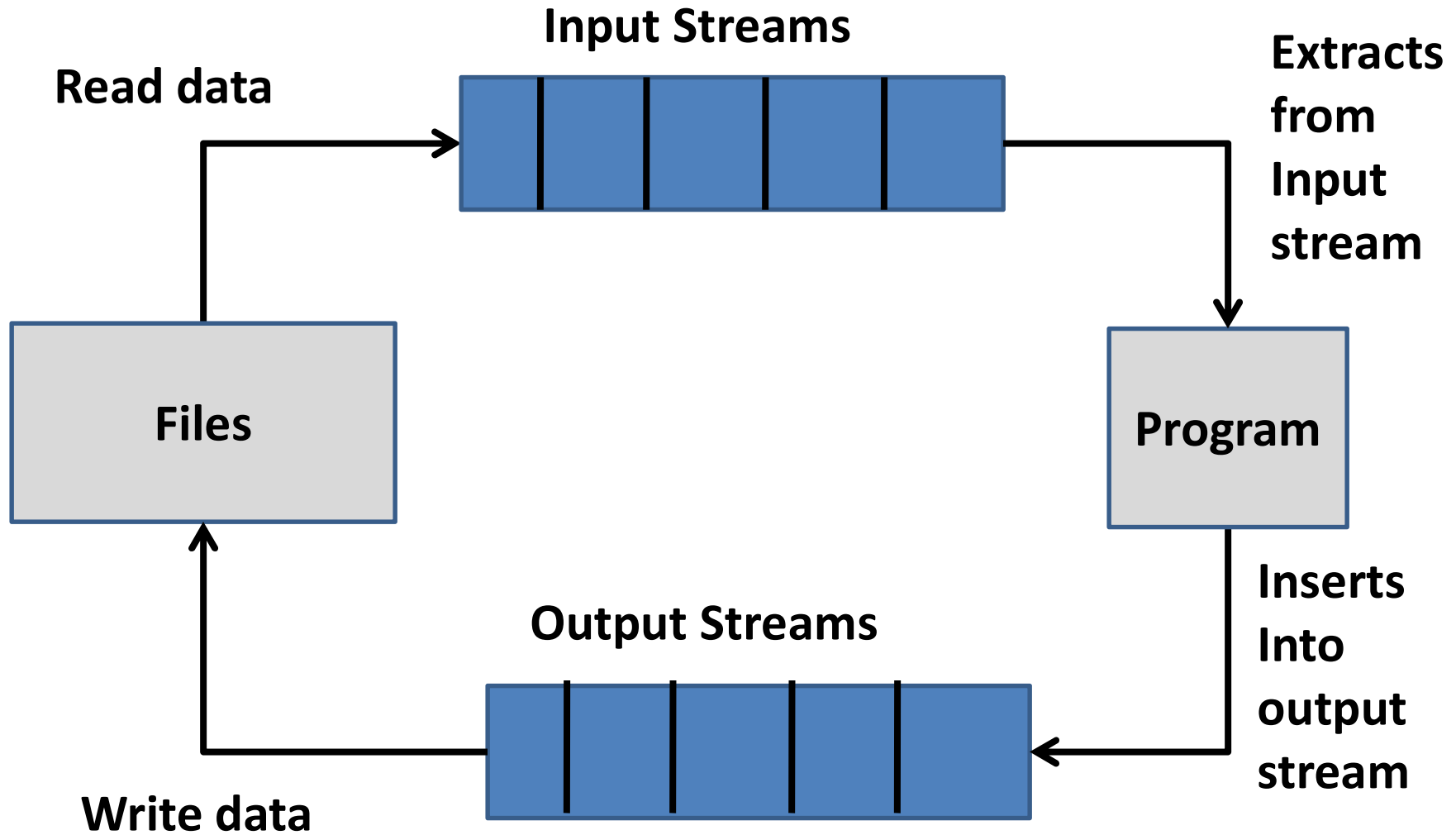
File I/O Operations



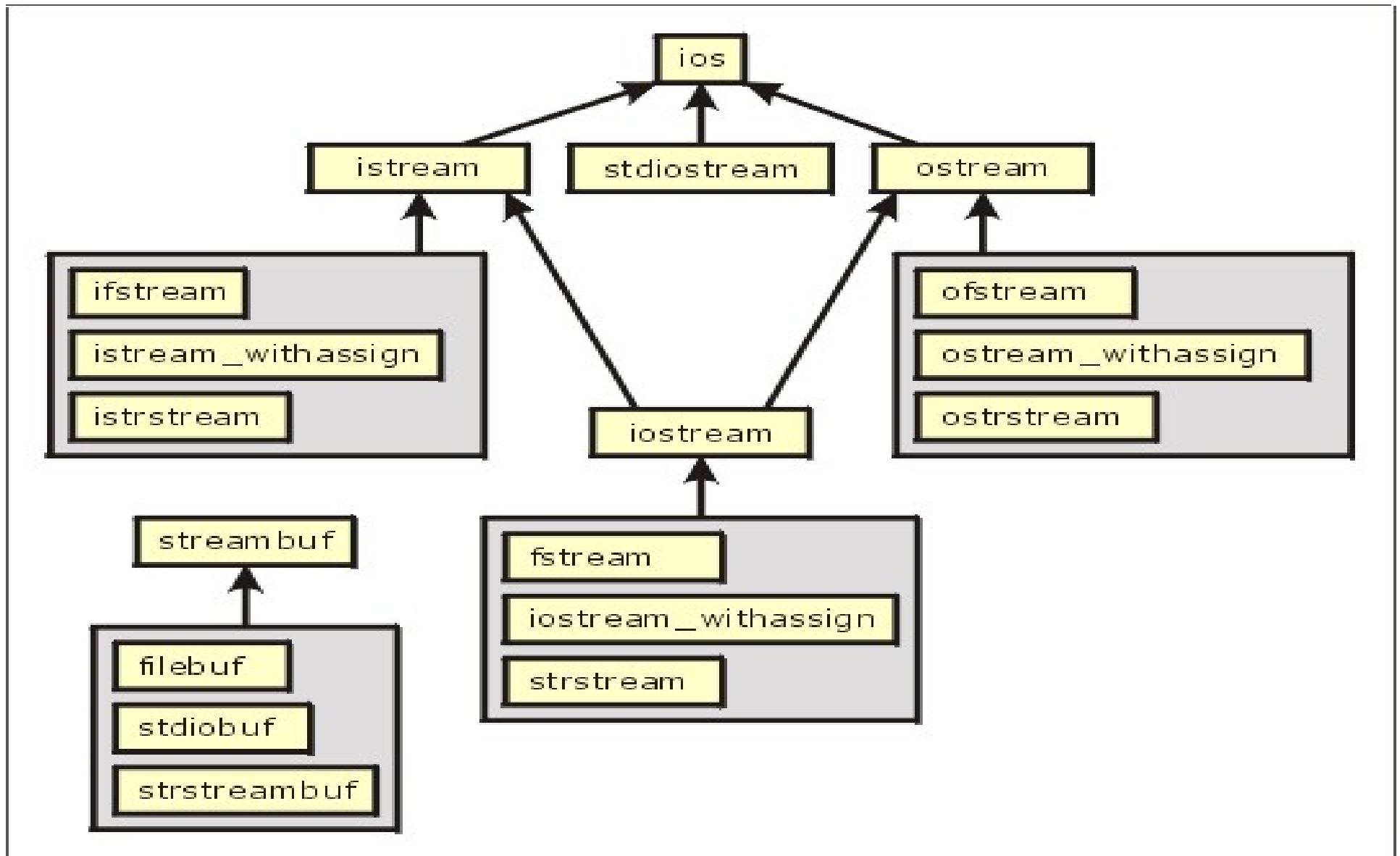
The *fstream.h* header file

- A stream is a general term used to name flow of data.
- Streams act as an interface between files and programs.
- A Stream is sequence of bytes.
- They represent as a sequence of bytes and deals with the flow of data.
- Every stream is associated with a class having member functions and operations for a particular kind of data flow.
- File → Program (Input stream) - reads
- Program → File (Output stream) – write
- All designed into `fstream.h` and hence needs to be included in all file handling programs.
- Diagrammatically as shown in next slide

File Input & Output streams



I/O Stream Class Hierarchy



FUNCTIONS OF FILE STREAM CLASSES

- ✓ **filebuf** – It sets the buffer to read and write, it contains close() and open() member functions on it.
- ✓ **fstreambase** – this is the base class for fstream and, ifstream and ofstream classes. therefore it provides the common function to these classes. It also contains open() and close() functions.
- ✓ **ifstream** – Being input class it provides input operations it inherits the functions get(), getline(), read(), and random access functions seekg() and tellg() functions.
- ✓ **ofstream** – Being output class it provides output operations it inherits put(), write() and random access functions seekp() and tellp() functions.
- ✓ **fstream** – it is an i/o class stream, it provides simultaneous input and output operations.

File TYPES

✓ A File can be stored in two ways

✓ **Text File**

✓ **Binary File**

Text Files : Stores information in ASCII characters. In text file each line of text is terminated by with special character known as EOL (End of Line) In text file some translations takes place when this EOL character is read or written.

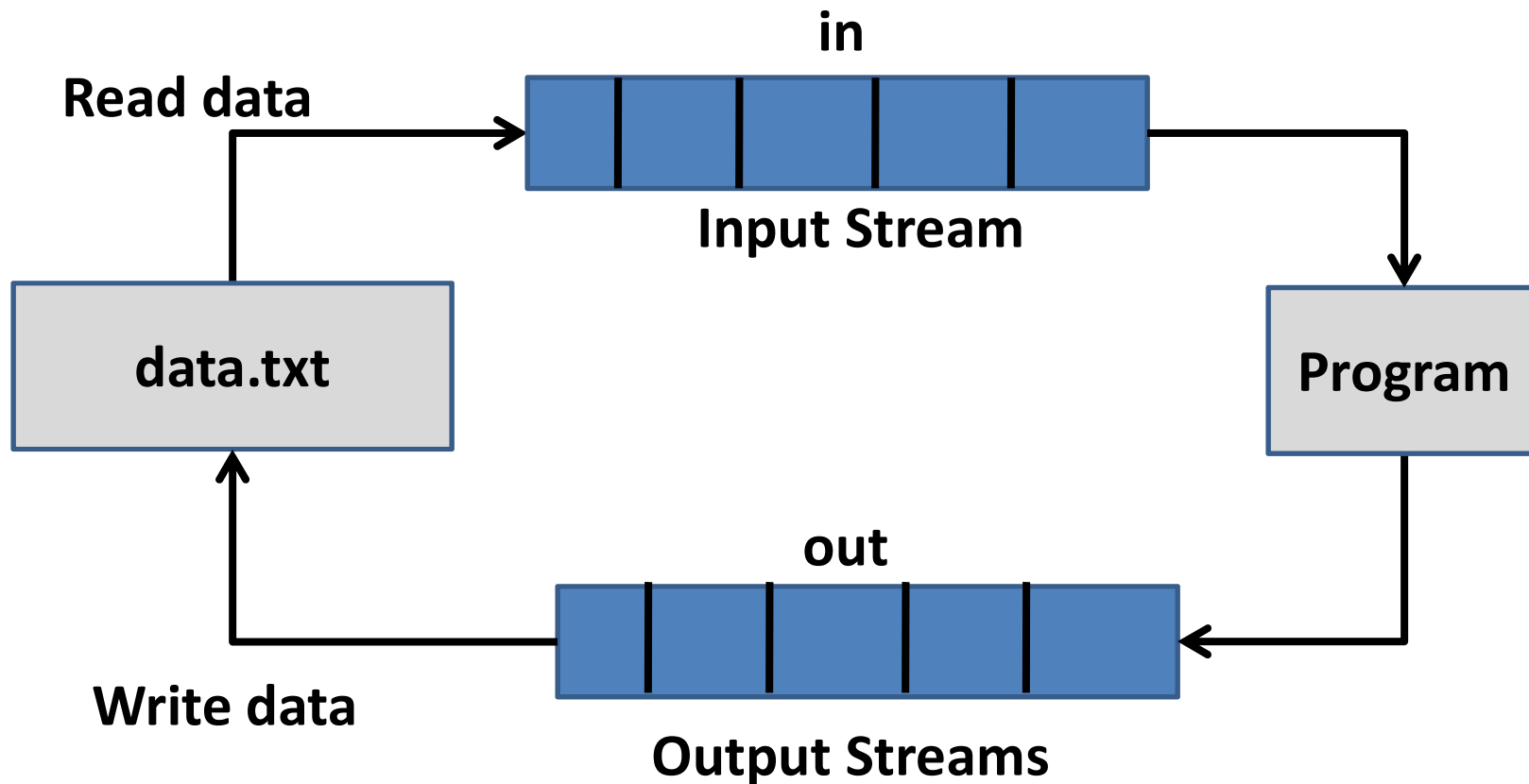
Binary File: it contains the information in the same format as it is held in the memory. In binary file there is no delimiter for a line. Also no translation occur in binary file. As a result binary files are faster and easier for program to read and write.

Opening & Closing a File

- **Opening (default mode):**
 - Create a file stream
 - Link it to the filename
 - Two methods to Open a file
 - Using constructor function of the class
 - Using member function **open()** of the class
- **Closing**
 - Delinking the file stream from filename

Using constructor of the class

- `ofstream out("data.txt");`
- `ifstream in("data.txt");`



Using member function `open()` of the class

- **creating a filestream for writing**
ofstream out;
out.open("result.txt",ios::app);
- **creating a filestream for reading**
ifstream in;
in.open("inputdata.txt",ios::app);
- **closing a file**
 - out.close();
 - in.close();

File modes

✓WHAT IS FILE MODE?

✓The File Mode describes how a file is to be used ; to read from it, write to it, to append and so on

Syntax

```
Stream_object.open("filename",mode);
```

File Modes

✓**ios::out**: It open file in output mode (i.e write mode) and place the file pointer in beginning, if file already exist it will overwrite the file.

✓**ios::in** It open file in input mode(read mode) and permit reading from the file.

File modes

- ✓ **ios::app** It open the file in write mode, and place file pointer at the end of file i.e to add new contents and retains previous contents. If file does not exist it will create a new file.
- ✓ **ios::ate** It open the file in write or read mode, and place file pointer at the end of file i.e input/ output operations can performed anywhere in the file.
- ✓ **ios::trunc** It truncates the existing file (empties the file).
- ✓ **ios::nocreate** If file does not exist this file mode ensures that no file is created and `open()` fails.
- ✓ **ios::noreplace** If file does not exist, a new file gets created but if the file already exists, the `open()` fails.
- ✓ **ios::binary** Opens a file in binary mode.

Closing a File

- ✓ A File is closed by disconnecting it with the stream it is associated with. The `close()` function is used to accomplish this task.

Syntax:

```
Stream_object.close( );
```

Example :

```
fout.close();
```

Steps To Create A File

1. Declare an object of the desired file stream class(ifstream, ofstream, or fstream)
2. Open the required file to be processed using constructor or open function.
3. Process the file.
4. Close the file stream using the object of file stream.

eof () Function

This function determines the end-of-file by returning true(non-zero) for end of file otherwise returning false(zero).

Syntax

```
Stream_object.eof( );
```

Example :

```
fout.eof( );
```

Text File Functions

get() – read a single character from text file and store in a buffer.

e.g file.get(ch);

put() - writing a single character in textfile

e.g. file.put(ch);

getline() - read a line of text from text file store in a buffer.

e.g file.getline(s,80);

We can also use **file>>ch** for reading and **file<<ch** writing in text file. But **>>** operator does not accept white spaces.

Program to create a text file using strings I/O

```
#include<fstream.h> //header file for file operations
void main()
{
char s[80], ch;
ofstream file("myfile.txt"); //open myfile.txt in default output mode
do
{ cout<<"\n enter line of text";
gets(s); //standard input
file<<s; // write in a file myfile.txt
cout<<"\n more input y/n";
cin>>ch;
}while(ch!='n' || ch!='N');
file.close();
} //end of main
```

Program to read content of 'myfile.txt' and display it on monitor.

```
#include<fstream.h> //header file for file operations
void main()
{
char ch;
ifstream file("myfile.txt"); //open myfile.txt in default input mode
while(file)
{ file.get(ch) // read a
character from text file '
myfile.txt'
cout<<ch; // write a character in text file 'myfile.txt '
}
file.close();
} //end of main
```

Binary File Functions

read()- read a block of binary data or reads a fixed number of bytes from the specified stream and store in a buffer.

Syntax : `Stream_object.read((char *)& Object, sizeof(Object));`

e.g `file.read((char *)&s, sizeof(s));`

write() – write a block of binary data or writes fixed number of bytes from a specific memory location to the specified stream.

Syntax : `Stream_object.write((char *)& Object, sizeof(Object));`

e.g `file.write((char *)&s, sizeof(s));`

Binary File Functions

Note: Both functions take two arguments.

- The first is the address of variable, and the second is the length of that variable in bytes. The address of variable must be type cast to type `char*` (pointer to character type)
- The data written to a file using `write()` can only be read accurately using `read()`.

Program to create a binary file 'student.dat' using structure.

```
#include<fstream.h>
struct student
{
char name[15];
float percent;
};
void main()
{
ofstream fout;
char ch;
fout.open("student.dat", ios::out | ios:: binary);
clrscr();
student s;
if(!fout)
{
cout<<"File can't be opened";
exit(0);
}
```

```
do
{ cout<<"\n
enter name of student";
gets(s);
cout<<"\n enter percentage";
cin>>percent;
fout.write((char *)&s,sizeof(s)); // writing a record in a student.dat file
cout<<"\n more record y/n";
cin>>ch;
}while(ch!='n' || ch!='N');
fout.close();
}
```


Program to read a binary file 'student.dat' display records on monitor.

```
#include<fstream.h>
struct student
{
char name[15];
float percent;
};
void main()
{
ifstream fin;
student s;
fin.open("student.dat",ios::in | ios:: binary);
fin.read((char *) &s, sizeof(student)); //read a record from file
'student.dat'
```

CONTD.....

```
while(file)
{
cout<<s.name;
cout<<"\n has the percent: "<<s.percent;
fin.read((char *) &s, sizeof(student));
}
fin.close();
}
```

File Pointer

The file pointer indicates the position in the file at which the next input/output is to occur.

Moving the file pointer in a file for various operations viz modification, deletion , searching etc. Following functions are used

`seekg()`: It places the file pointer to the specified position in input mode of file.

e.g `file.seekg(p,ios::beg)`; or

`file.seekg(-p,ios::end)`, or

`file.seekg(p,ios::cur)`

i.e to move to p byte position from beginning, end or current position.

File Pointer

`seekp()`: It places the file pointer to the specified position in output mode of file.

e.g `file.seekp(p,ios::beg);` or `file.seekp(-p,ios::end),` or `file.seekp(p,ios::cur)`

i.e to move to p byte position from beginning, end or current position.

`tellg()`: This function returns the current working position of the file pointer in the input mode.

e.g `int p=file.tellg();`

`tellp()`: This function returns the current working position of the file pointer in the output mode.

e.f `int p=file.tellp();`

File Pointers

- Each file has two associated pointers
 - **get pointer** : to reads from file from given location
 - **put pointer** : to writes to file from given location
- **Manipulation of get pointer**
 - **seekg**: moves get pointer to a specified location
 - **tellg**: gives the current position of the get pointer
- **Manipulation of put pointer**
 - **seekp**: moves put pointer to a specified location
 - **tellp**: gives the current position of the put pointer

Moving to a specified location in file

- **Syntax:**

- `seekg(n_bytes);` //can be + or – n bytes
- `seekg(n_bytes, reposition);`

- **reposition constants:**

- `ios::beg`
- `ios::cur`
- `ios::end`

NOTE:

+ → go forward by n bytes

- → go backwards by n bytes

Error Handling with Files

- File which we are attempting to open for reading does not exist.
- The filename used for a new file may already exist.
- attempting an invalid operation such as reading past the eof.
- attempting to perform an operation when a file is not opened for that purpose.

Function	Return value & meaning
eof()	returns true (non-zero) if end-of-file encountered while reading otherwise false(zero)
fail()	returns true when an input or output operation has failed
bad()	returns true if an invalid operation is attempted or any unrecoverable error has occurred. if it is false it may be possible to recover from any other error reported and continue operation
good()	returns true if no error has occurred, if it is false, no further operations can be carried out.