Object Oriented Programming with C++

Prepared by

ALOK KUMAR

MCA-2012

Part – A
1. Overview of OOP

Object Oriented Paradigm

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

Structured Vs Object Oriented Programming

Function Oriented	Object Oriented
Procedure Abstraction	Procedure & Data abstraction
Does not support External Interface	Supports External Interface
Free flow of Data	Secured Data & not freely flows
Also called FOP	Also called OOP

Elements of OOP

Well suited for:

- Modeling the real world problem as **close** as possible to the users perspective.
- **Interacting** easily with computational environment.
- Constructing **reusable** software components and easily extendable libraries.
- Easily **modifying** and **extending** implementations of components without having to recode every thing from scratch.

Elements of OOP

Definition of OOP:

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

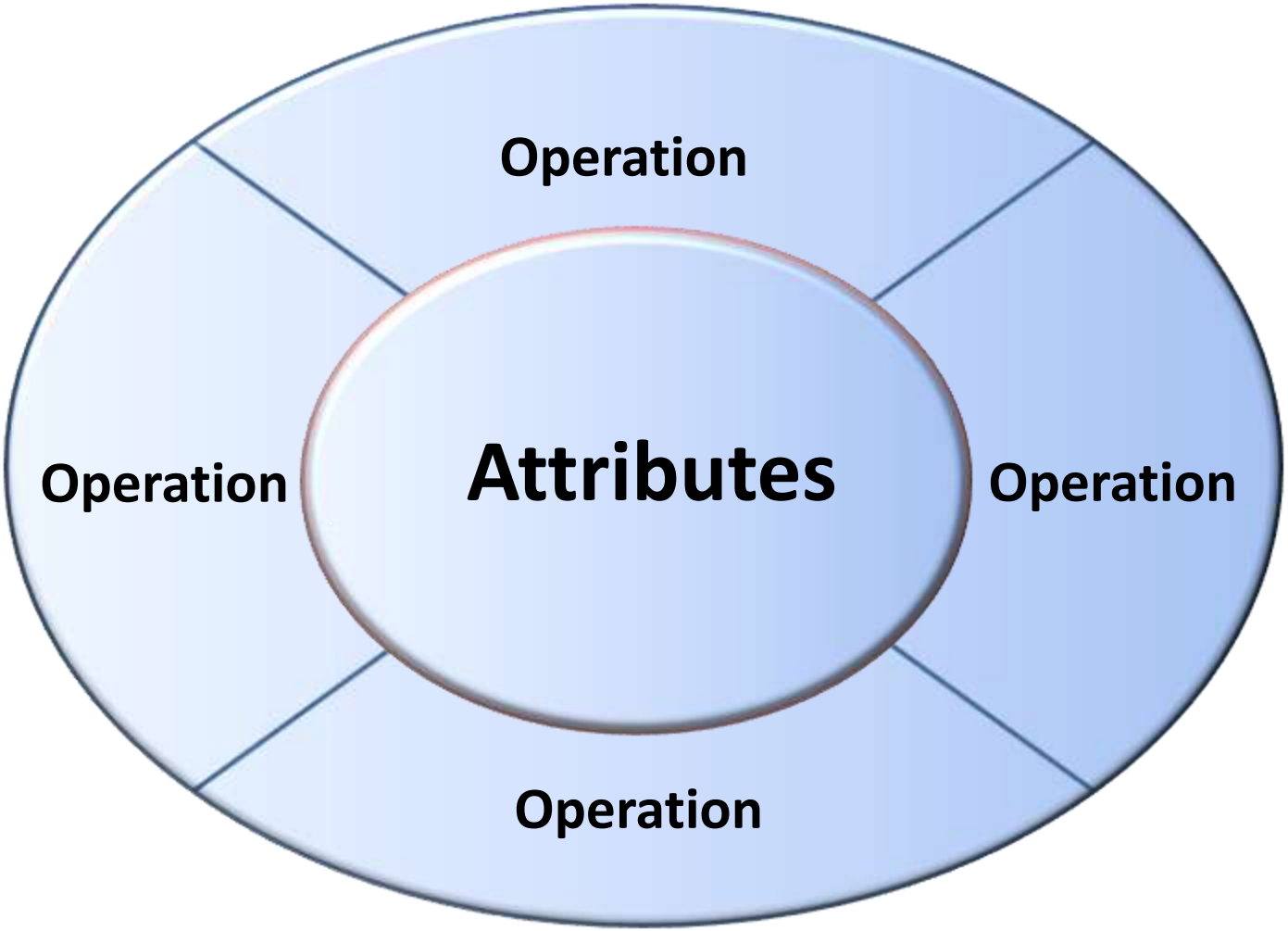
Elements of OOP

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

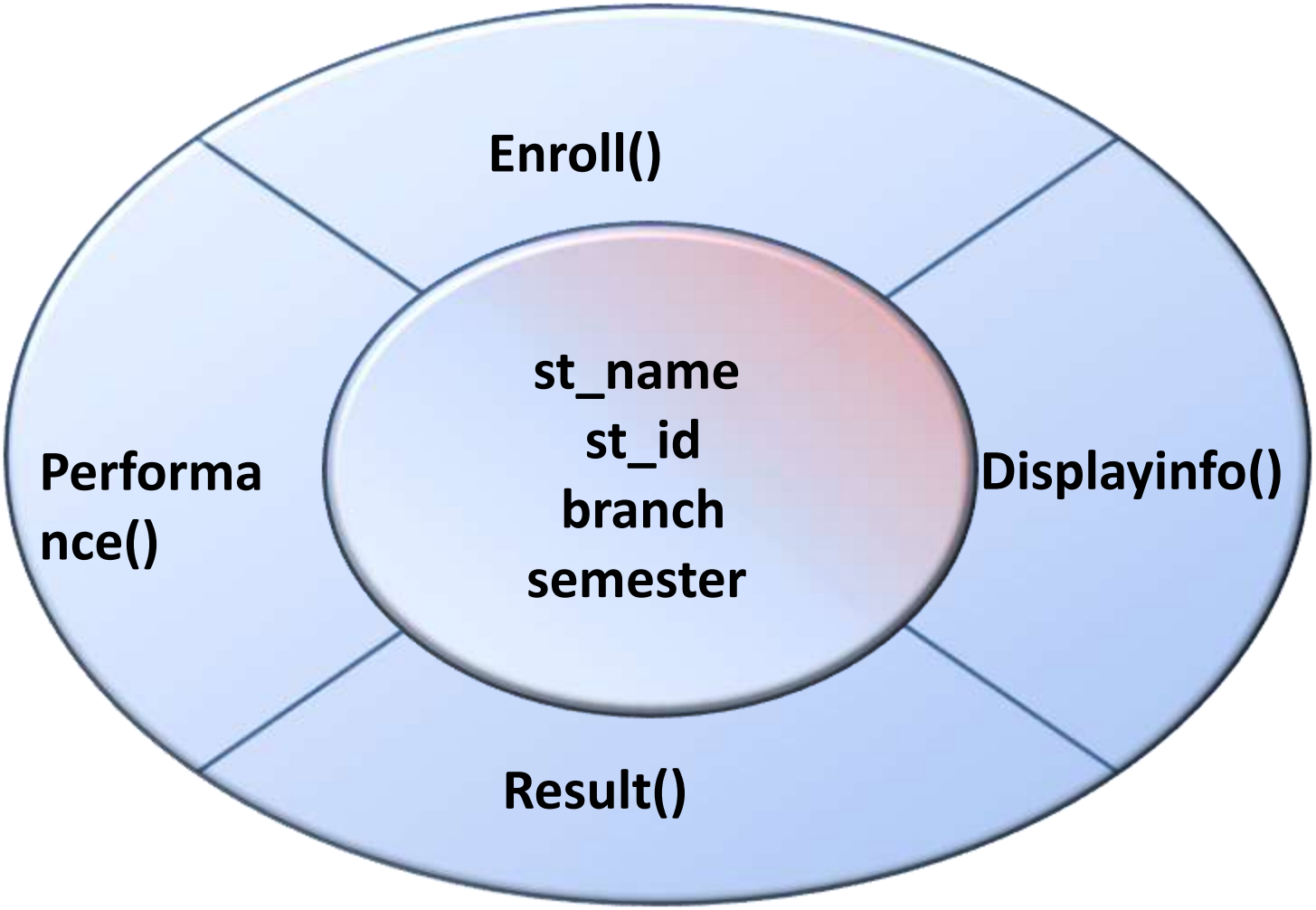
Objects

- 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 is an**Instance** of a particular class.

Object

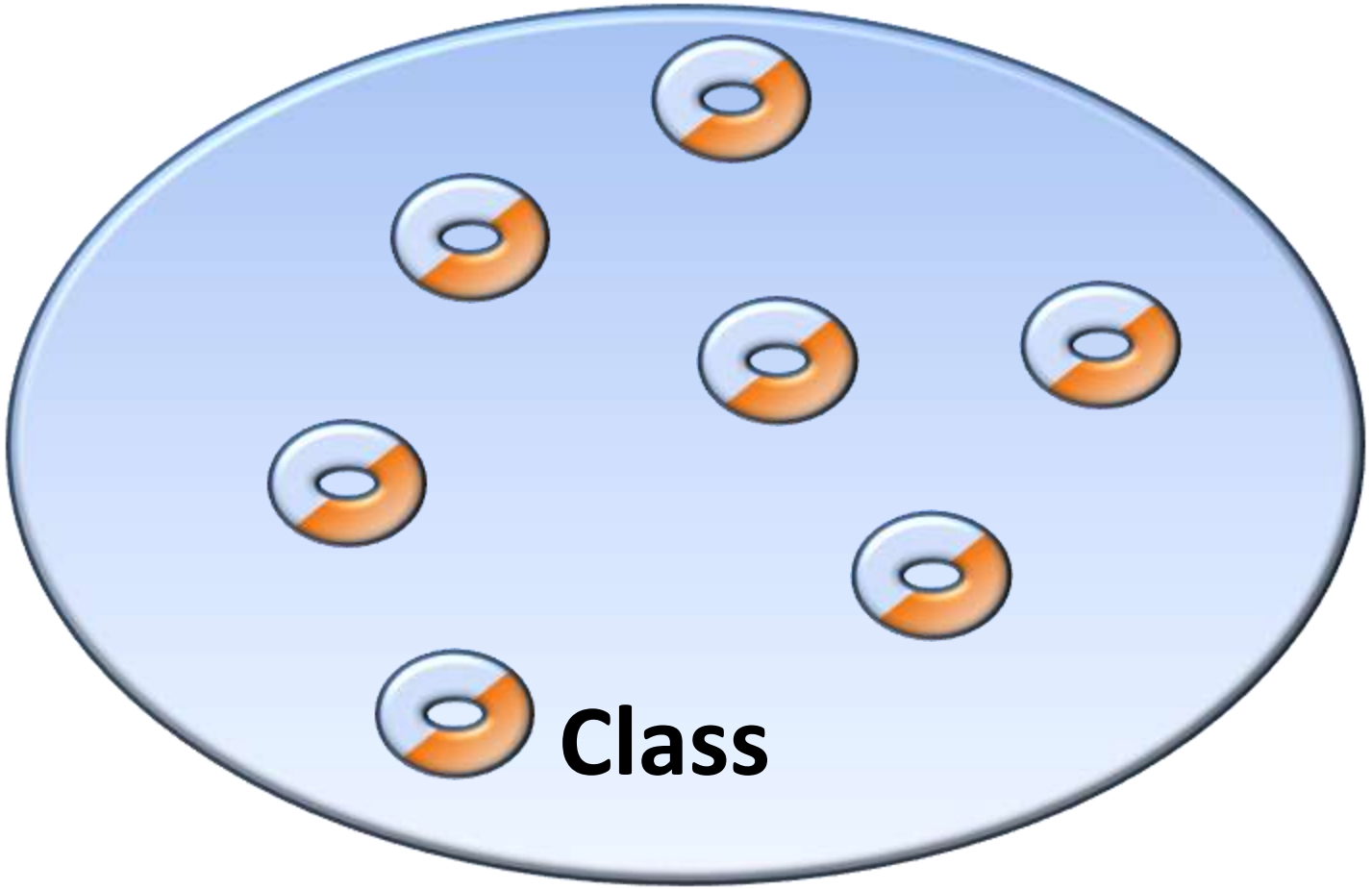


Example: StudentObject



Class

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



Encapsulation

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

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."

C++ Implementation

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

C++ Implementation

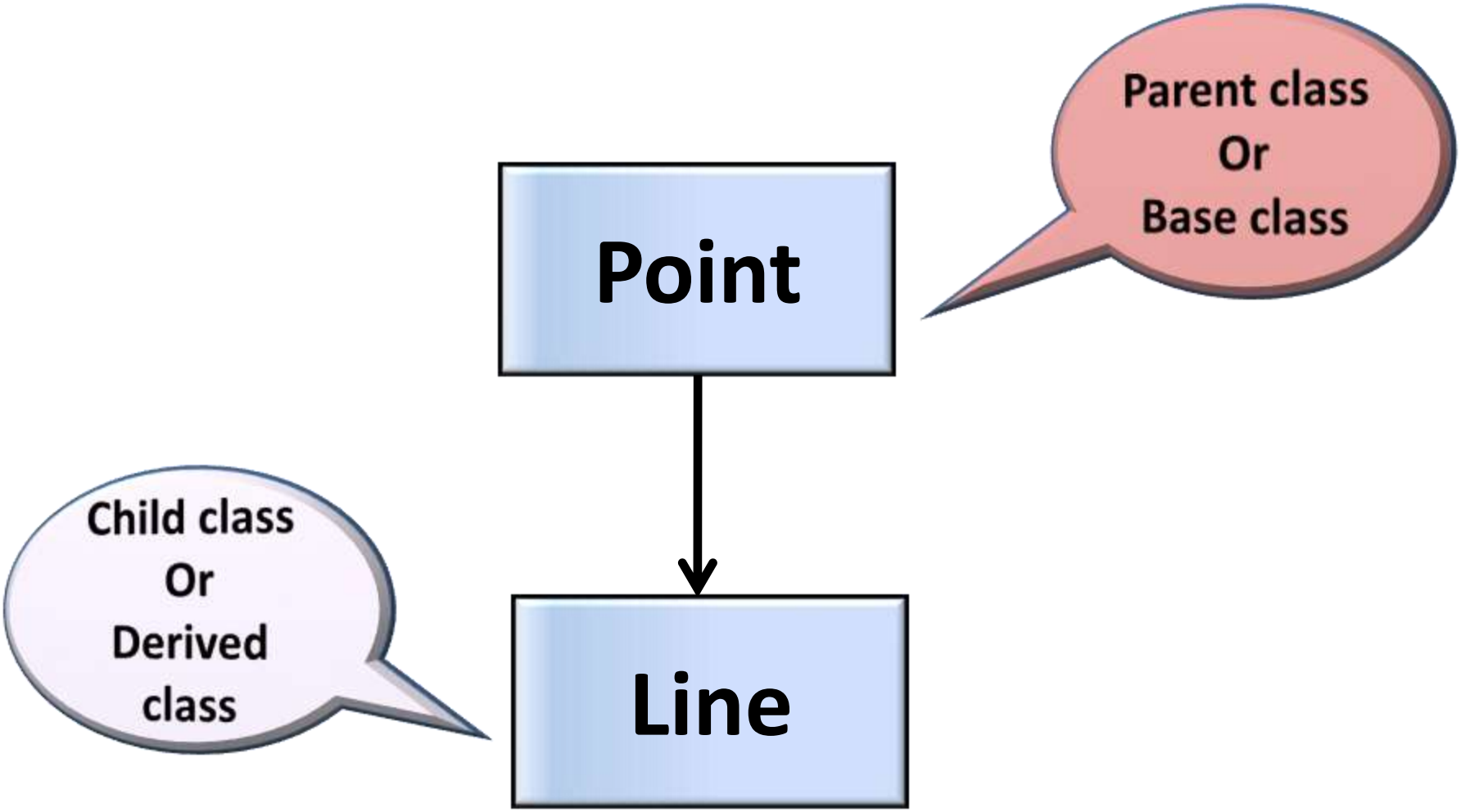
```
class student
{
char st_name[30];
char st_id[10];
char branch[10];
char semester[10];
Void Enroll( );
Void Displayinfo( );
Voide Result( );
Void Performance( );
};
```

```
class stack
 {
int stck[SIZE];
int tos;
void init();
void push(int i);
int pop();
};
```

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

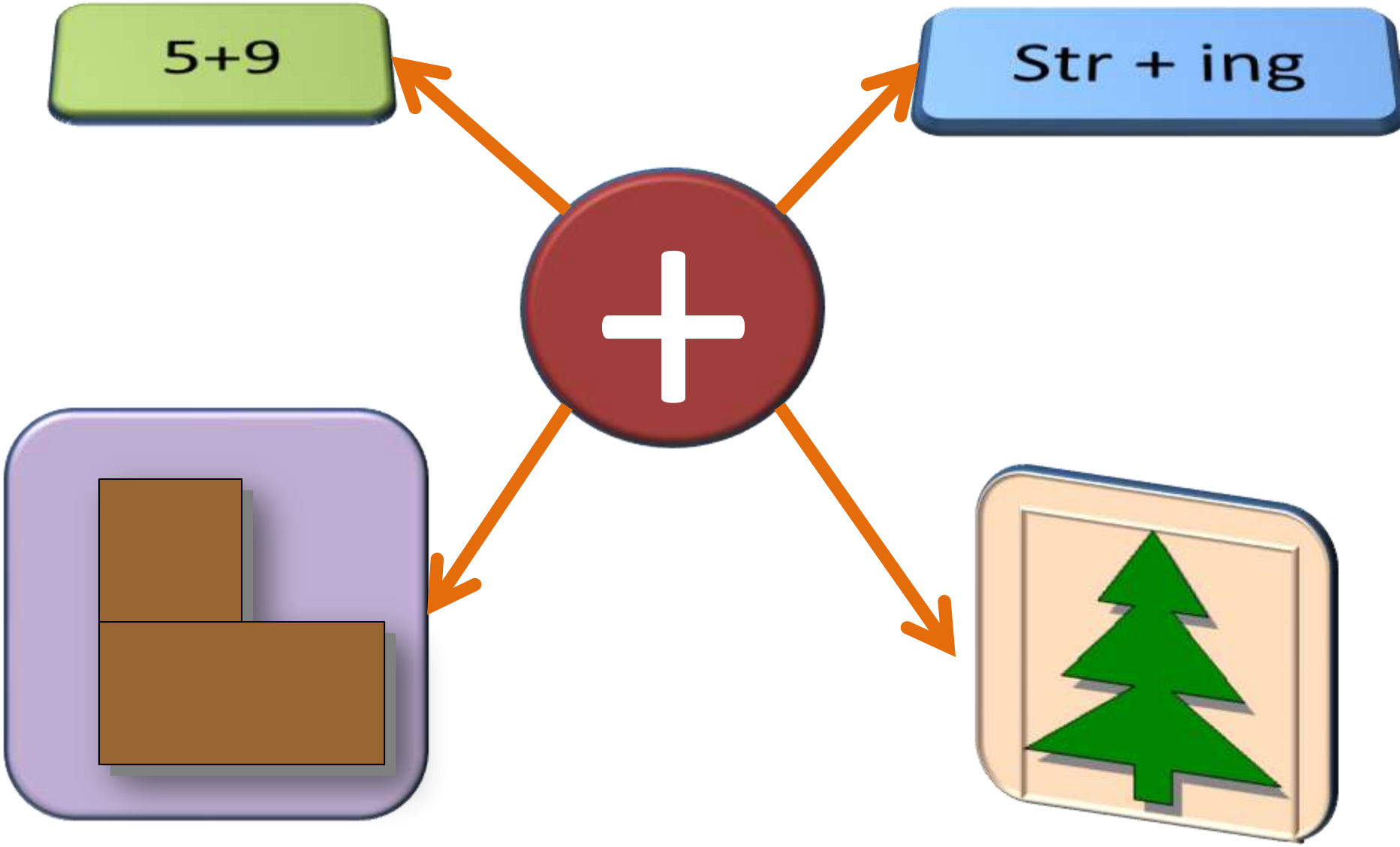


Polymorphism

- 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



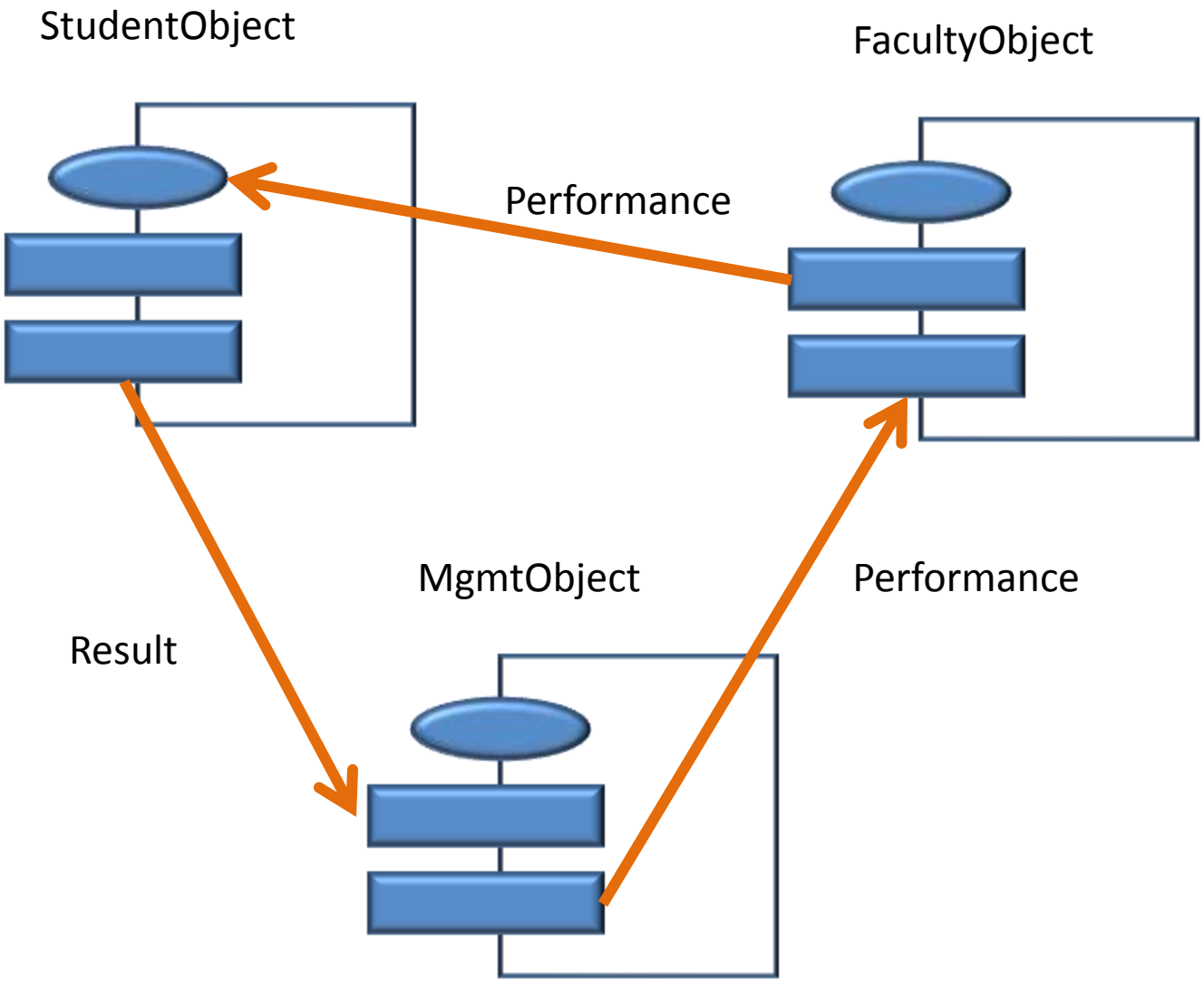
Dynamic Binding

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

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



Object Oriented Languages

1. **Object-Based programming Languages**

   Not support Inheritance & Dynamic Binding

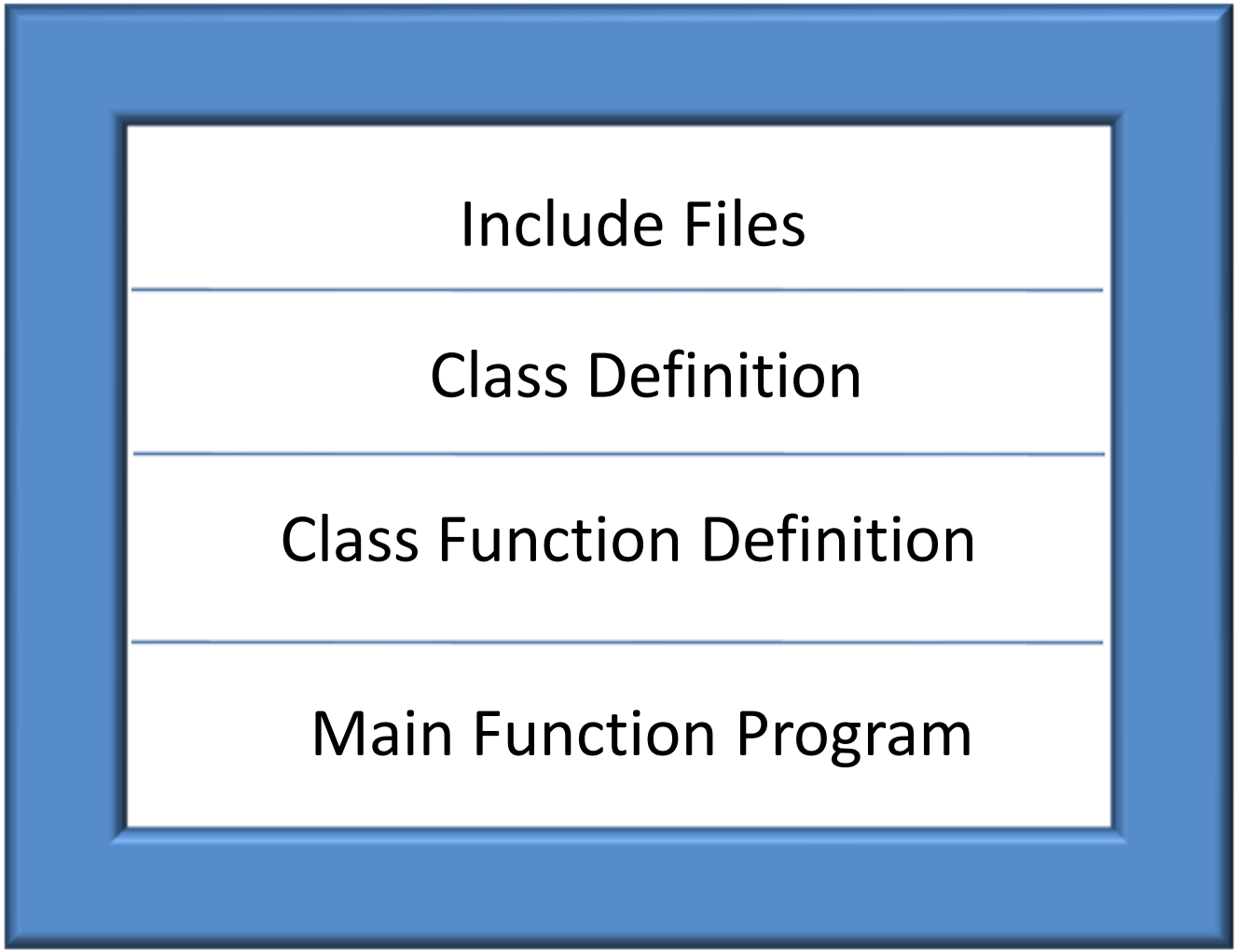
   Example: Ada

2. **Object-Oriented programming Languages**

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

2. C++ Overview

Structure of C++ Program



Simple C++ Program

```
// Hello World program

#include <iostream.h>

int main() {

  cout << "Hello World\n";

  return 0;
}
```

comment

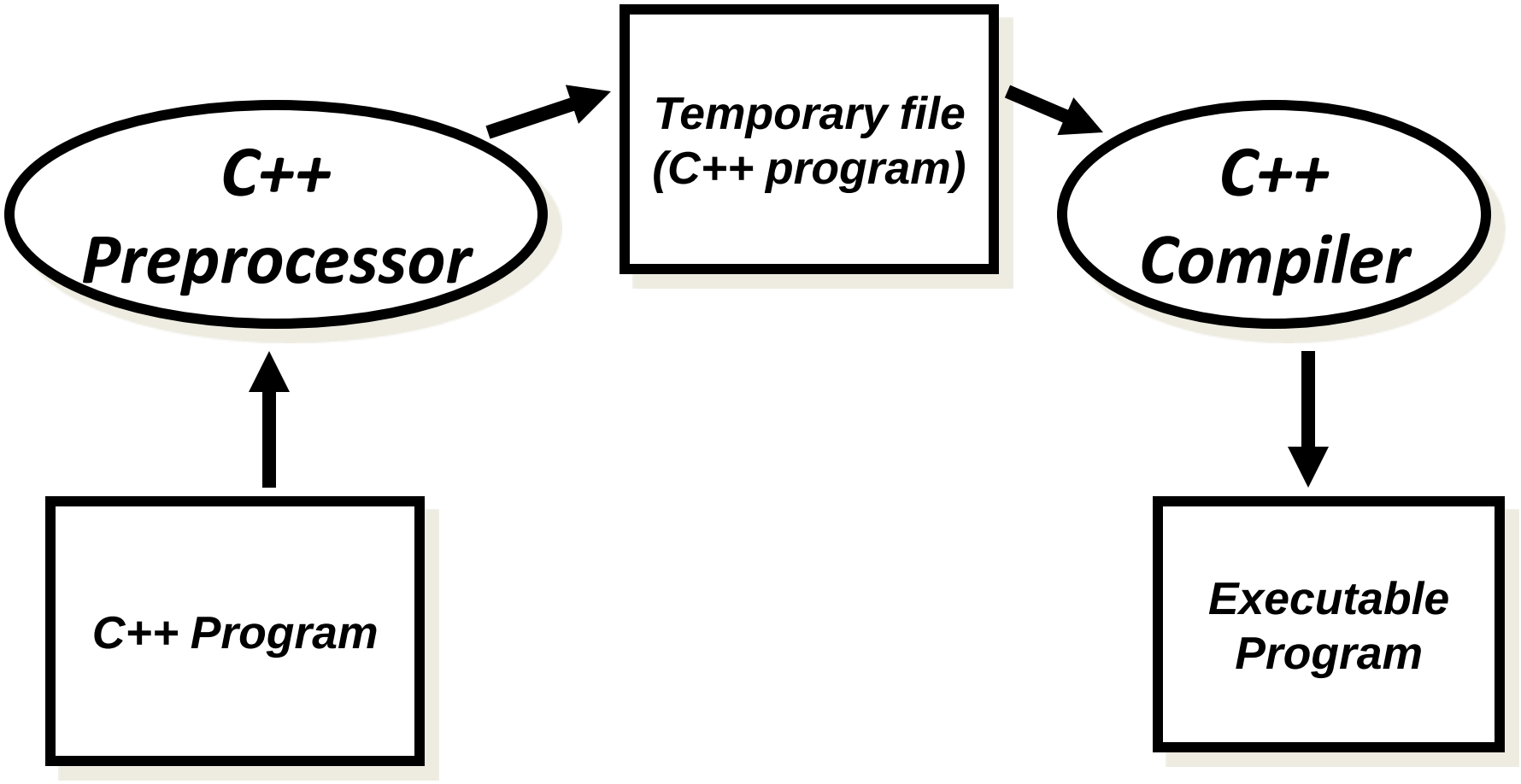
Allows access to an I/O library

Starts definition of special function main()

output (print) a string

Program returns a status code (0 means OK)

Preprocessing



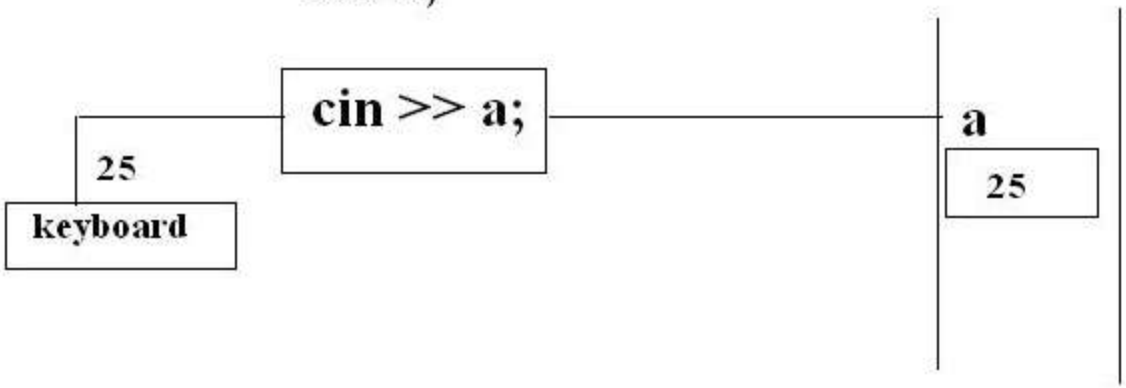
C++

Input and output of c++

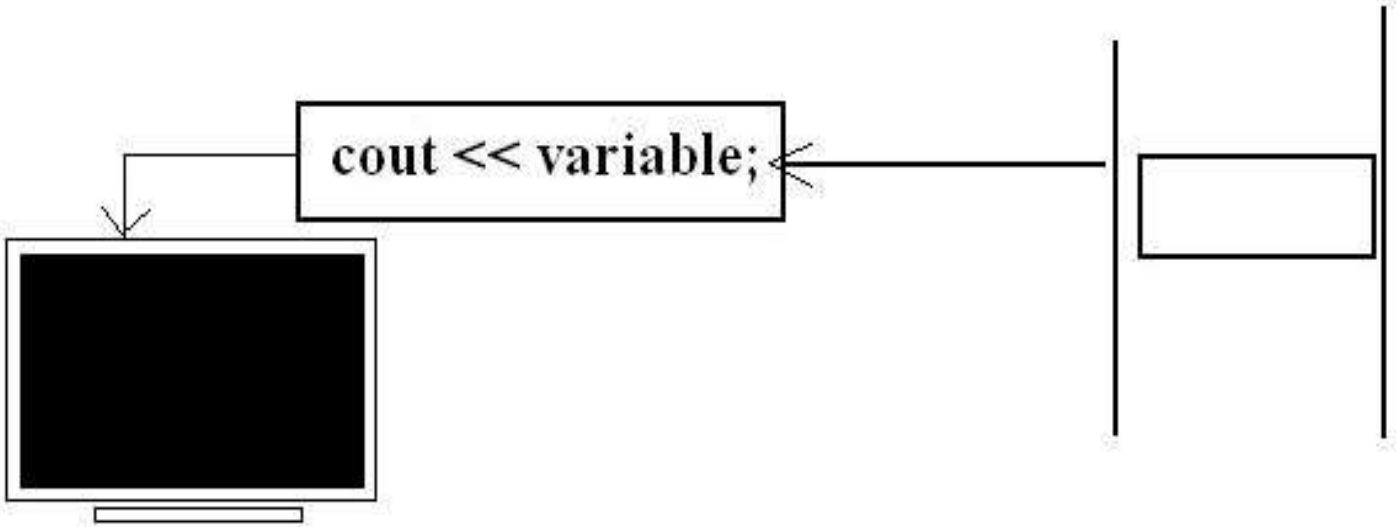
```
cin >> variable ;
```

example:

```
int a;
cin >> a;
```



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

C++ LANGUAGE

C++ Program is a collection of

- Tokens
- Comments
- White Space

C++ TOKENS

- RESERVED KEYWORDS
- IDENTIFIERS
- LITERALS
- OPERATORS
- SEPARATORS

RESERVED 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

IDENTIFIERS

- Programmer-designed tokens
- Meaningful & short
- Long enough to understand
- C++ rules for Identifiers
  - alphabets, digits, underscore
  - should not start with digits.
  - Case sensitive
  - Unlimited length
  - Declared anywhere

LITERALS

- Sequence of char. that represents constant values to be stored in variables

- C++ literals are:
  - Integer literals: 1,2,456,0xffff
  - Floating_point literals: 4.67,3.14E-05
  - Charater literals: ‘A’, ‘B’
  - String literals: “ABC”, “TOTAL”

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;

OPERATORS

- Is a symbol that takes more than one operands & operates on them to produce a result.
  - Arithmetic
  - Relational
  - Logical
  - Assignment
  - increment/Decrement
  - conditional
  - scope resolution(::)
  - special operators: new, delete, endl, setw

SEPARATORS

- Symbols used to indicate where groups of code are divided & arranged
- C++ separators:

  ()parentheses .. Methods, precedence in exp

  {} braces .. Arrays init., block of codes, scopes

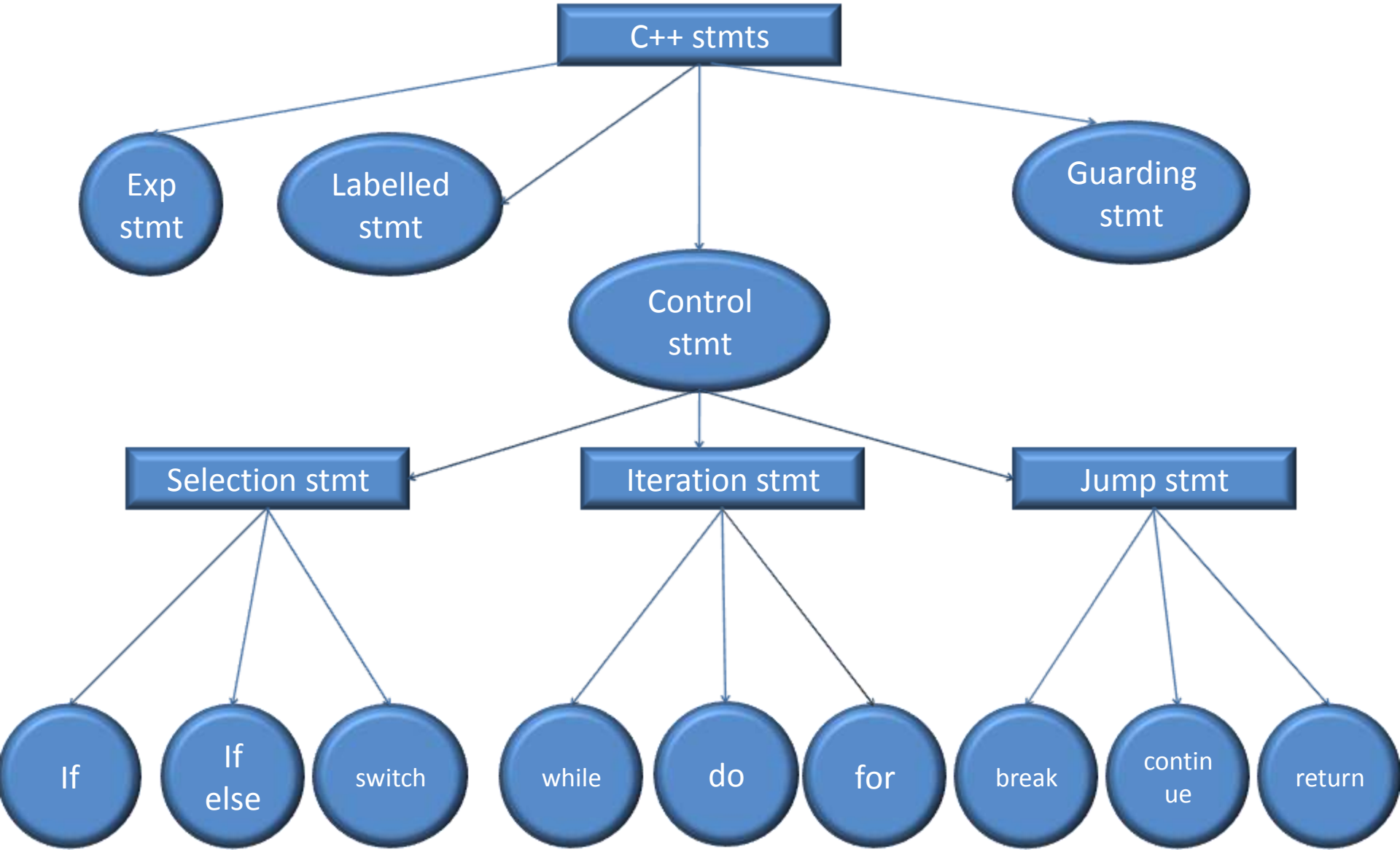
  ; semicolon

  ,comma.. Separate multiple identifiers, chain more than one stmt

  . Period.. Data members, methods

  []Brackets.. Array referencing/dereferencing

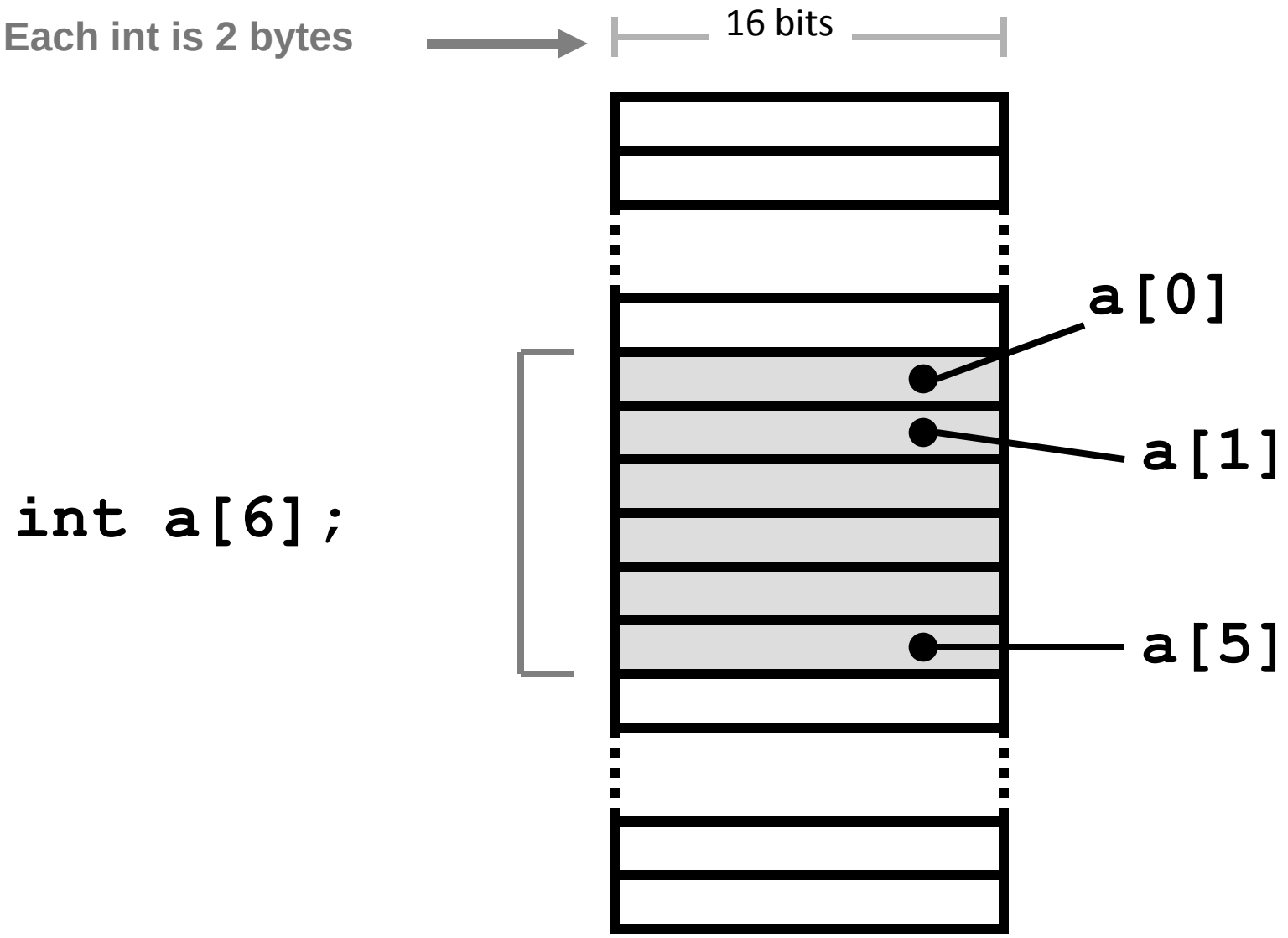
C++ STATEMENTS



Arrays in C++

- An array is a consecutive group of memory locations.
- An array is a collection of similar data elements.

Memory and Arrays



Array Initialization

We can initialize an array :

```
int a[5] = { 1,8,3,6,12};

double d[2] = { 0.707, 0.707};

char s[] = { 'R', 'P', 'I' };
```

NOTE:
Need not have to specify a size when initializing, the compiler will count automatically.

An array printing function

Can pass an array as a parameter.
Need not have to say how big it is!

```
void print_array(int a[], int len)
{
  for (int i=0;i<len;i++)
     cout << "[" << i << "] = "
          << a[i] << endl;
}
```

Arrays of `char` are special

- C++ provides a special way to deal with arrays of characters:

```
char str[] = "C++ char Arrays";
```

- char arrays can be initialized with string literals.

2-D Array: `int A[3][4]`

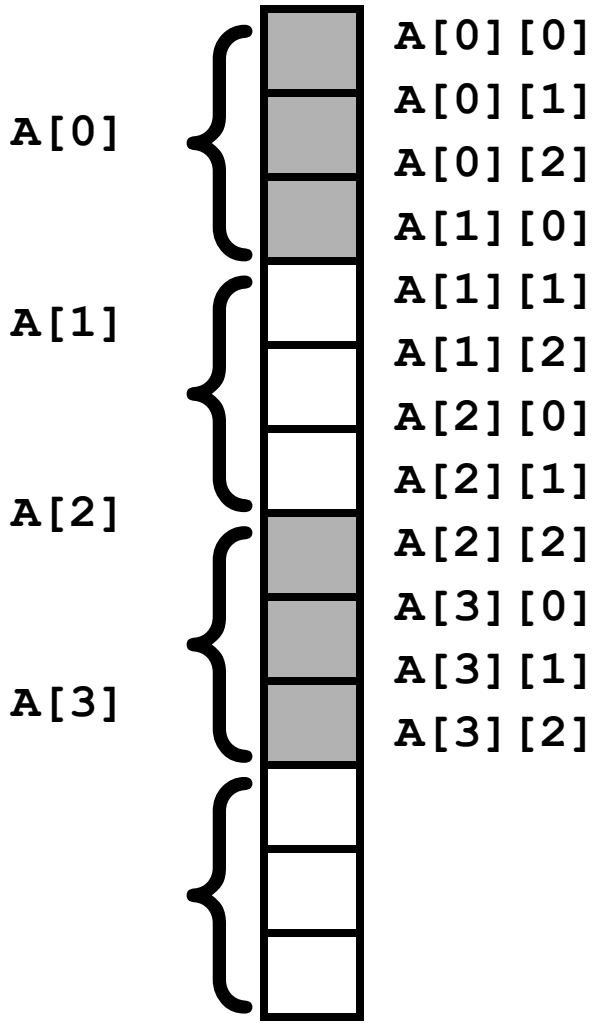
	Col 0	Col 1	Col 2	Col 3
Row 0	`A[0][0]`	`A[0][1]`	`A[0][2]`	`A[0][3]`
Row 1	`A[1][0]`	`A[1][1]`	`A[1][2]`	`A[1][3]`
Row 2	`A[2][0]`	`A[2][1]`	`A[2][2]`	`A[2][3]`

2-D Memory Organization

```
char A[4][3];
```

A is an array of size 4.

Each element of A is an array of 3 chars



2-D Array

Need not have to specify the size of the first dimension
But must include all other sizes!

```
double student_average( double g[][NumHW],
  int stu)
{
  double sum = 0.0;
  for (int i=0;i<NumHW;i++)
      sum += g[stu][i];

  return(sum/NumHW);
}
```

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

3. Modular Programming with Functions

Modular Programming

"The process of **splitting** of a **large program** into **small manageable tasks** and **designing** them **independently** is known as Modular **Programming or Divide-&-Conquer Technique.**"

C++ Functions

- **“Set of program statements** that can be processed independently.”

- Like in other languages, called **subroutines** or **procedures**.

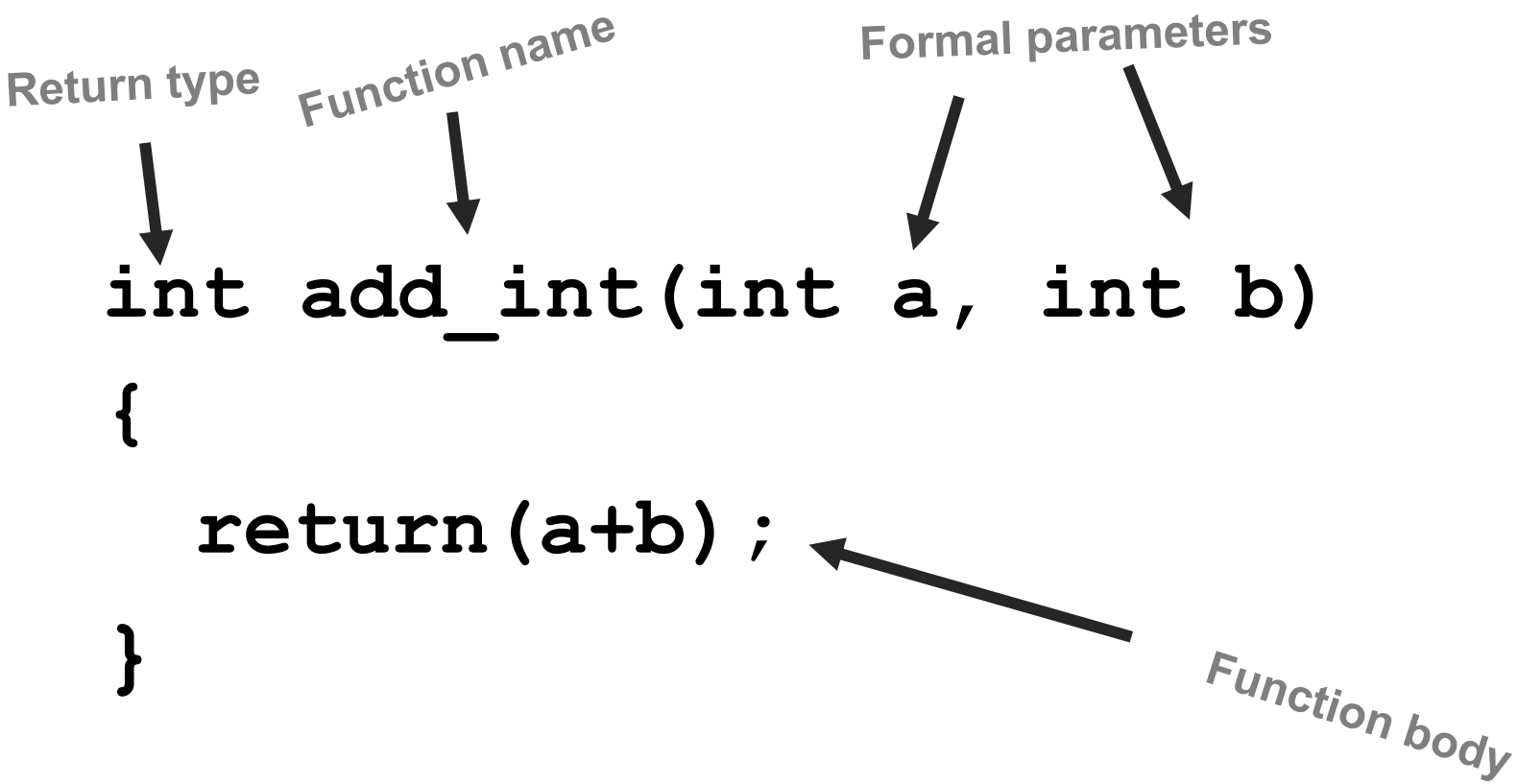
Advantages …?

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

Function Components

- Function Prototypes
- Function Definition(declaration & body)
- Function Parameters(formal parameters)
- return statement
- Function call(actual parameters)

Sample function



Using Math Library functions

- C++ includes a library of Math functions that we use.
- We have to know how to *call* these functions before we use them.
- We have to know what they return.
- We don't have to know how they work!
- **`#include <math.h>`**

Math Library Functions

```
ceil    floor

cos     sin     tan

exp     log     log10   pow

Etc.,
```

Function parameters

- The Formal parameters are local to the function.
  - When the function is called they will have the values *passed in*.
  - The function gets *a* ***copy*** of the values passed in.

Local variables

- Parameters and variables declared **inside** the definition of a function are *local*.
- They only exist inside the function body.
- Once the function returns, the variables no longer exist!

Block Variables

- We can also declare variables that exist only within the *body* of a compound statement

  (a block):

```
{
int res;
    …
    …
}
```

Global variables

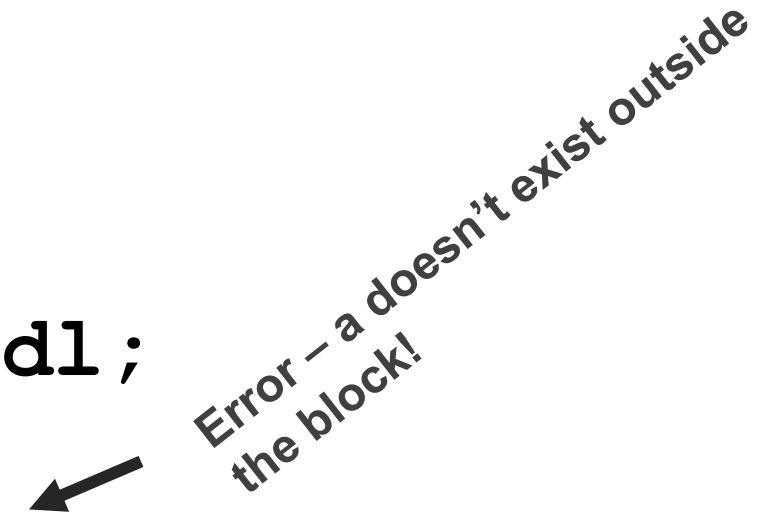
- We can declare variables **outside** of any function definition – these variables are ***global variables*****.**
- Any function can access/change global variables.
- Example: flag that indicates whether debugging information should be printed.

Scope

- The ***scope*** of a variable is the portion of a program where the variable has meaning (where it exists).
- A **global** variable has global (unlimited) scope.
- A **local** variable's scope is restricted to the function that declares the variable.
- A **block** variable's scope is restricted to the block in which the variable is declared.

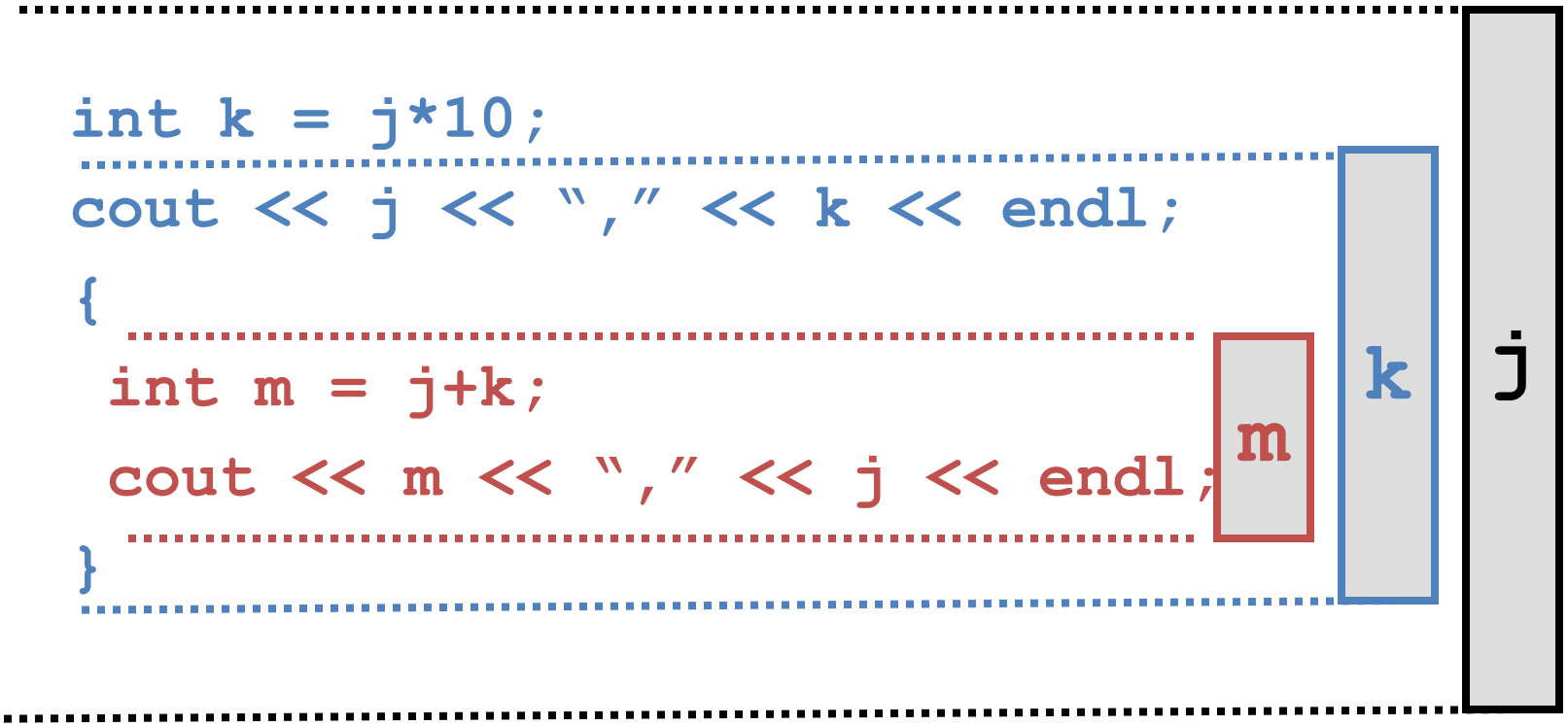
Block Scope

```
int main(void)
{
  int y;

  {
     int a = y;
     cout << a << endl;
  }
  cout << a << endl;
}
```



Nested Blocks

```
void example()
 {
  for (int j=0;j<10;j++)
  {
      int k = j*10;
      cout << j << ",” << k << endl;
      {
       int m = j+k;
       cout << m << ",” << j << endl;
      }
  }
}
```



Storage Class

- Each variable has a *storage class*.
  - Determines the **life time** during which the variable exists *in memory*.
  - Some variables are **created only once**
    - Global variables are created only once.
  - Some variables are **re-created** many times
    - Local variables are re-created each time a function is called.

Storage Classes

- **`auto`** – created each time the block in which they exist is *entered*.
- **`register`** – same as **`auto`**, but tells the compiler to make as fast as possible.
- **`static`** – created only once, even if it is a local variable.
- **`extern`** – global variable declared elsewhere.

Argument Passing

- Pass by Value
- Pass by Reference
- Program to swap two numbers
- Program to sort list of numbers

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.

Inline Function

- Pgm to find square of a given number using inline function

- Pgm to implement queue with its basic operations: enqueue(), dequeue(), display() where queue_full() and queue_empty() are inline functions

Function Overloading

" Multiple functions to share the **same name** with **different signatures(types** or **numbers)**."

```
int myfunc(int i)
{
return i;
}
```

```
int myfunc(int i, int j)
{
return i*j;
}
```

Function Templates

- “A **generic function** defines a general set of operations that will be applied to various types of data.”

- A single general procedure can be applied to a wide range of data.

Function Templates

- **Syntax:**

  template <class *Ttype*> *ret-type func-name(parameter list)*

  {

  // body of function

  }

Function Templates

- Pgm to sort integer list and float list using function template
- Pgm to add matrix elements using function template (for both int and double matrix)
- Pgm to find sum of a given list(int and float)using functoin template
- Pgm to implement integer and float stack using function template

Recursive Functions

- Pgm to find Fibonacci sequence upto n number
- Pgm to simulate Tower of Hanoi
- Pgm to add array of integers
- Pgm to multiply two natural numbers
- Pgm to find factorial of a given number

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

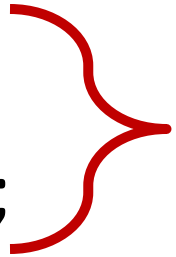
};

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.

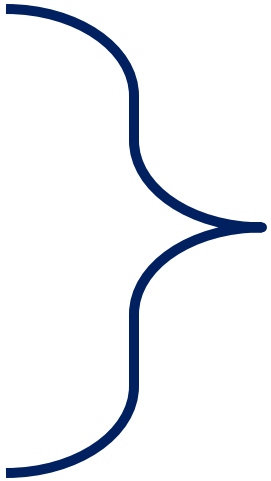
  protected - similar to private, but accessed by all the member functions of immediate derived class

  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();
};
```

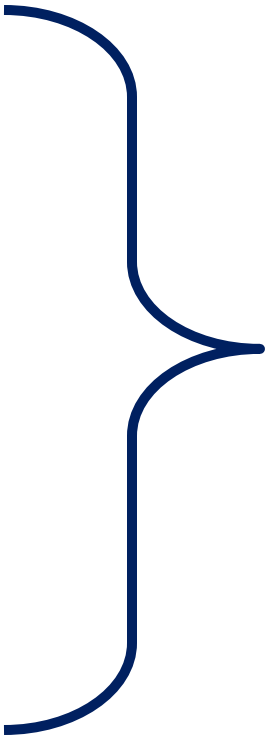


private / default visibility

Class specification

- **class** Student

```
{
   public:
      int st_id;
      char st_name[];
   public:
       void read_data();
       void print_data();
};
```



public visibility

Class Objects

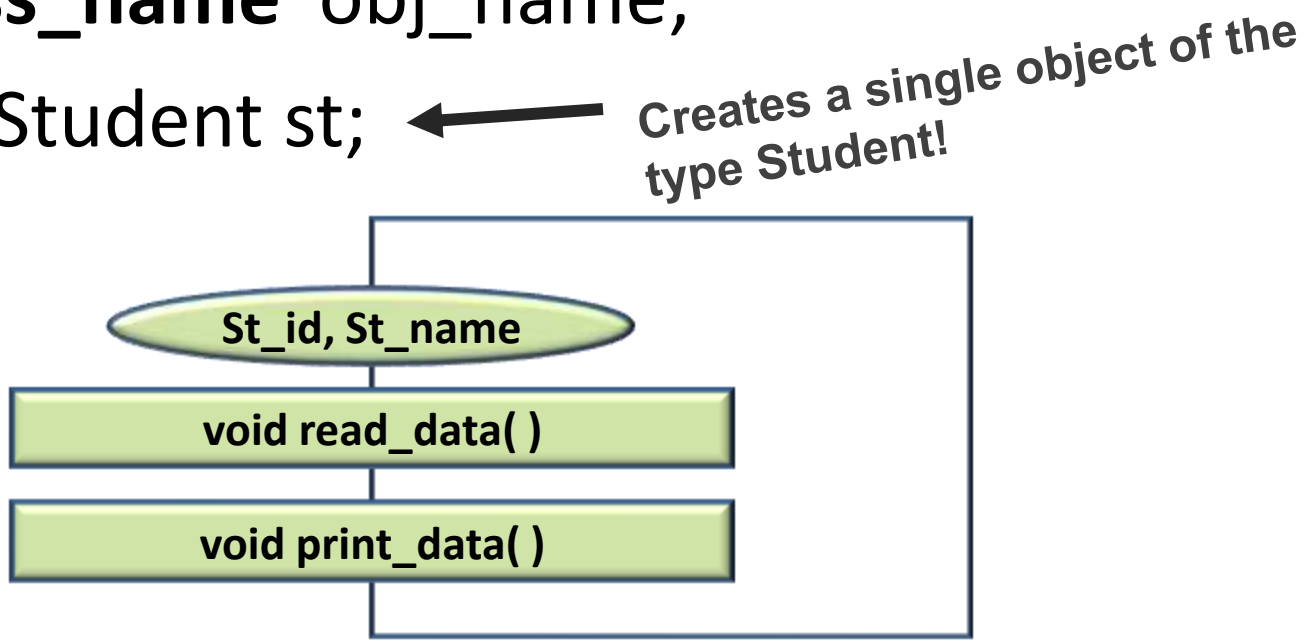
- **Object Instantiation:**

  The process of creating object of the type class

- **Syntax:**

  class_name obj_name;

  ex: Student st;



Class Object

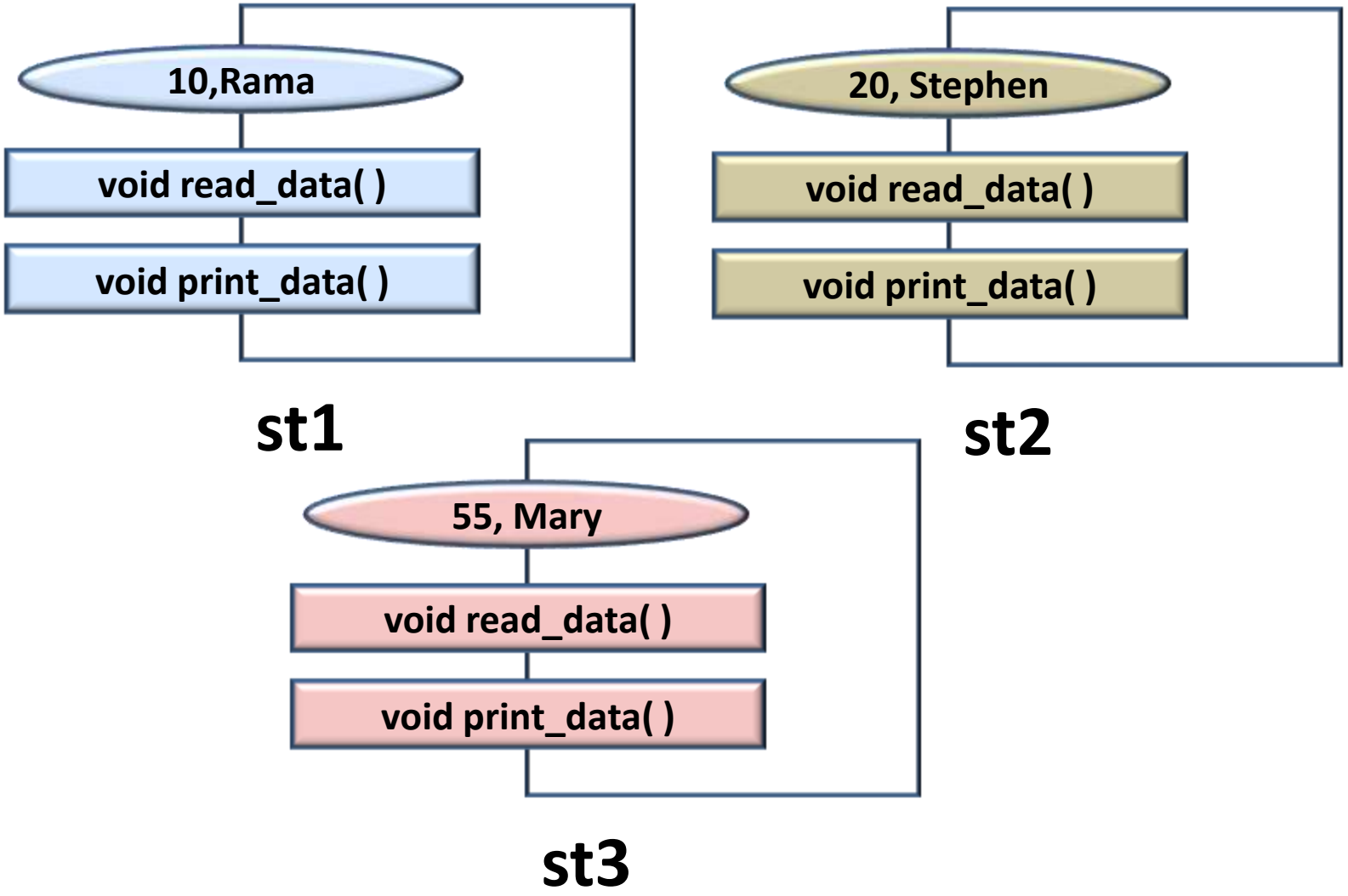
- **More of Objects**

  ex: Student st1;

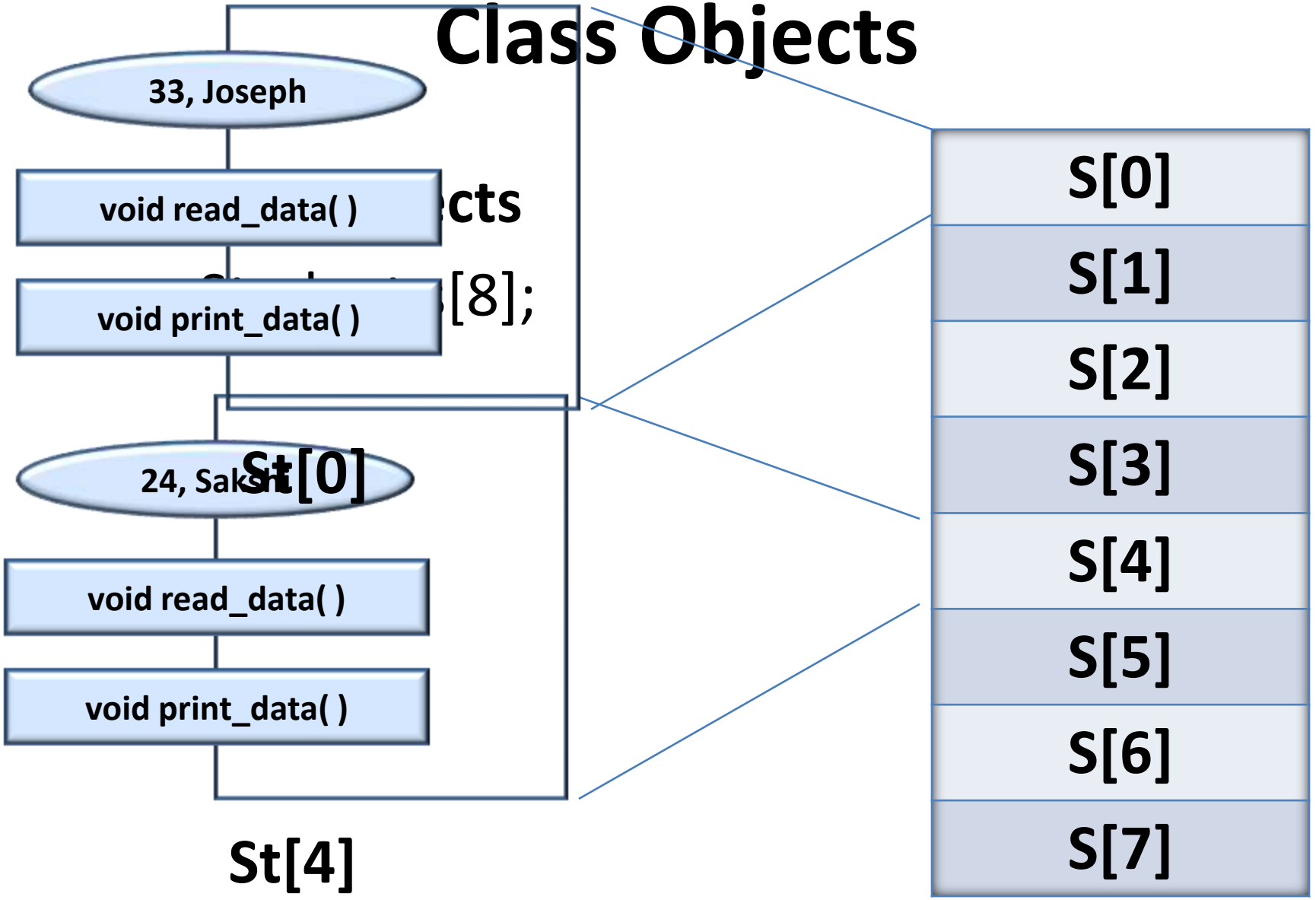
  Student st2;

  Student st3;

Class Objects



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

  }

Accessing Member Functions

- **Syntax: (single object)**

  obj_name . Memberfunction(act_parameters);

  ex: st.read( );

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

  obj_name[i] . Memberfunction(act_parameters);

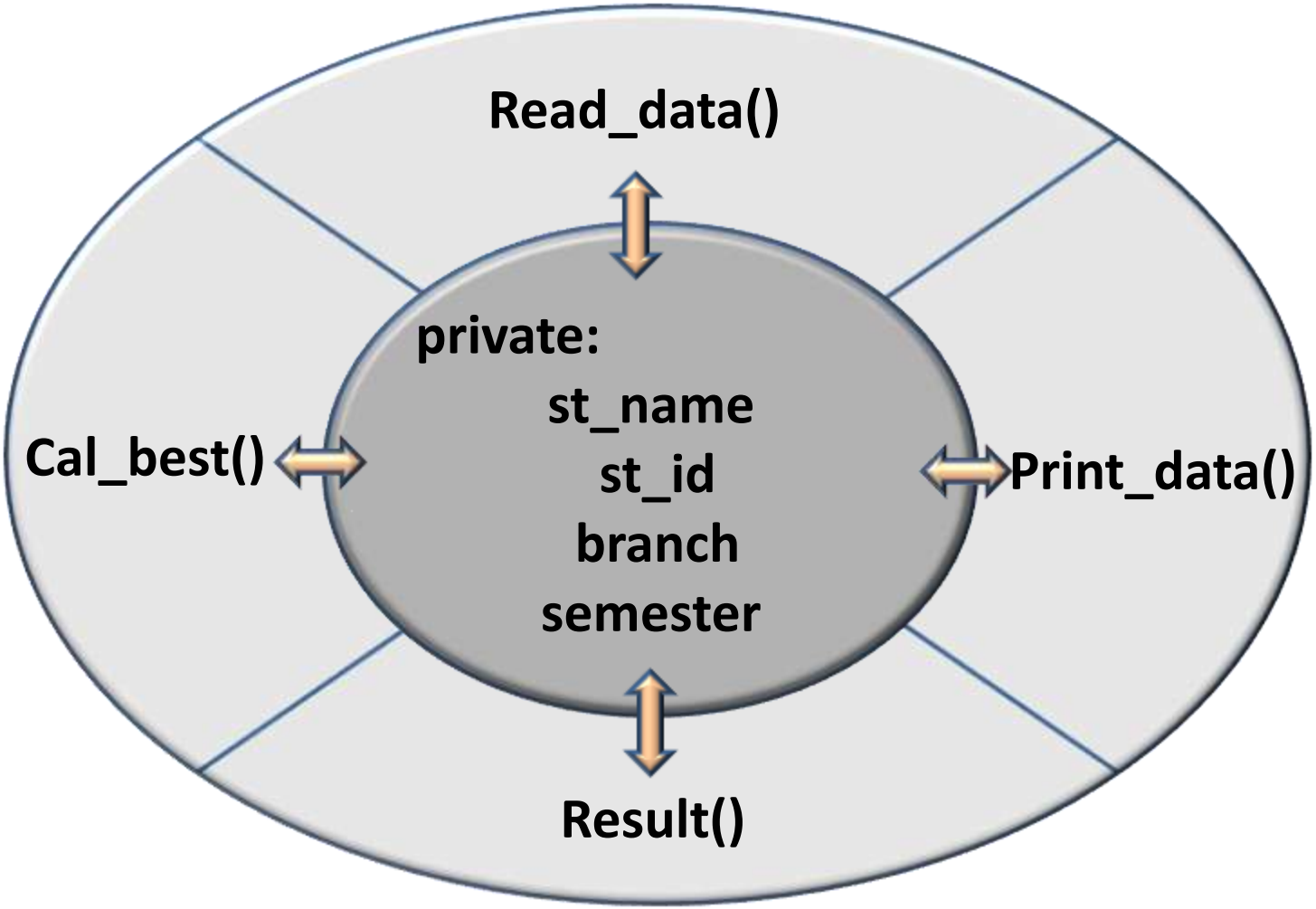
  ex: st[i].read( );

Data Hiding

- "**Data hiding** is the mechanism of **implementation details** of a class such a way that are **hidden** from the user."

- The concept of restricted access led programmers to write specialized **functions** for performing the operations on **hidden members** of the class.

Data Hiding


Read_data()
private:
st_name
st_id
branch
semester
Cal_best()
Print_data()
Result()

Data Hiding

- The **access specifie**r acts as the key strength **behind** the concept of **security.**

- Provides access to only to the member functions of class. Which prevents **unauthorized access**.

Data Hiding

Advantages:

- Makes Maintenance of Application Easier
- Improves the Understandability of the Application
- Enhanced Security

Inline Functions with Class

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

  inline ret_type fun_name(formal parameters)

  {

  function body

  }

Inline Functions with Class

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

  inline ret_type **class_name::**fun_name (formal parameters)

  {

  function body

  }

Inline Functions with Class

When to use Inline Function…..?

- If a function is very small.
- If the time spent to function call is more than the function body execution time.
- If function is called frequently.
- If fully developed & tested program is running slowly.

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

Synatax:

```
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

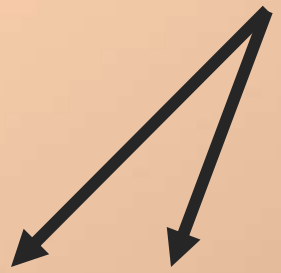
- Pgm to create a class **Addition** to add two integer values. Use constructor to initialize values.
- Pgm to create a class **Circle** to compute its area. Use constructor to **initialize** the data members.

Types of Constructors

- Parameterized constructors
- Overloaded constructors
- Constructors with default argument
- Copy constructors
- Dynamic constructors

Parameterized Constructors

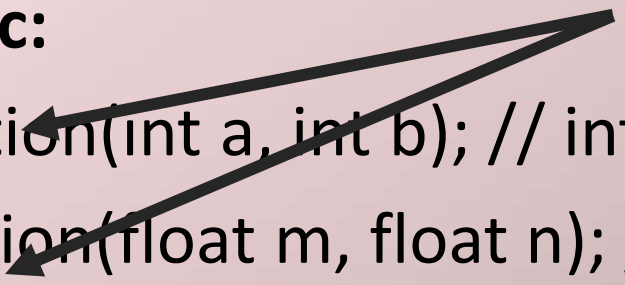
```
class Addition
{
        int num1;
        int num2;
        int res;
        public:
        Addition(int a, int b); // constructor
        void add( );
        void print();
};
```



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

Overloaded Constructors

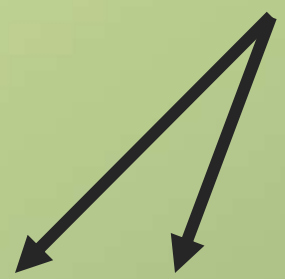
```
class Addition
{
    int num1,num2,res;
    float num3, num4, f_res;
    public:
    Addition(int a, int b); // int constructor
    Addition(float m, float n); //float constructor
    void add_int( );
    void add_float();
    void print();
};
```



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

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

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

Dynamic Constructors

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

Destructors

- "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

Destructors

Synatax:

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

Example:

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

Programs for Implementation

- Pgm to create a class **Complex** to add two complex numbers using **parmeterized** constructor.
- Pgm to create a class **Complex** to add two complex numbers using **copy constructor**.
- Pgm to create a class **Complex** to add dynamically created integer to a complex number using **Dynamic constructor.**

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();
}
```

Multiple Classes

Synatax:

```
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:
    viod init_test()
          {
           m.t[0]=25;
           m.t[1]=22;
           m.t[2]=24;
          }
};
```

Nested Classes

Synatax:

```
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;}
};
```

Program for Implemention

Pgm to create a class STUDENT with properties: id, name, semester & three TEST marks(which should be defined as a separate class suitable properties) . The behaviors of the STUDENT should include:

1. To read the data
2. To Calculate average of best two test marks
3. To print the data

Program for Implementation

Pgm to create a class EMPLOYEE with properties: id, name, designation & date of birth as DOB (which should be defined as inner class with suitable properties) . The behaviors of EMPLOYEE should include:

1. To read the data
2. To Calculate age
3. To print the data

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.

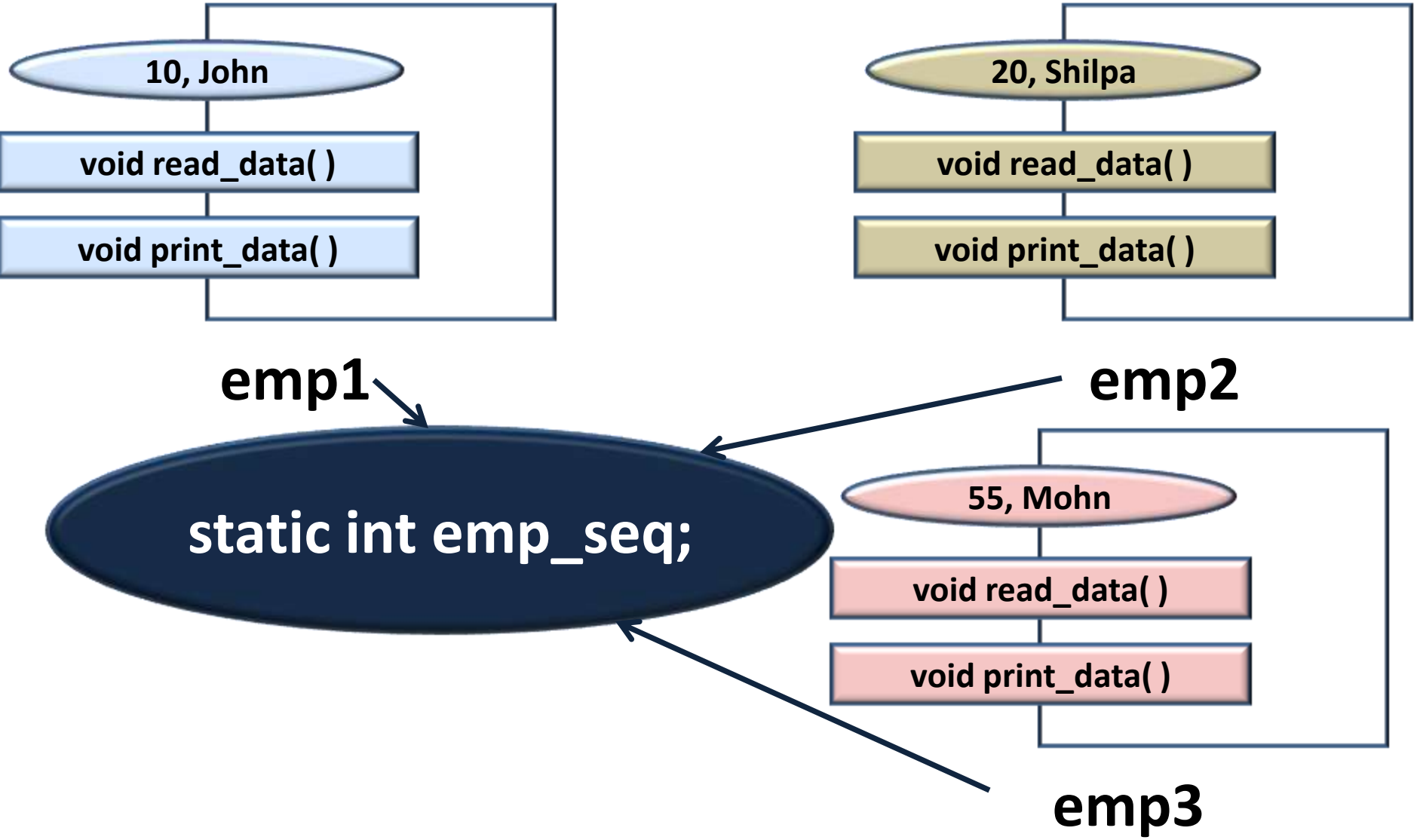
Static Data Members

- 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.

Synatax:

```
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.**

Scope Resolution Operator

```
int i; // global i
void f()
{
int i; // local i
i = 10; // uses local i
.
.
.
}
```

```
int i; // global i
void f()
{
int i; // local i
::i = 10; // now refers to global i
.
.
.
}
```

Solution…..?

Scope Resolution Operator

- The **:: operator** links a class name with a member name in order to tell the compiler **what class the member belongs to**.

- **Has another related use:**

  Allows to access to a name in an enclosing scope that is **"hidden"** by **a local declaration** of the same name.

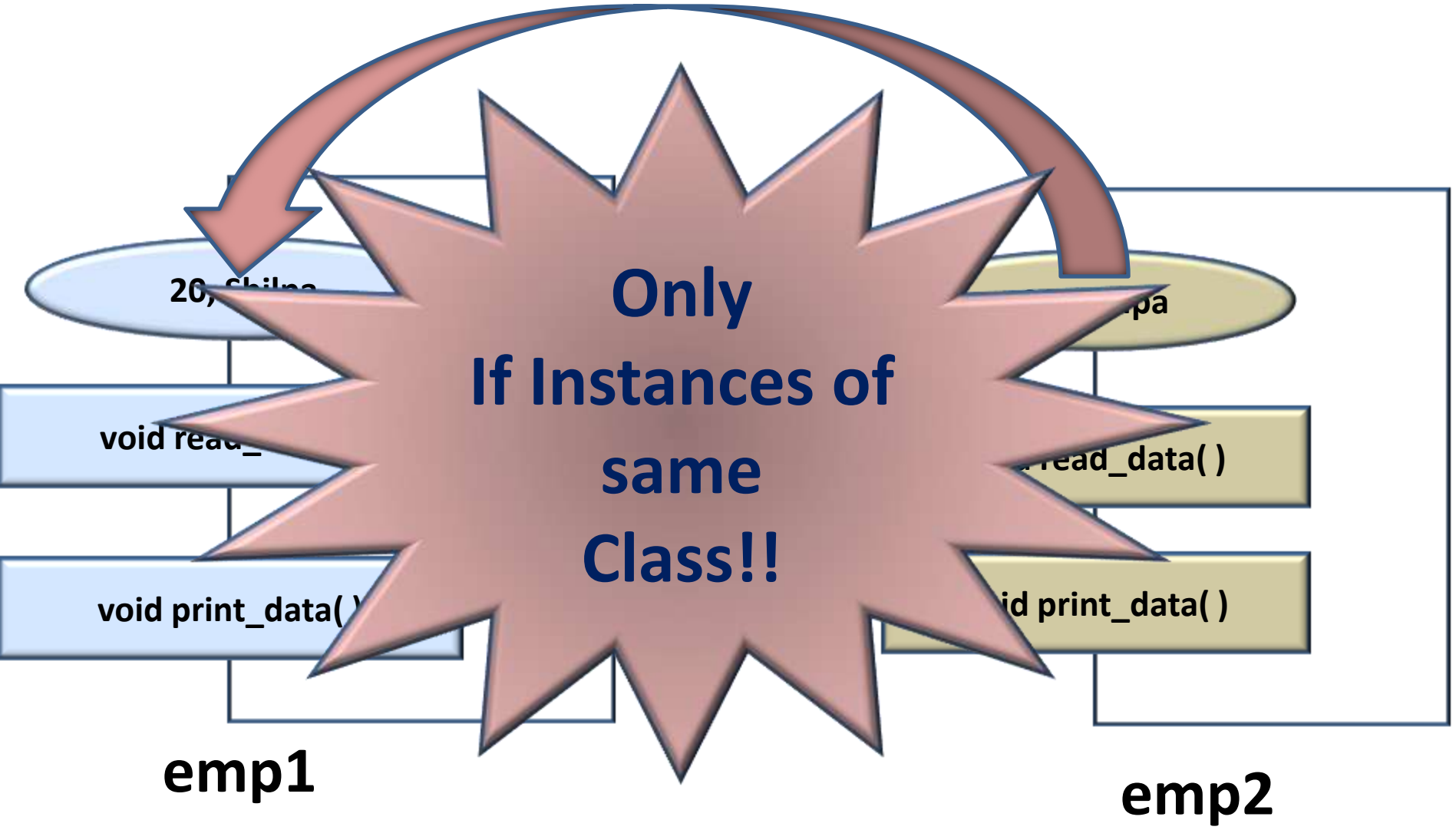
5. Objects with Functions

Passing Objects as Arguments

- 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.

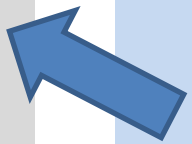
Object Assignment



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) ;
            ……
            ……
}
```

Program for Implementation

Pgm to create class PAPER with its properties: width & height. Find the characteristics of a magzine Cover: width, height, perimeter and Area using passing object as argument(s) .

Friend Functions

- “Friend function is a **non-member function** which can **access** the **private members** of a class”.

- To declare a **friend function,** its **prototype** should be included within the class, preceding it with the keyword **friend.**

Friend Functions

Example:

```
class myclass
 {
    int a, b;
    public:
    friend int sum(myclass x);
    void set_val(int i, int j);
};
```

Syntax:

```
class class_name
{
//class definition
public:
friend rdt fun_name(formal parameters);
};
```

Friend Functions

Advantages…?

- When we overload operators.
- When we create I/O overloaded functions.
- When two or more classes members are interrelated and to carry out the communication to other parts of the program.

Program for Implementation

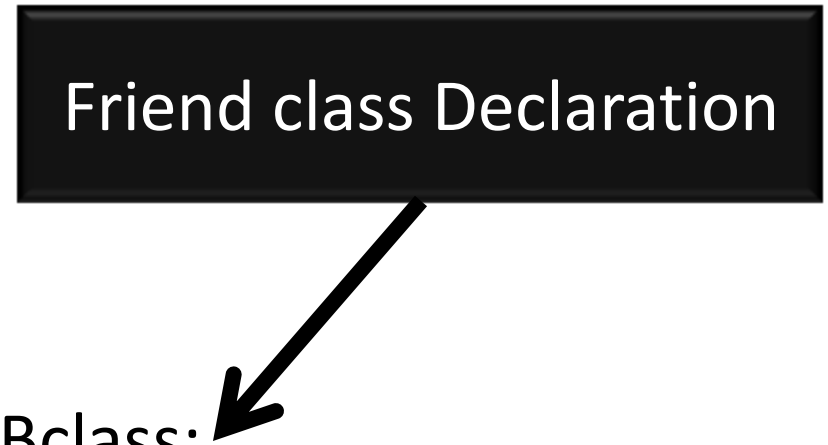
Pgm to create a class ACCOUNTS with function read() to input sales and purchase details. Create a Friend function to print total tax to pay. Assume 4% of profit is tax.

Friend Classes

"A class can be a friend of another class, allowing access to the **protected** and **private** members of the class in which is defined."

Friend class and all of its member functions have access to the private members defined within the that class.

Friend Classes




```
class Aclass
{
public :
        :
friend class Bclass;
private :
int Avar;
};
```

```
class Bclass
{
public :
        :
  void fn1(Aclass ac)
  {
    Bvar = ac. Avar;
  }
  private :
   int Bvar;
};
```

Program for Implementation

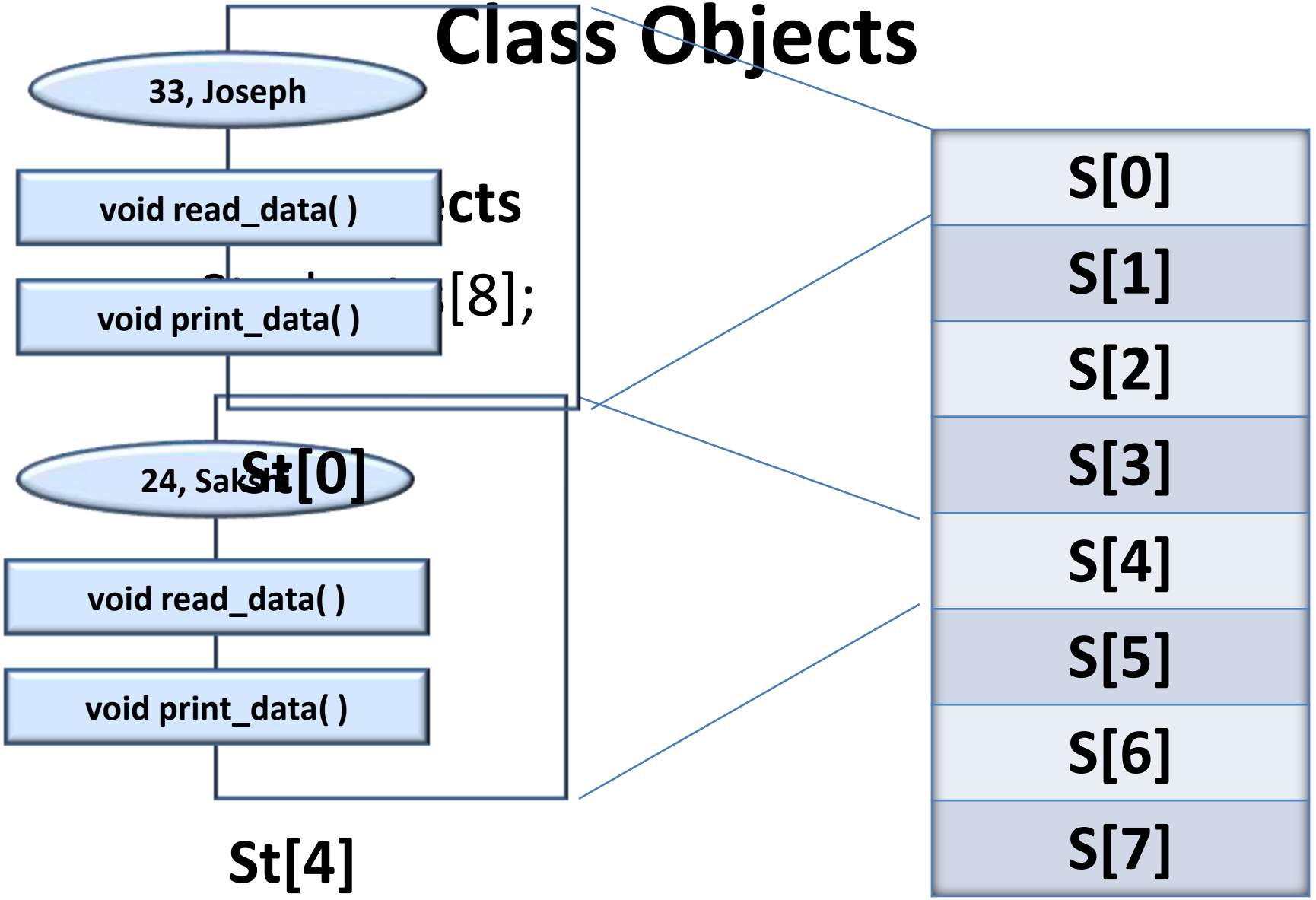
Program to create a class **MinValue,** is a **friend** of class **TwoVaules** contains two private members **a** and **b**. Where **minimum()** is a member function of MinValue that finds out the minimum of two values by **accessing data members** of class TwoValue. Initialize the values for two integer numbers through **constructor**.

Arrays of Objects

- **Several** objects of the **same class** can be declared as an array and used just like an array of any other data type.

- The **syntax** for declaring and using an object array is **exactly** the **same** as it is for any other type of array.

Class Objects



Program for Implementation

Pgm to create a class **STUDENT** with properties: rollno, name, IAmarks for 6 subjects, EndExamMarks for 6 subjects. Create function **read()** to read the details of student, **calc_percent()** to calculate the percentage marks for each student and **show()** to display the details of all students. Assume the following things to calculate percentage:

Total_sub_marks = IAmarks + EndExamMarks

& **Total_marks** = Addition of **Total_sub_marks** of 6 subjects.

Dynamic Objects

" Dynamic objects are objects that are **created** / **Instantiated** at the **run time** by the class".

- They are **Live Objects**, initialized with necessary data at run time.

- Its life time is explicitly managed by the program(should be handled by programmer).

Dynamic Objects

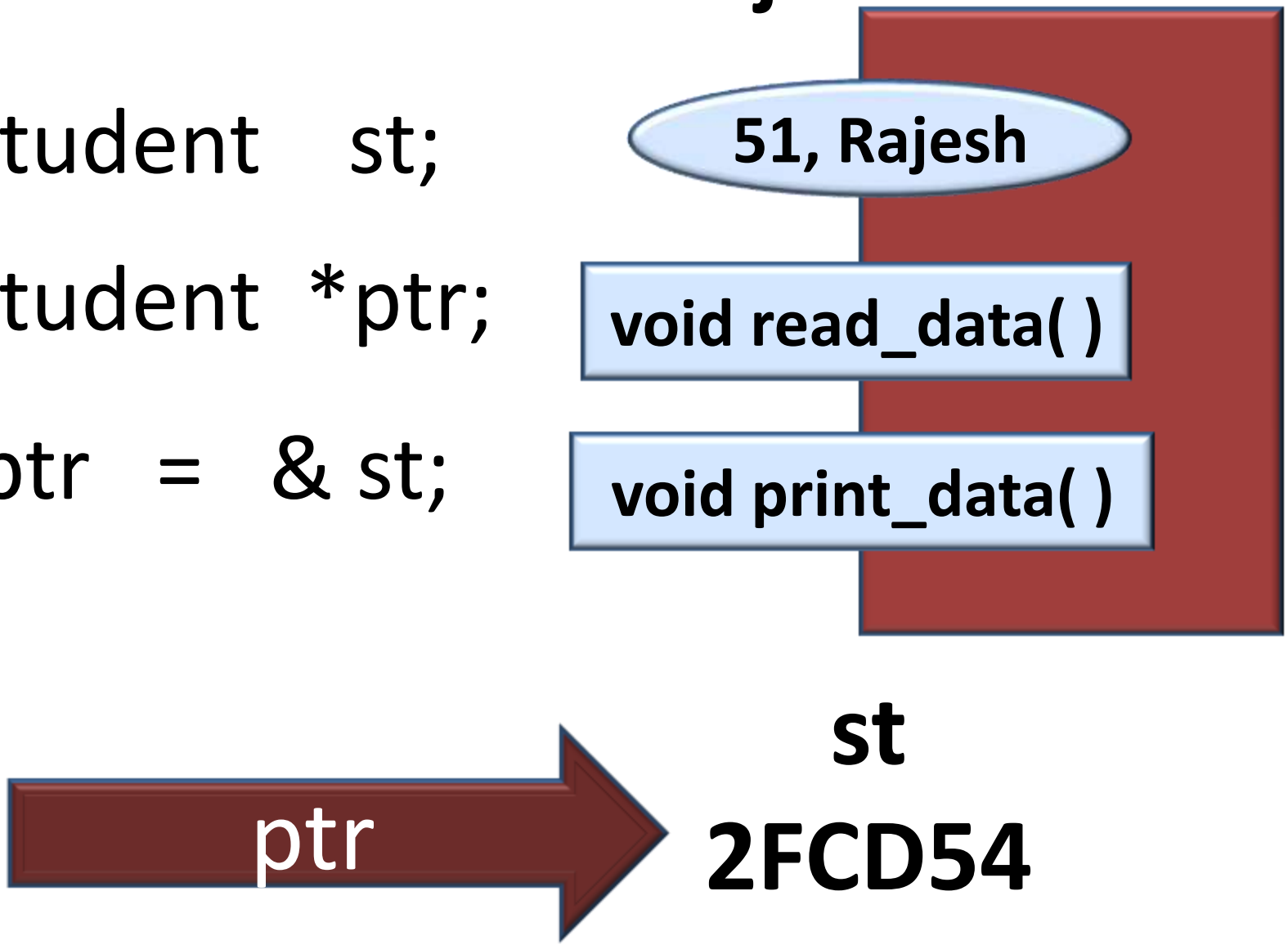
- The **new** operator is used to create dynamic objects.

- The **delete** operator is used to release the memory allocated to the dynamic objects.

NOTE:
C++ does not have **Default Garbage Collector.**

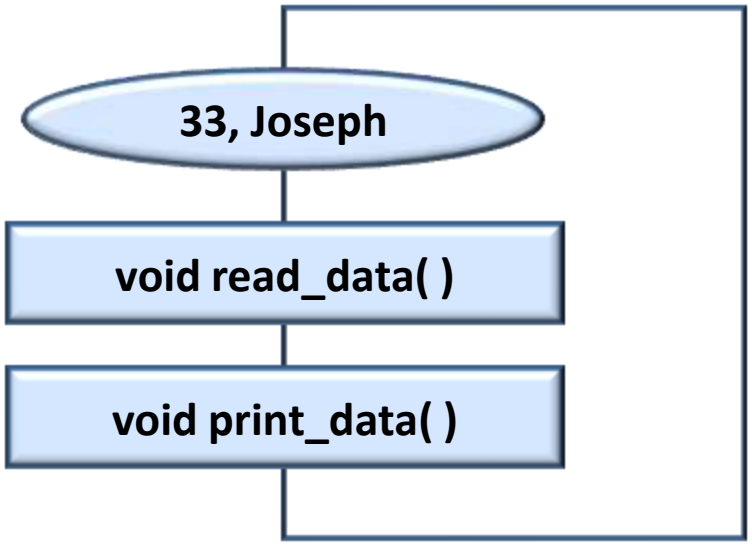
Pointers to Objects

```
student   st;

student  *ptr;

 ptr   =   & st;
```



Pointers to Objects

"**Pointers** can be defined to **hold** the **address** of an **object**, which is created statically or dynamically".



st

2FCDA4

Statically created object:

```
student     *stp;
stp  =  &st;
```

Dynamically created object:

```
student   *stp;
stp    =    new student;
```

Pointers to Objects

- **Accessing Members of objects:**

  Syntax:

  ptr_ob j → member_name;

  ptr_obj → memberfunction_name( );

  Example:

  stp → st_name;

  stp → read_data ( );

The *this* Pointer

"The **this** pointer points to the object that invoked the function".

- When a member function is called **with** an **object**, it is **automatically passed** an implicit argument that is a **pointer** to the invoking object (that is, the object on which the function is called).

The *this* Pointer

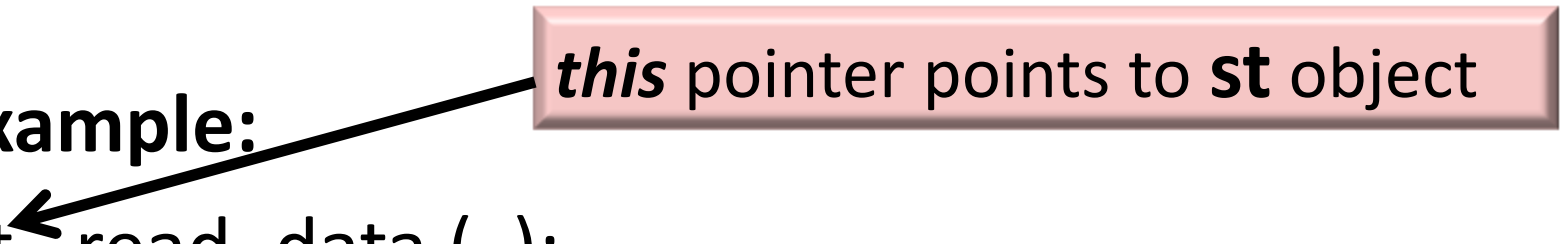
- **Accessing Members of objects:**

  Syntax:

  obj . memberfunction_name( );

  Example:

  st . read_data ( );

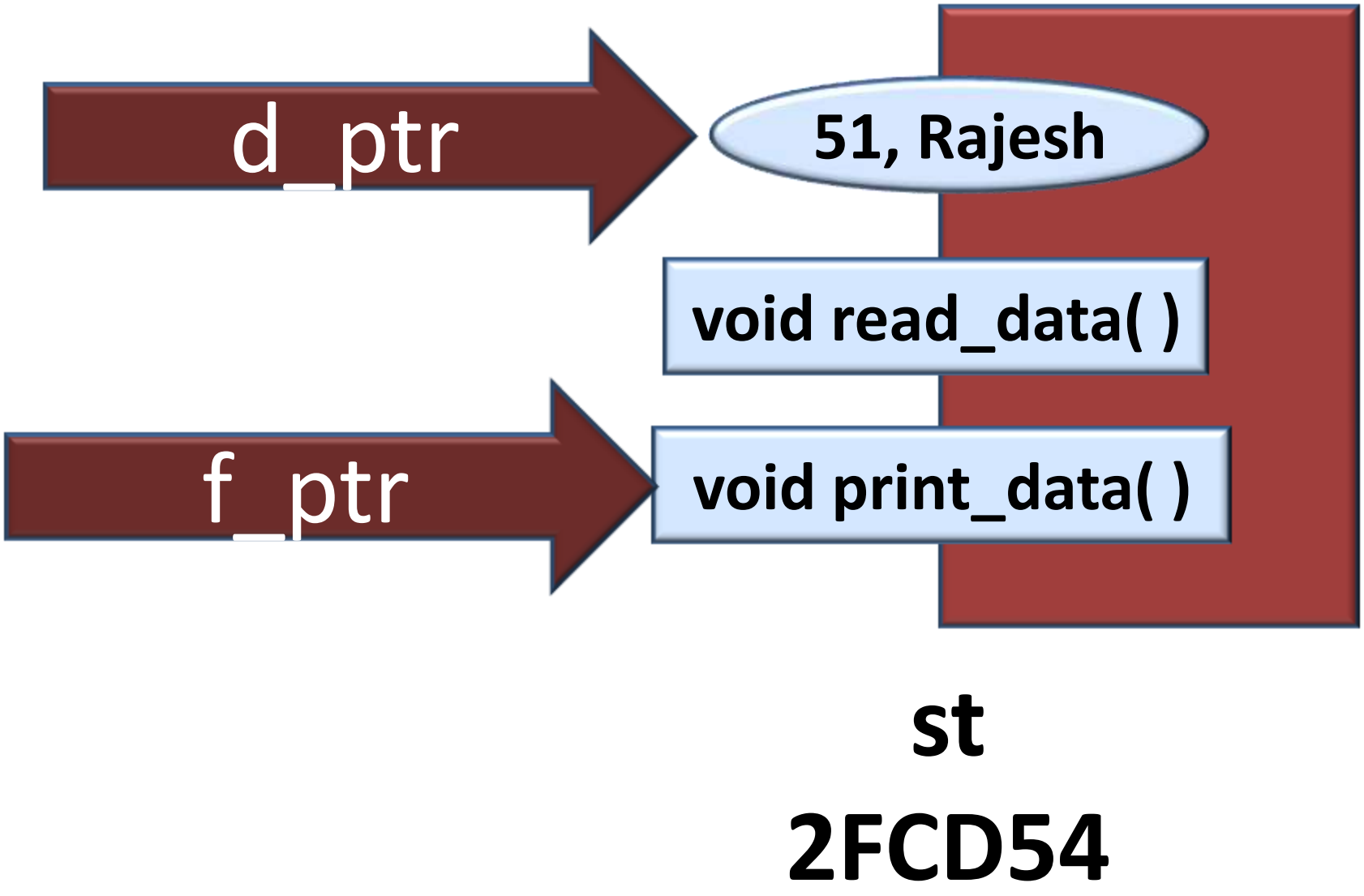


Program for Implementation

- Pgm to find factorial of a number using ***this*** pointer.

- Pgm to generate Fibonacci sequence upto nth value using ***this*** pointer.

Pointer to Class Member



Pointer to Class Member

"A special type of pointer that "points" generically to a **member of a class**, not to a **specific instance** of that **member** in an object".

Pointer to a class member is also called **pointer-to-member.**

Pointer to Class Member

- It provides only an **offset** into an **object** of the member's class at which that member can be found.

- Member pointers are not **true** pointers, the **.** and **->** cannot be applied to them.

- A pointer to a member is **not** the same as a normal **C++ pointer**.

Pointer to Class Member

- **To access a member of a class:**

  special pointer-to-member operators

  1) **.***
  2) **–>***

Pointer to Class Member

- **Syntax to create pointer to data member of a class:**

  Data_type class_name ::* data_member_ptr;

  int student::*d_ptr;

- **Syntax to create pointer to member function of a class:**

  rtn_dt (class_name::* mem_func_ptr)(arguments);

  int (student::*f_ptr)();

Programs for Implementation

- Pgm to find number of vowels in a given string using pointer to class members.

- Pgm to find square of a given number if it is even else compute its cube, using pointer to class members.

Operator Overloading

- “Operator overloading is the **ability** to **tell** the **compiler** how to perform a certain **operation** when its corresponding **operator** is used on one or more variables.”

- Operator overloading is **closely** related to **function overloading**.

- Allows the full **integration** of **new class types** into the programming environment.

Operator Overloading

- Overloading of operators are achieved by creating **operator function**.

- "An ***operator function*** defines the operations that the overloaded operator can perform relative to the class".

- An operator function is created using the keyword **operator.**

Operator Overloading

- Operator functions can be either **members** or **nonmembers** of a class.

- **Non-member** operator functions are always **friend functions** of the class.

Operator Overloading

- **Overloadable operators are:**

  + - * / = < > += -= *= /= << >>

  <<= >>= == != <= >= ++ -- % & ^

  ! | ~ &= ^= |= && || %= [] ()

  new delete

Operator Overloading

- **Creating a Member Operator Function**

Within Class:

ret-type operator#(argument-list);

Outside Class:

ret-type class-name::operator#(arg-list)

{

// operations

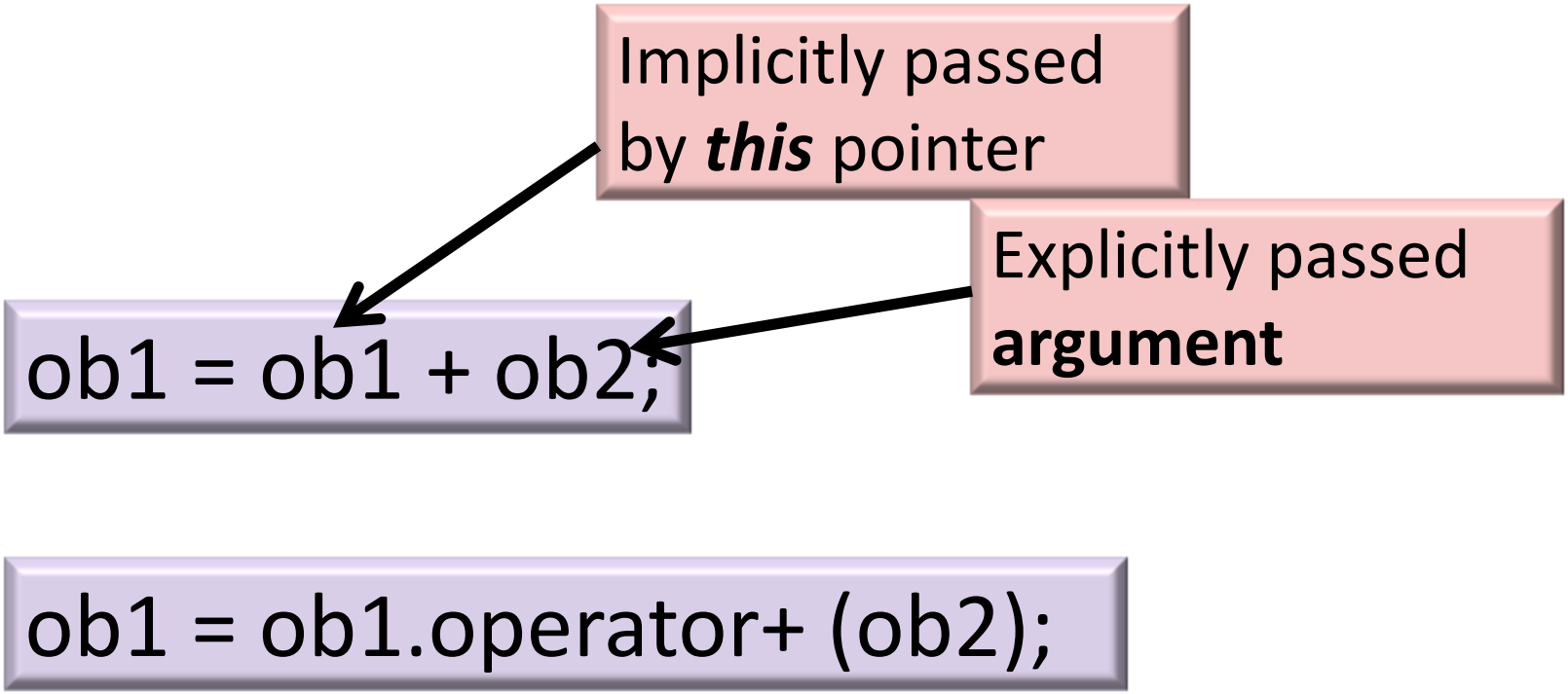
}

Operator Overloading

Example:

```
comp    comp::operator+(comp      op2  )
{
comp temp;
temp.real = op2.real + real;
temp.img = op2.img + img;
return     temp;
}
```

Operator Overloading



Operator Overloading Restrictions

- Should not alter the **precedence** of an operator.
- Should not change the **number of operands** that an operator takes.
- Operator functions cannot have **default arguments**.
- Operators cannot be overloaded are:

  . : : .* ?: sizeof

Why not . : : .* ?: sizeof() operators?

- The restriction is for safety. If we overload . operator then we cant access member in normal way.

- The :? takes 3 argument rather than 2 or 1. There is no mechanism available by which we can pass 3 parameter during operator overloading.

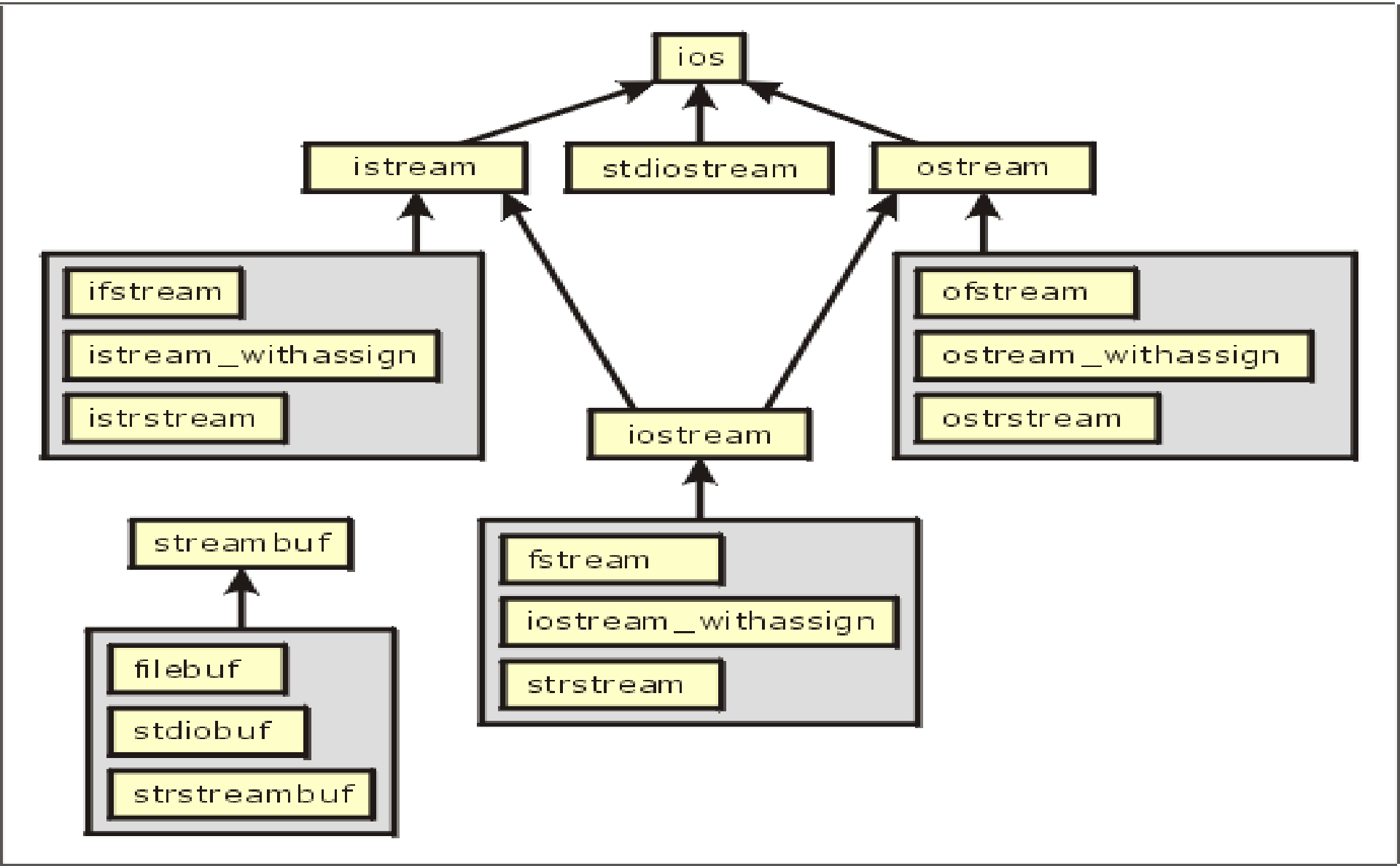
Why not . : : .* ?: sizeof() operators?

- Example:

  c = a+b - both a & b actually refer to some memory location, so "+" operator can be overloaded,

  but the "." operator, like a.i actually refers to the name of the variable from whom the memory location has to be resolved at time and thus it cannot be overloaded.

I/O Stream Class Hierarchy



Overloading >>

- **Prototype:**

  friend istream& operator >>(istream&, Matrix&);

- **Example:**

```
istream& operator >> (istream& in, Matrix& m)
{
    for (int i=0; i<row*col; i++)
    {
        in >> m[i];
    }
    return in;
}
```

```
void main()
{
  :
Cin>>mobj;
  :
}
```

Overloading <<

- **Prototype:**

  friend ostream& operator <<(ostream&, Matrix&);

- **Example:**

```
ostream& operator << (ostream& out, Matrix& m)
{
    for (int i=0; i<row; i++)
    {
        for (int j=0; j<col; j++)
        {   out<< m.a[i][j] << "\t";
        }
        out << endl;
    }
```

```
void main()
{
    :
cout<<mobj;
    :
}
```

Program for Implementation

- Pgm to create a class Matrix assuming its properties and add two matrices by overloading + operator. Read and display the matrices by overloading input(>>) & output(<<) operators respectively.

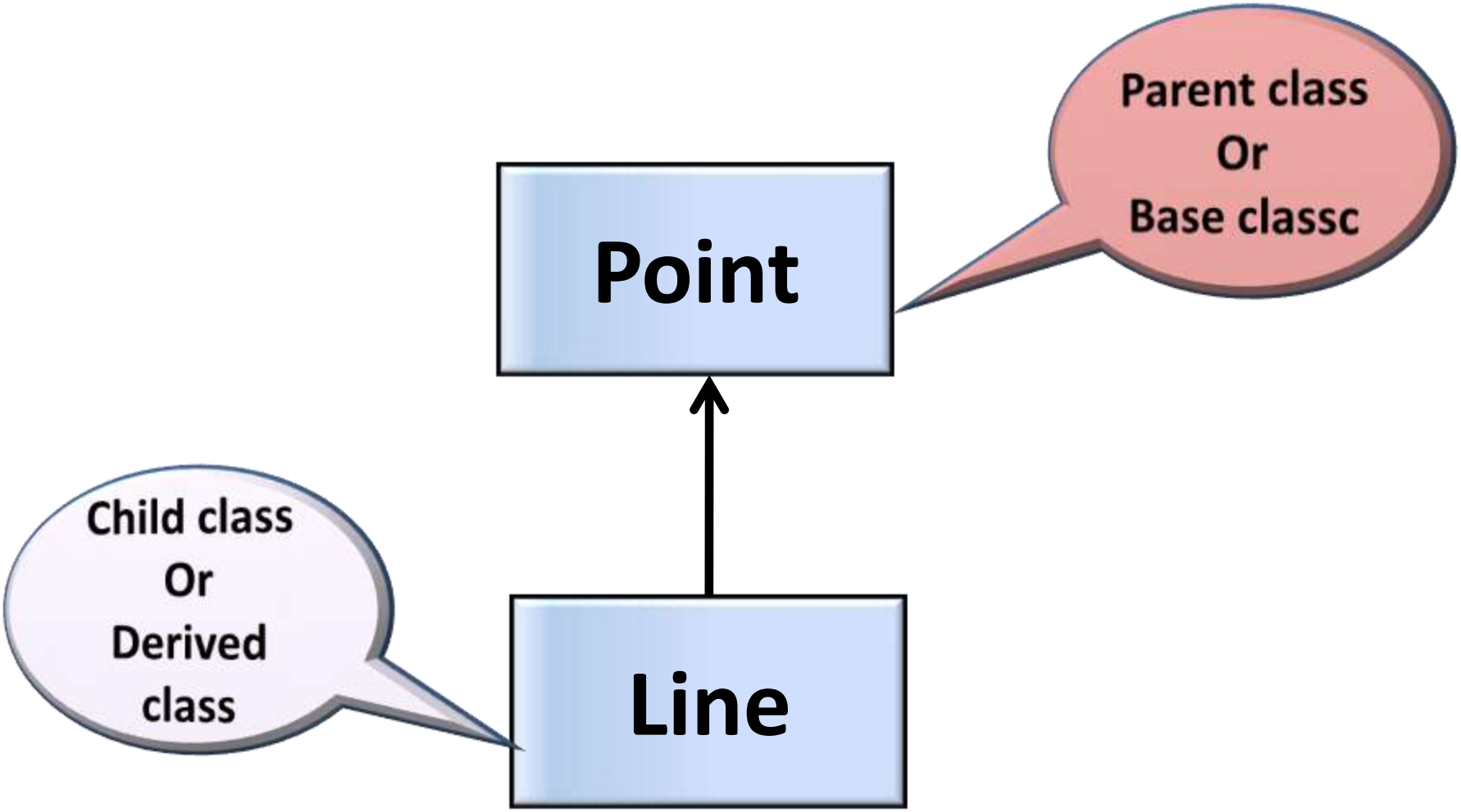
- Pgm to create a class RATIONAL with numerator and denominator as properties and perform following operations on rational numbers.
  - **r** = **r1** * **r2**; (by overloading * operator)
  - To check equality of **r1** and **r2** (by overloading == operator)

6. Inheritance
&
Virtual Functions

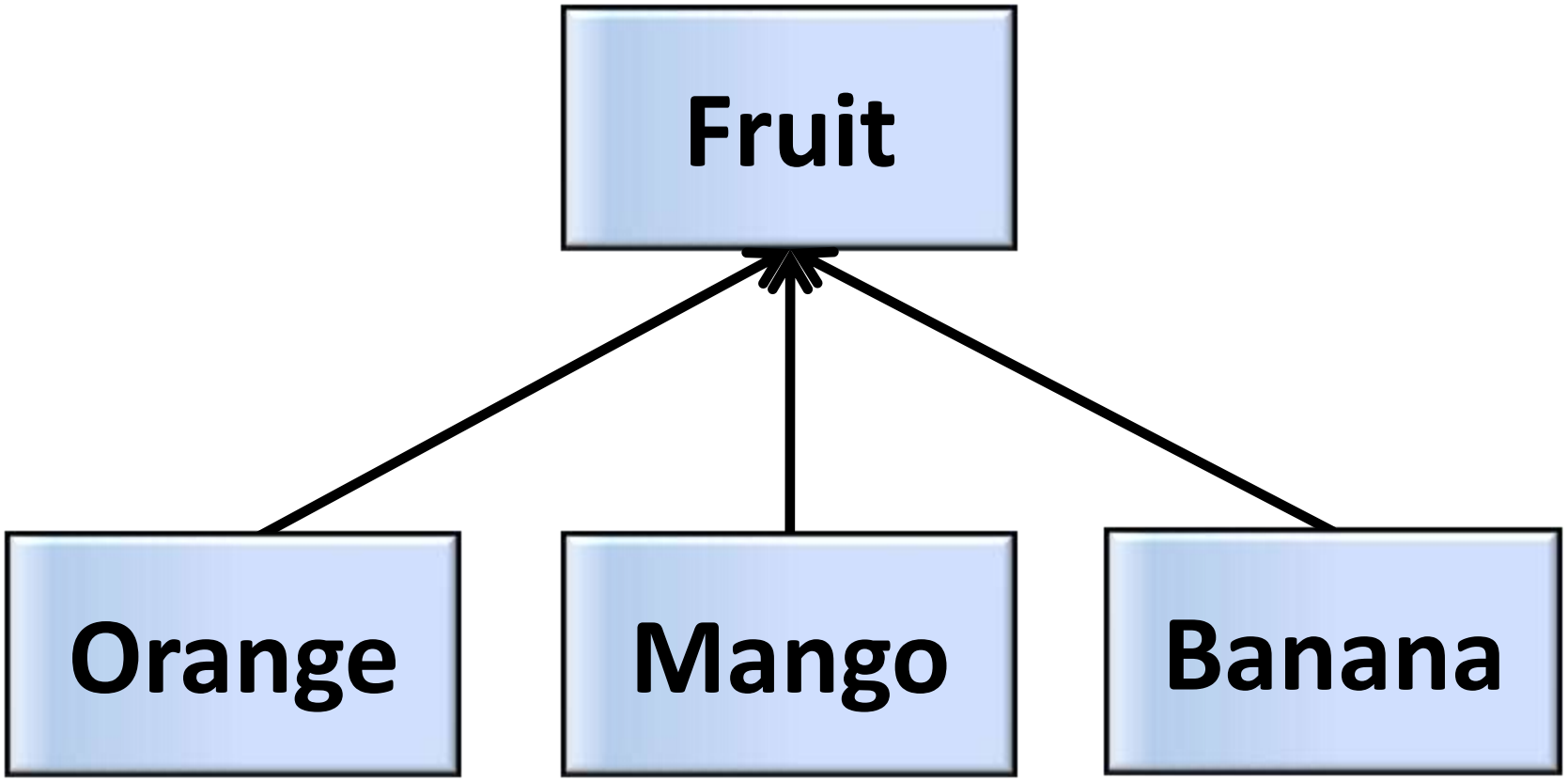
Inheritance

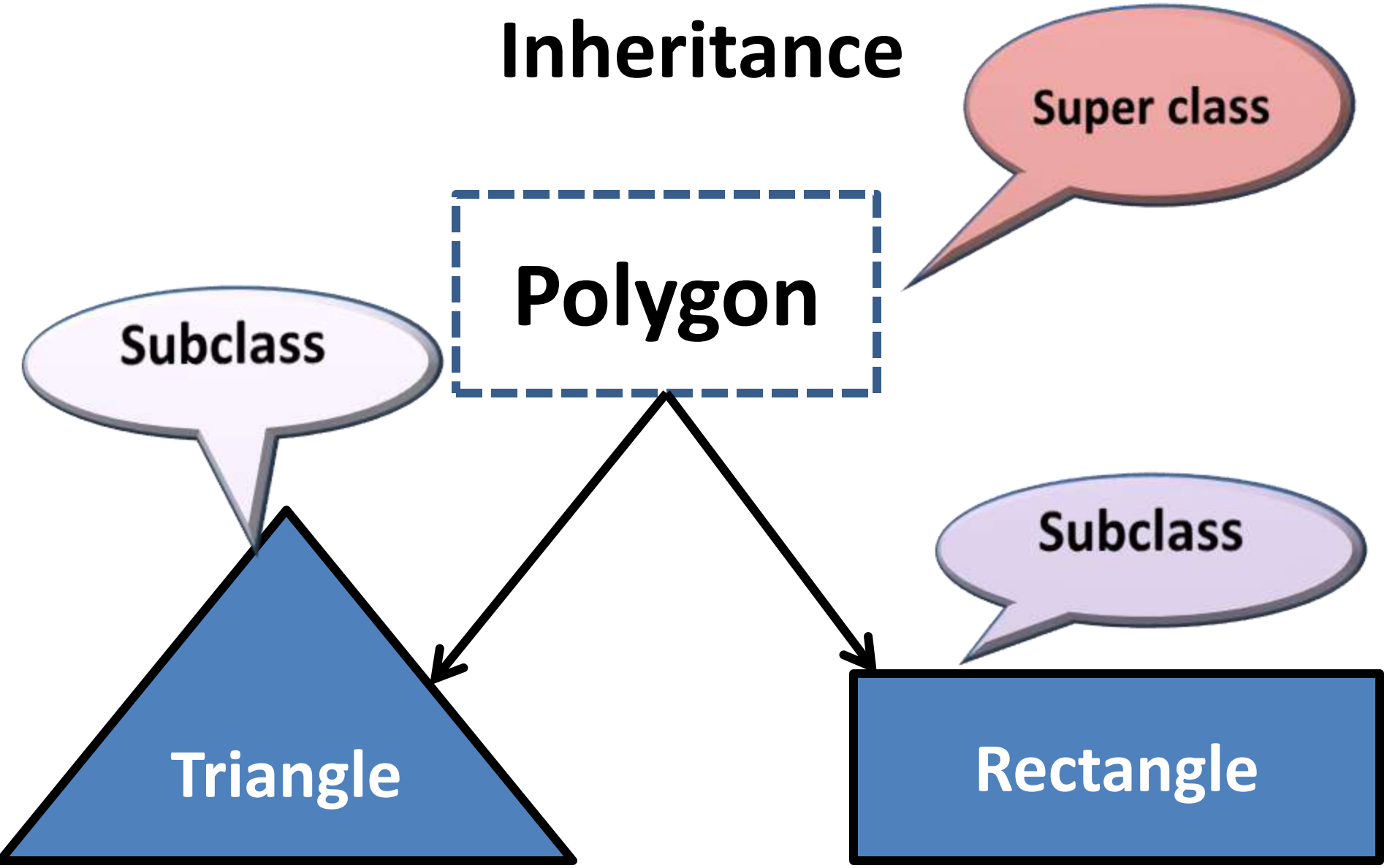
- “Inheritance is the mechanism to provides the power of **reusability** and **extendibility**.”
- “Inheritance is the process by which one **object can acquire the properties of another object**.”
- “Inheritance is the process by which **new** classes called ***derived*** **classes** are created from **existing** classes called ***base*** **classes**.”
- Allows the creation of **hierarchical** classifications.

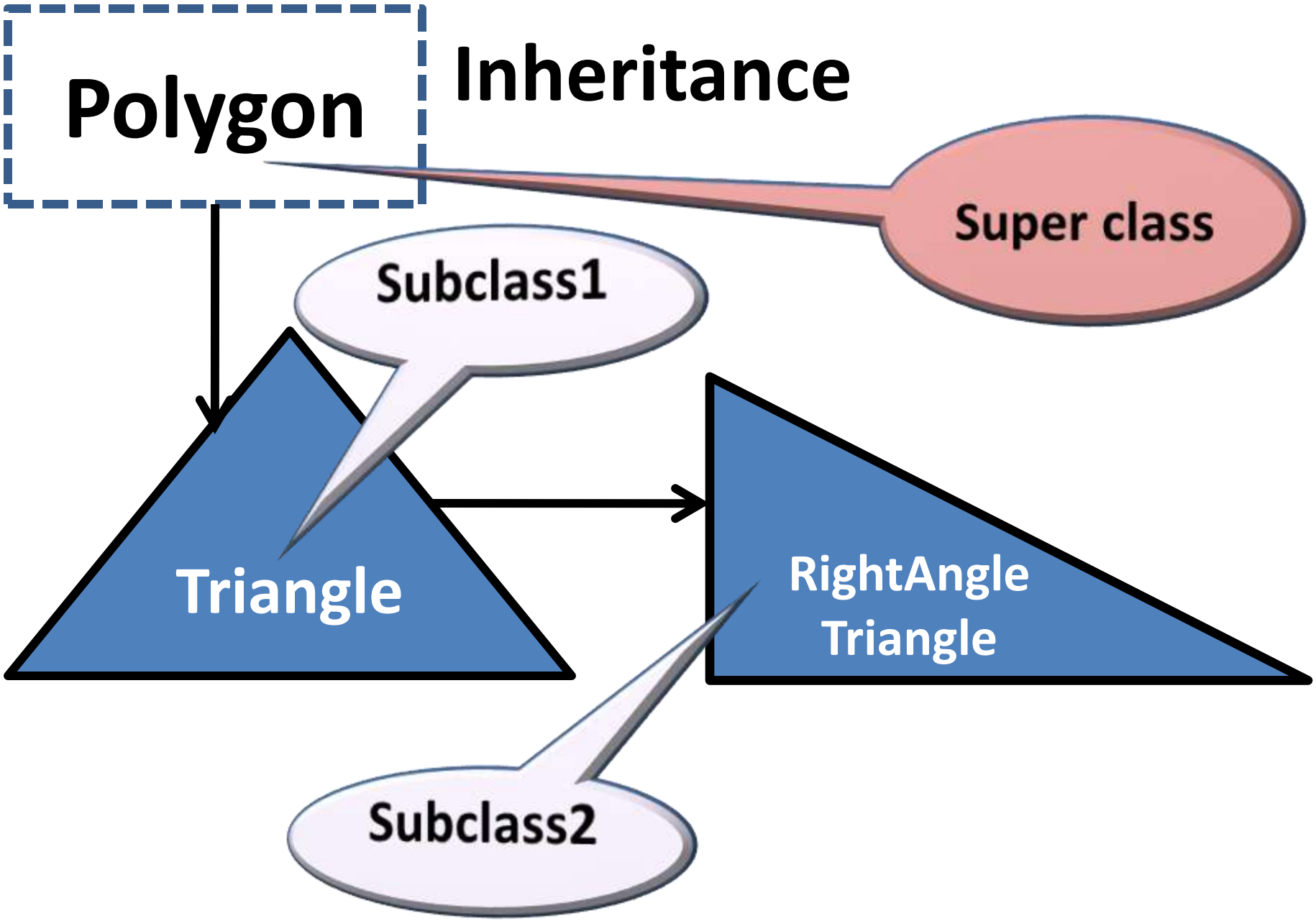
Inheritance



Inheritance




Inheritance
Super class
Polygon
Subclass
Subclass
Triangle
Rectangle


Polygon
Inheritance
Super class
Subclass1
Triangle
RightAngle
Triangle
Subclass2

Base Class

- **“Base class** is a class which defines those **qualities common to all objects** to be derived from the base.”

- The base class represents the most **general description.**

- A class that is **inherited** is referred to as a base class.

Derived Class

- “The classes derived from the base class are usually referred to as **derived classes**.”

- *“*A **derived class** includes **all** features of the generic base class and then adds **qualities specific** to the derived class.”

- The class that does the **inheriting** is called the derived class.

Inheritance

Note:

Derived class can be used as a **base class** for another derived class.

- In C++, inheritance is achieved by allowing one class to incorporate another class into its declaration.

Inheritance

- **Syntax:**

  class derived_class**: Acesss_specifier** base_class

  { };

- **Example:**

  class CRectangle**: public** Cpolygon{ };

  class CTriangle**: public** Cpolygon{ };

Inheritance & Access Specifier

Access	public	protected	private
Members of the same class	Yes	Yes	Yes
Members of derived classes	Yes	Yes	No
Non-members	Yes	No	No

Public base class Inheritance

- All **public** members of the base class become **public** members of the derived class.

- All **protected** members of the base class become **protected** members of the derived class.

Private base class Inheritance

- All **public** and **protected** members of the base class become **private** members of the derived class.

- But **private** members of the base class remain **private to base class** only, **not accessible** to the derived class.

Protected Members of Base Class

- Member is not accessible by other **non member** elements of the program.

- The base class' **protected members** become **protected members** of the derived class and are, therefore, accessible by the derived class.

Protected Base-Class Inheritance

- All **public** and **protected** members of the base class become **protected** members of the derived class.

- All **public** members of the base class become **unavailable** to **main()** function.

- All **private** members of the base class become **unavailable** to the derived class.

Inheritance & Access Specifier

Access	public	protected	private
Members of the same class	Yes	Yes	Yes
Members of derived classes	Yes	Yes	No
Non-members	Yes	No	No

Inheriting Multiple Base Classes

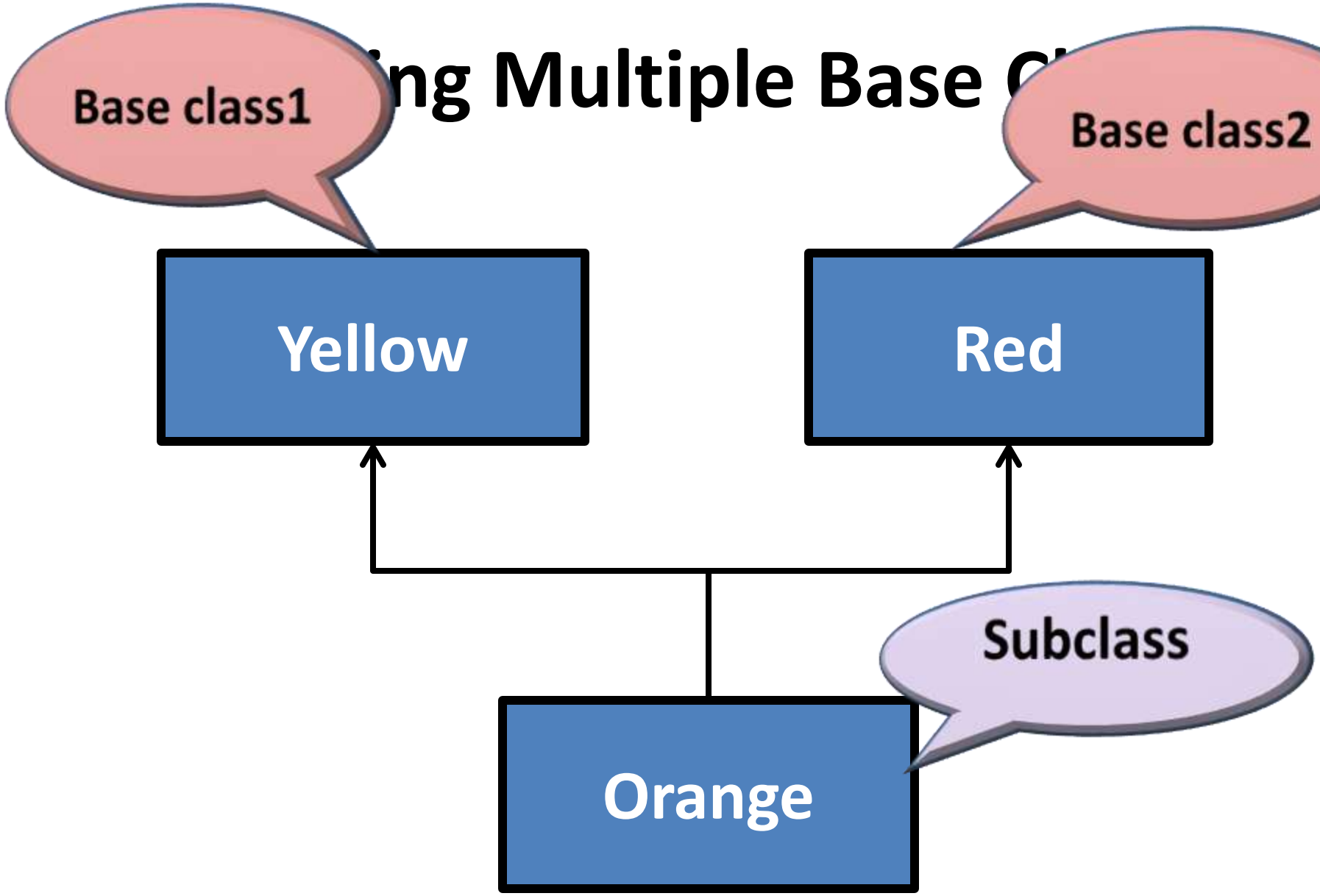
- **Syntax:**

  class derived: Acess_specifier base1,
  Acess_specifier base2
  { };

- **Example:**

  class Orange: Acess_specifier Yellow,
  Acess_specifier Red
  { };

ing Multiple Base C

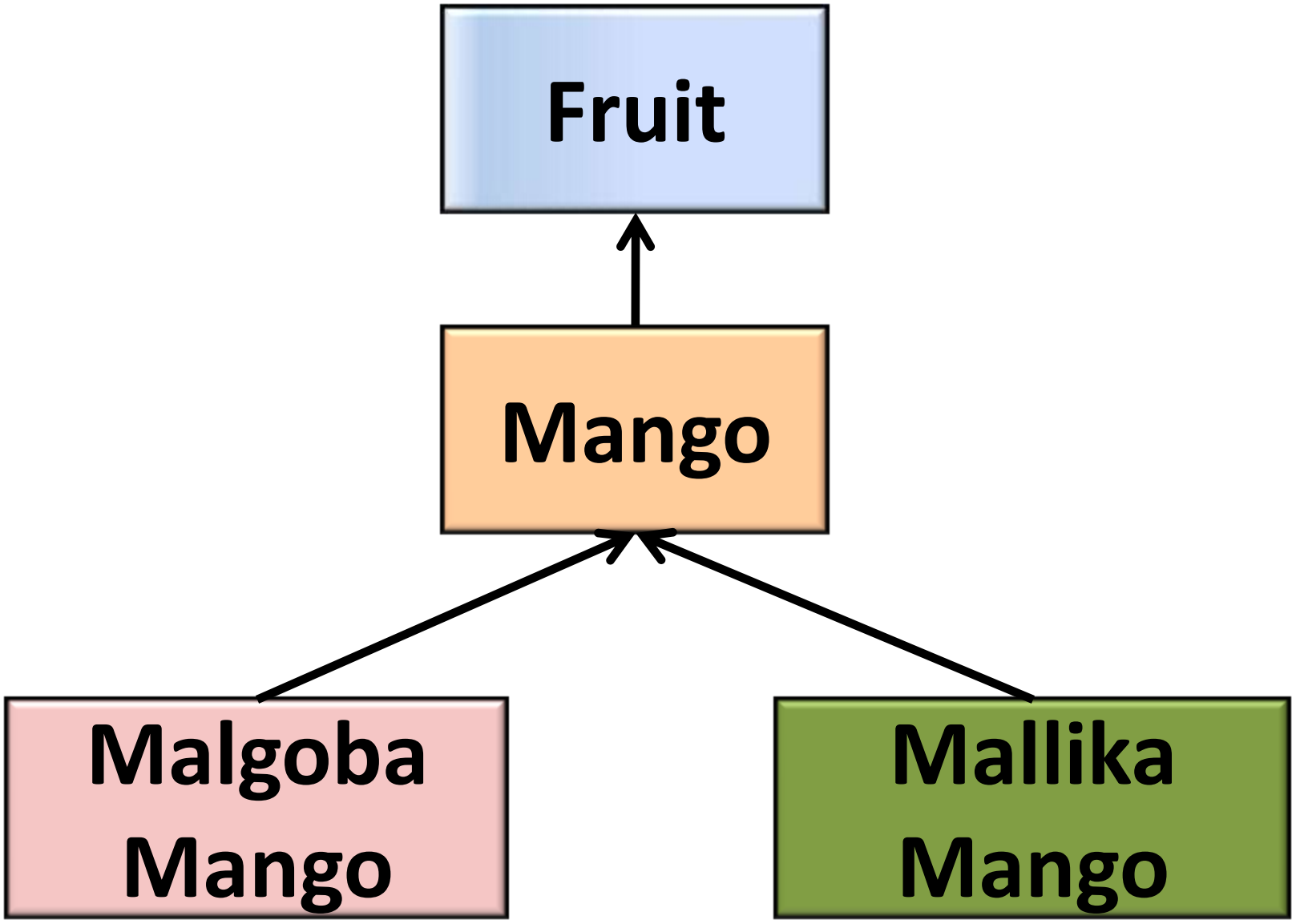


Constructors, Destructors & Inheritance

- Constructor functions are executed in their **order of derivation**.

- Destructor functions are executed in **reverse order of derivation**.

Inheritance



Constructors, Destructors & Inheritance

- When an object of a derived class is created,

  if the base class contains a constructor, it will be called first, followed by the derived class' constructor.

- When a derived object is destroyed, its destructor is called first, followed by the base class' destructor.

Passing Parameters to Base-Class Constructors

- Making use of an expanded form of the **derived class's constructor** declaration, we can pass arguments to one or more base-class constructors.

- **Syntax:**

```
derived-constructor(arg-list) : base1(arg-list),
                                base2(arg-list), … baseN(arg-list)
{   // body of derived constructor    }
```

Passing Parameters to Base-Class Constructors

- As we are **arguments** to a **base-class** constructor are passed via **arguments** of the **derived class**' constructor.

- Even if a **derived class**‘ constructor does **not** use any **arguments**, we need to **declare** a **constructor** as if the base class requires it.

- The **arguments passed** to the **derived class** are simply passed along to the **base class constructor**.

Granting Access

- When a base class is inherited as **private:**

  - all **public** and **protected** members of that class become **private members** of the derived class.

- But in some certain circumstances, we want to **restore** one or more inherited members to their **original access** specification.

Granting Access

- To accomplish this :

using

access declaration

Granting Access

- **using** statement**:**

  is designed primarily to support namespaces.

- **Access declaration:**

  restores an inherited member's access specification

  Syntax:

  base_class_name::member;

Granting Access

- Access declaration is done under the appropriate **access heading** in the **derived class'** declaration.

Note:

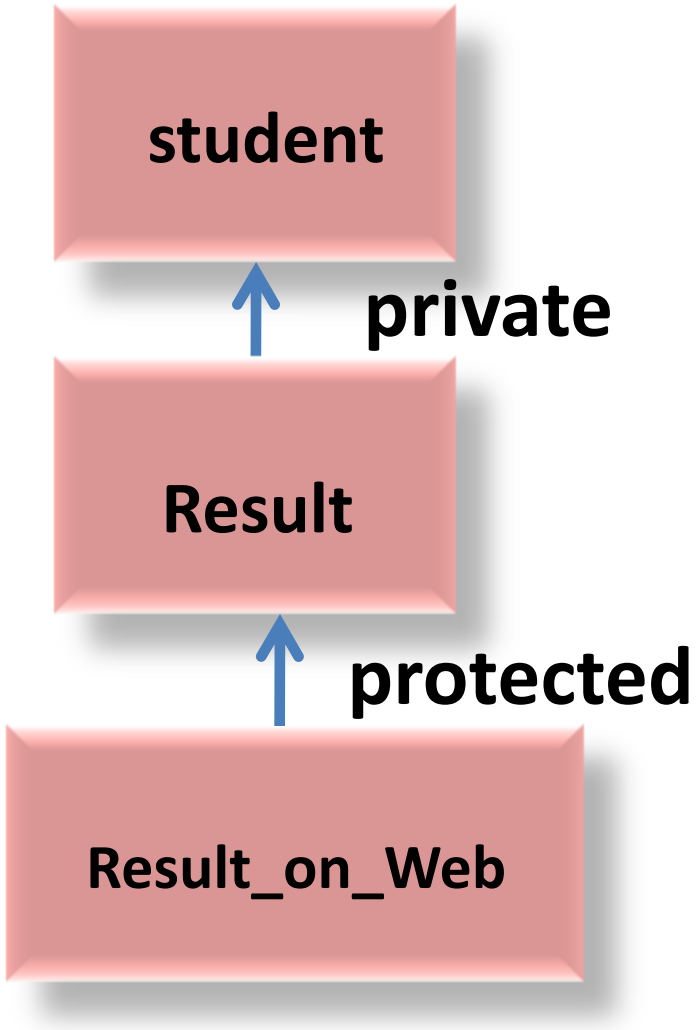
No type declaration is required.

Granting Access

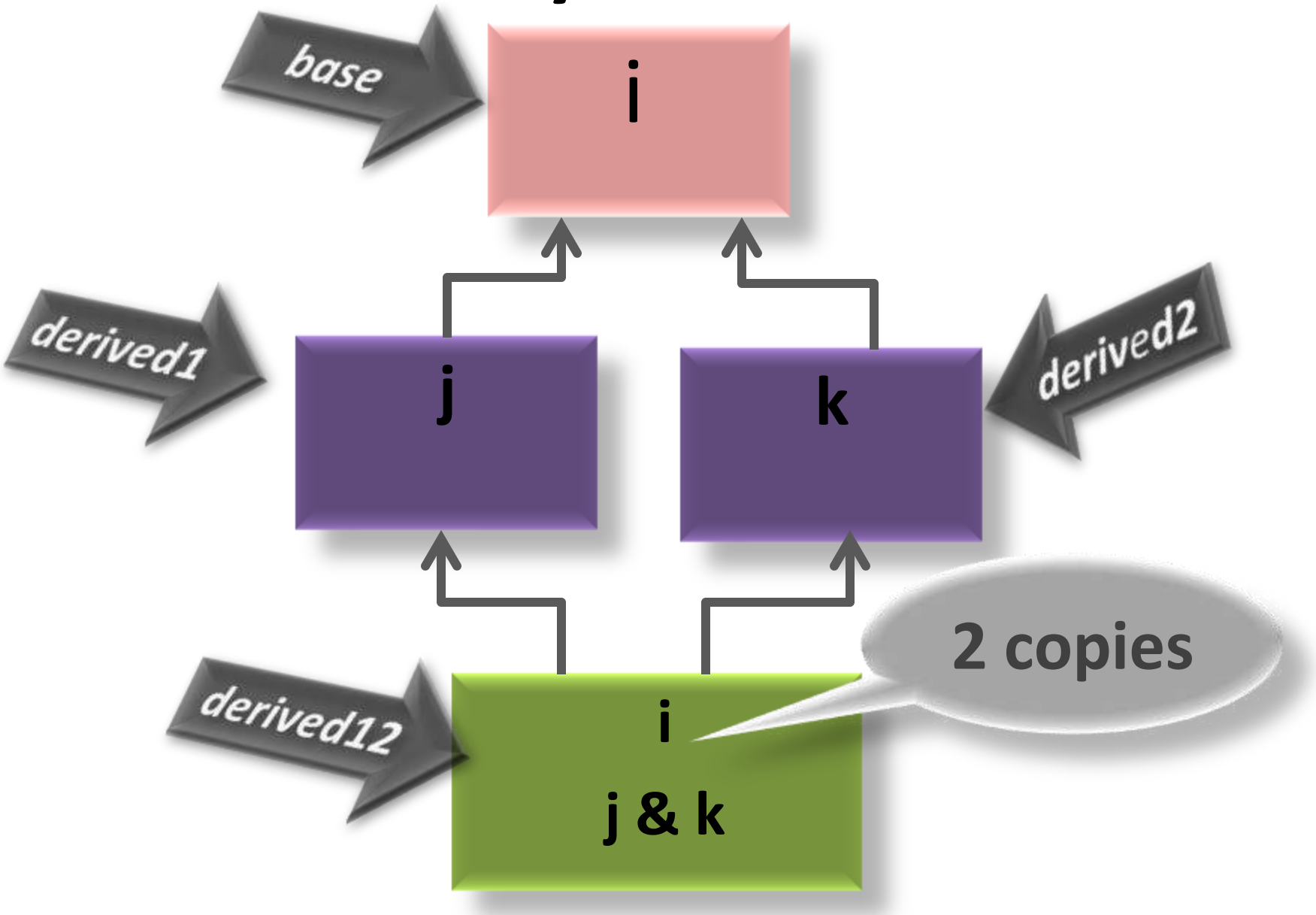

```
class base {
public:
int j;
};

class derived: private base {
public:
// here is access declaration
base::j;
};
```

Program for Implementation

Program to implement the given hierarchy assuming proper properties for **student** class. The **Result** class computes the result of every student. The **Result_on_Web** class displays the result upon getting the **grant** for **USN**.



Hierarchy of Classes



Hierarchy of Classes

Remedy…..?

Virtual Base Classes

- Used to **prevent** multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual when it is inherited.**

- **Syntax:**

  class derived : **virtual** public base

  { . . . };

Virtual Functions

- “A virtual function is a member function that is declared **within** a **base class** and **redefined** by a **derived class**.”

- Virtual functions implements the **"one interface, multiple methods"** philosophy under polymorphism.

Virtual Functions

- The virtual function within the base class defines the form of the **interface** to that function.

- Each **redefinition** of the virtual function by a derived class implements **its operation** as it relates specifically to the derived class. That is, the redefinition creates a **specific** method.

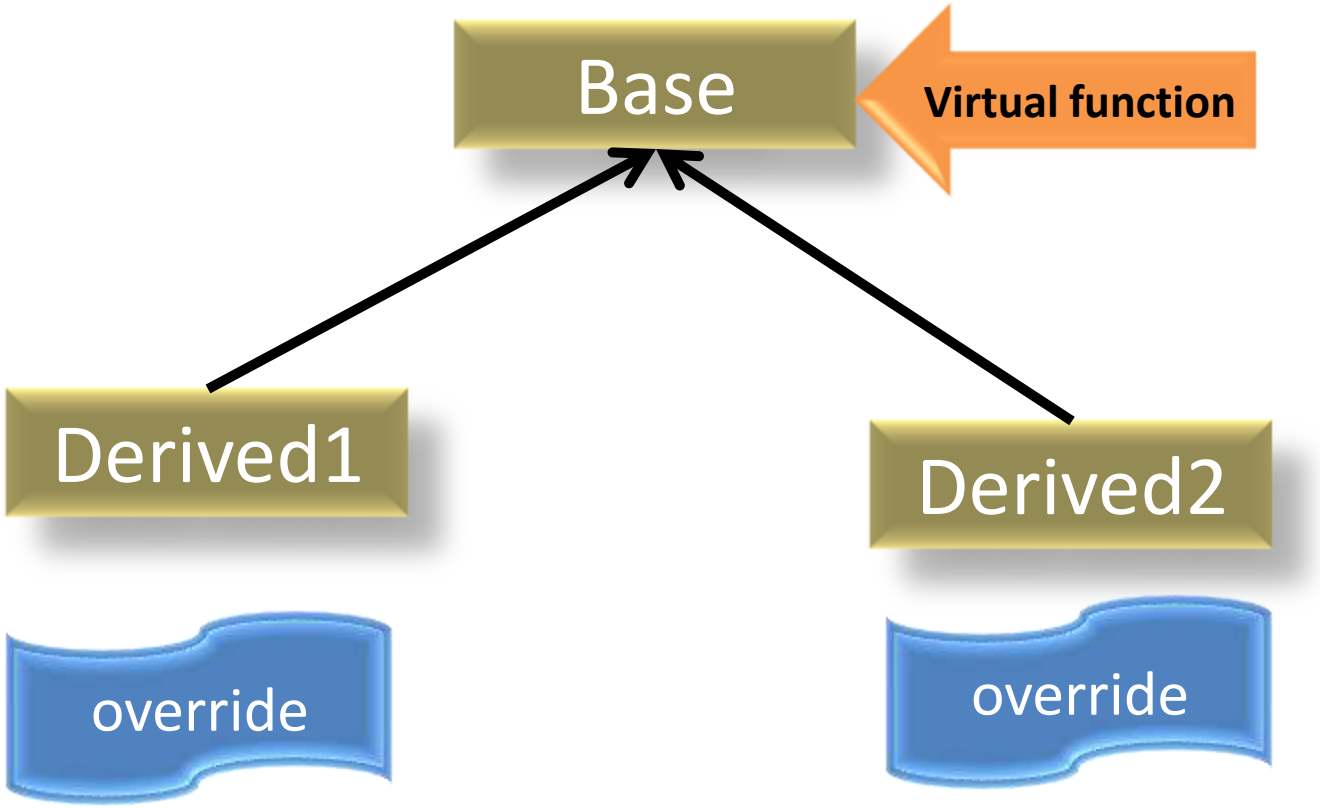
Virtual Functions

- To create a virtual function, precede the function‘s declaration in the base class with the keyword **virtual.**

- **Example:**

```
class base {
    public:
        virtual void member_func(){   }
};
```

Virtual Functions



Virtual Functions

- When accessed **"normally"** virtual functions behave just like any other type of class member function.

- But virtual functions' importance and capacity lies in supporting the **run-time polymorphism** when they accessed via a pointer.

Virtual Functions

- **How to implement run-time polymorphism?**
  - create base-class pointer can be used to point to an object of any class derived from that base
  - initialize derived object(s) to base class object.

- Based upon which derived class objects' assignment to the base class pointer, c++ determines which version of the virtual function to be called. And this determination is made at run time.

Virtual Functions

- The **redefinition of a virtual function** by a derived class appears similar to **function overloading**?
- No

- The prototype for a redefined virtual function must match exactly the prototype specified in the base class.

Virtual Functions

Restrictions:

- All aspects of its prototype must be the **same** as base class virtual function.
- Virtual functions are of **non-static** members.
- Virtual functions can not be **friends**.
- **Constructor** functions can**not** be **virtual**.
- But **destructor** functions **can** be **virtual**.

NOTE:

Function overriding is used to describe virtual function redefinition by a derived class.

Destructor functions can be virtual?

- Yes.
- In large projects, the destructor of the derived class was not called at all.
- This is where the virtual mechanism comes into our rescue. By making the Base class Destructor virtual, both the destructors will be called in order.

Function overriding

"A **function overriding** is a process in which a member function that is declared **within** a **base class** and **redefined** by a **derived class** to implement the **"one interface, multiple methods"** philosophy under polymorphism."

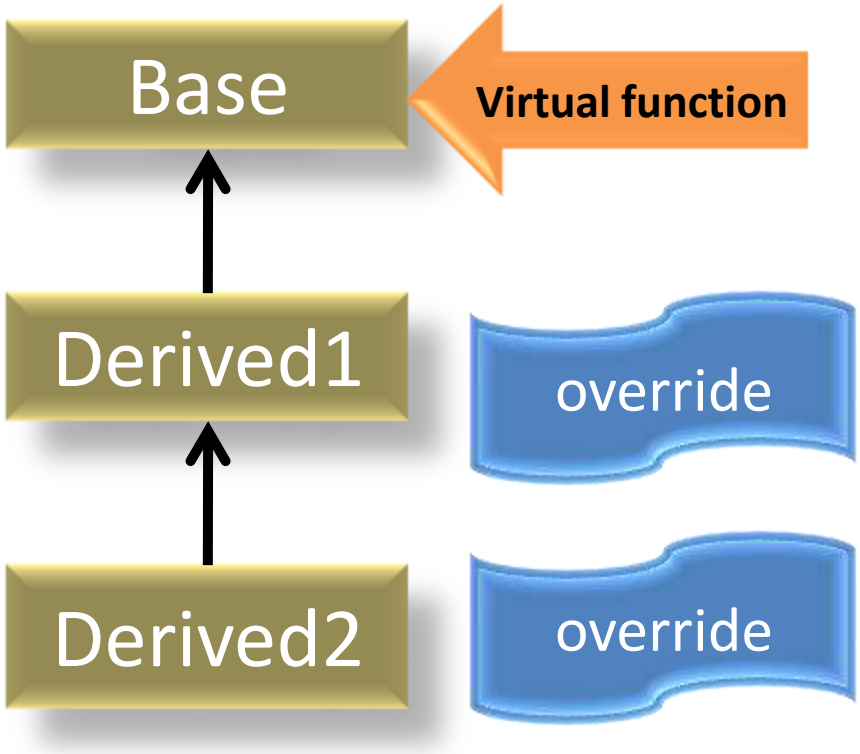
Calling a Virtual Function Through a Base Class Reference

- Since **reference is an implicit pointer**, it can be used to access virtual function.

- When a virtual function is called through a **base-class reference**, the version of the function executed is determined by the **object** being **referred** to at the time of the call.

The Virtual Attribute Is Inherited

- When a virtual function is inherited, its virtual nature is also inherited.

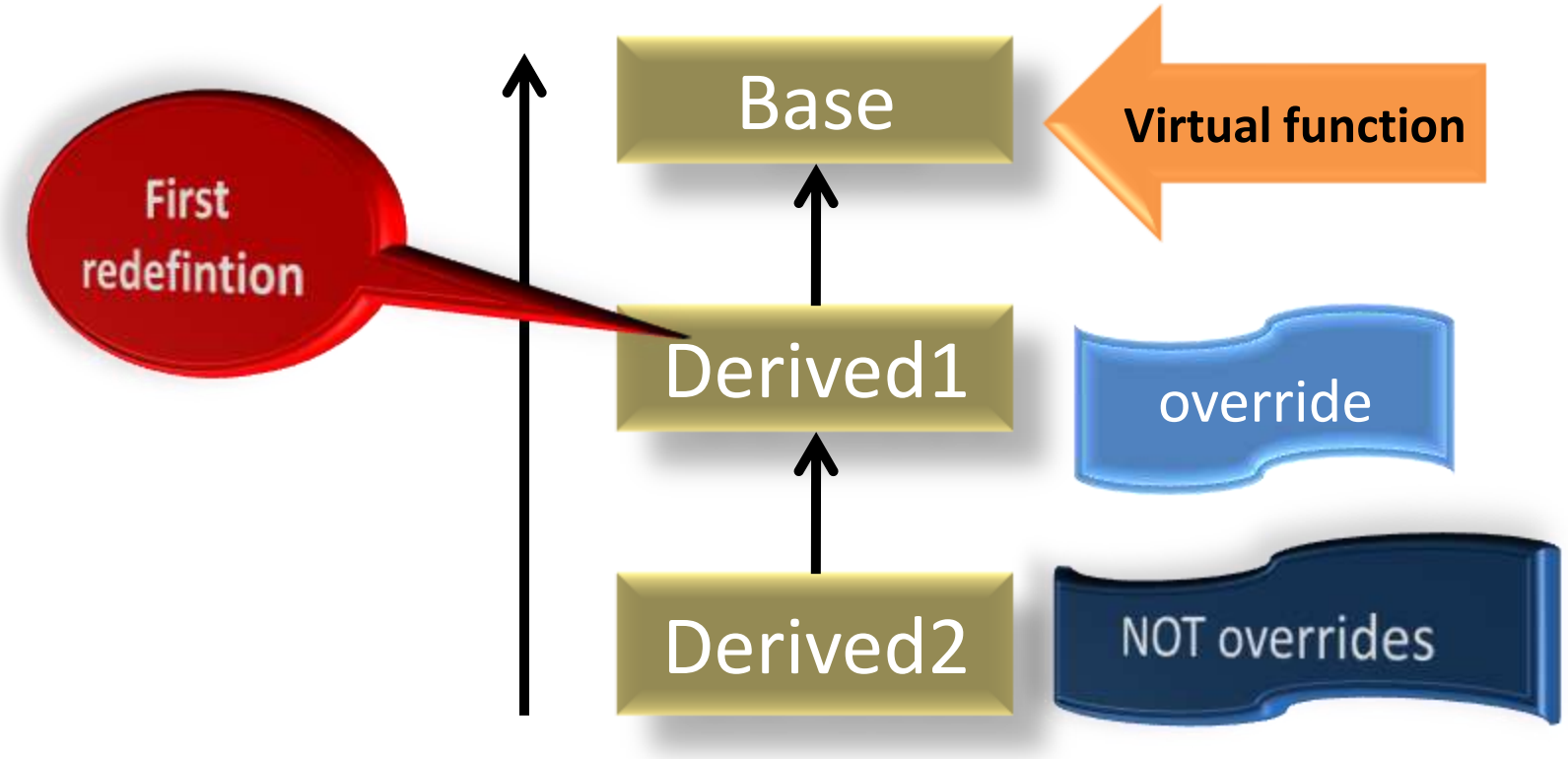


Virtual Functions Are Hierarchical

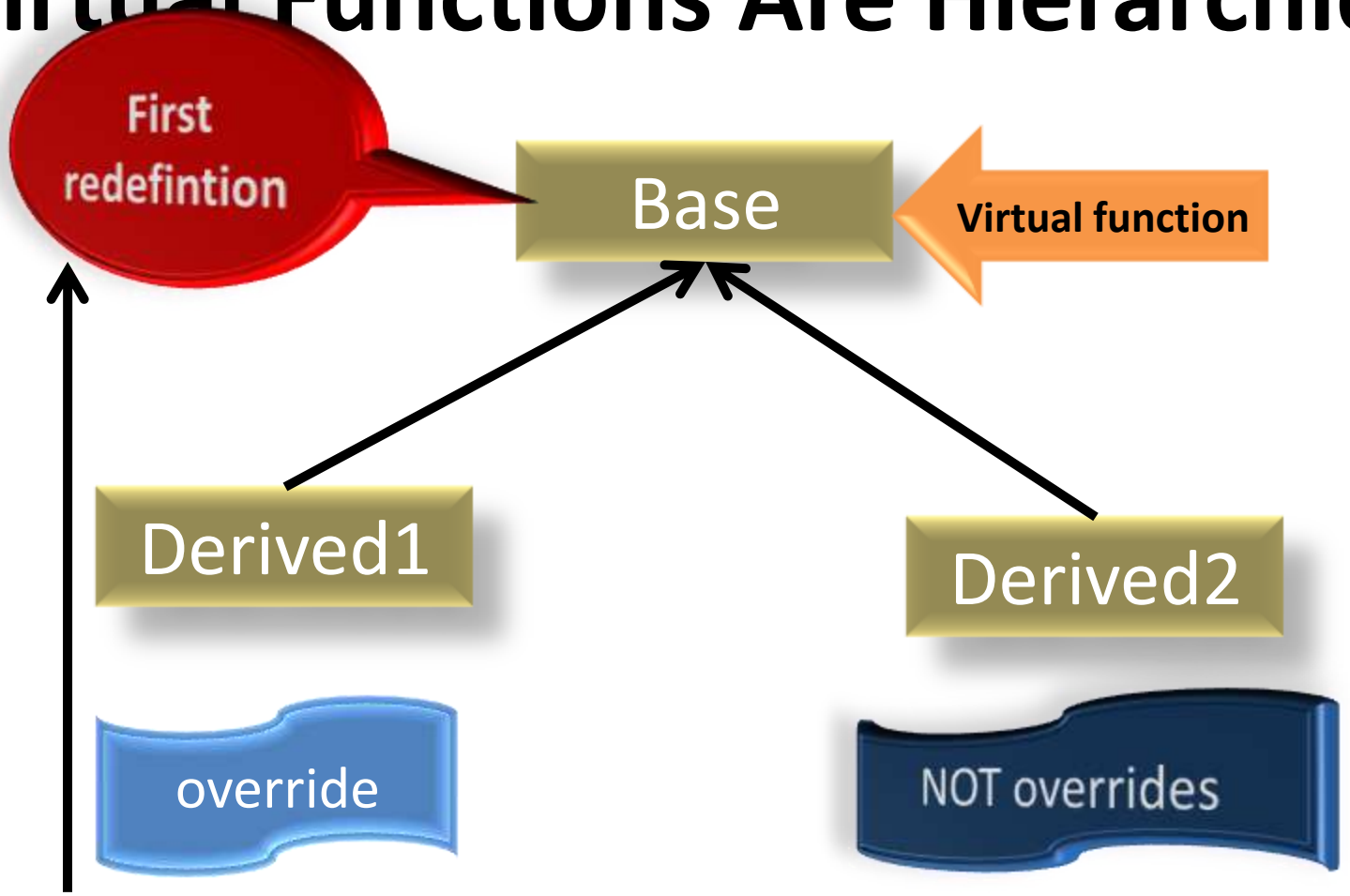
- Virtual functions are also **hierarchical in nature**.

- This means that when a **derived class fails** to override a virtual function, then **first redefinition** found in **reverse** order of derivation is used.

Virtual Functions Are Hierarchical



Virtual Functions Are Hierarchical



Pure Virtual Functions

- “A pure virtual function is a virtual function that has **no definition** within the base class.”

- To declare a pure virtual function:

  Syntax:

  virtual rtype func-name(parameter-list) = 0;

Pure Virtual Functions

- When a **virtual function** is made **pure**, any **derived** class must **provide** its **definition**.

- If the **derived** class **fails** to override the pure virtual function, a **compile-time error** will result.

NOTE:
When a virtual function is declared as pure, then all derived classes must override it.

Abstract Classes

- “A class that contains **at least** one **pure virtual** function then it is said to be abstract class.”

- **No objects** of an abstract class be **created**.

- Abstract class **constitutes** an **incomplete type** that is **used** as a **foundation** for **derived**. **classes**.

Using Virtual Functions

- We can achieve the most powerful and flexible ways to implement the **"one interface, multiple methods“**.

- We can create a class hierarchy that moves from **general to specific** (base to derived).

Using Virtual Functions

- We can define all **common** features and interfaces in a base class.
- **Specific** actions can be implemented only by the derived class.
- We can **add new** case easily.

Early vs. Late Binding

- “Early binding refers to events that occur at **compile time**.”
- Early binding occurs when all information needed to call a function is known at compile time.
- **Examples :**

  function calls ,overloaded function calls, and overloaded operators.

Early vs. Late Binding

- “Late binding refers to function calls that are **not resolved until run time**.”
- Late binding can make for somewhat **slower** execution times.
- **Example:**

  virtual functions

7. IO Stream Library

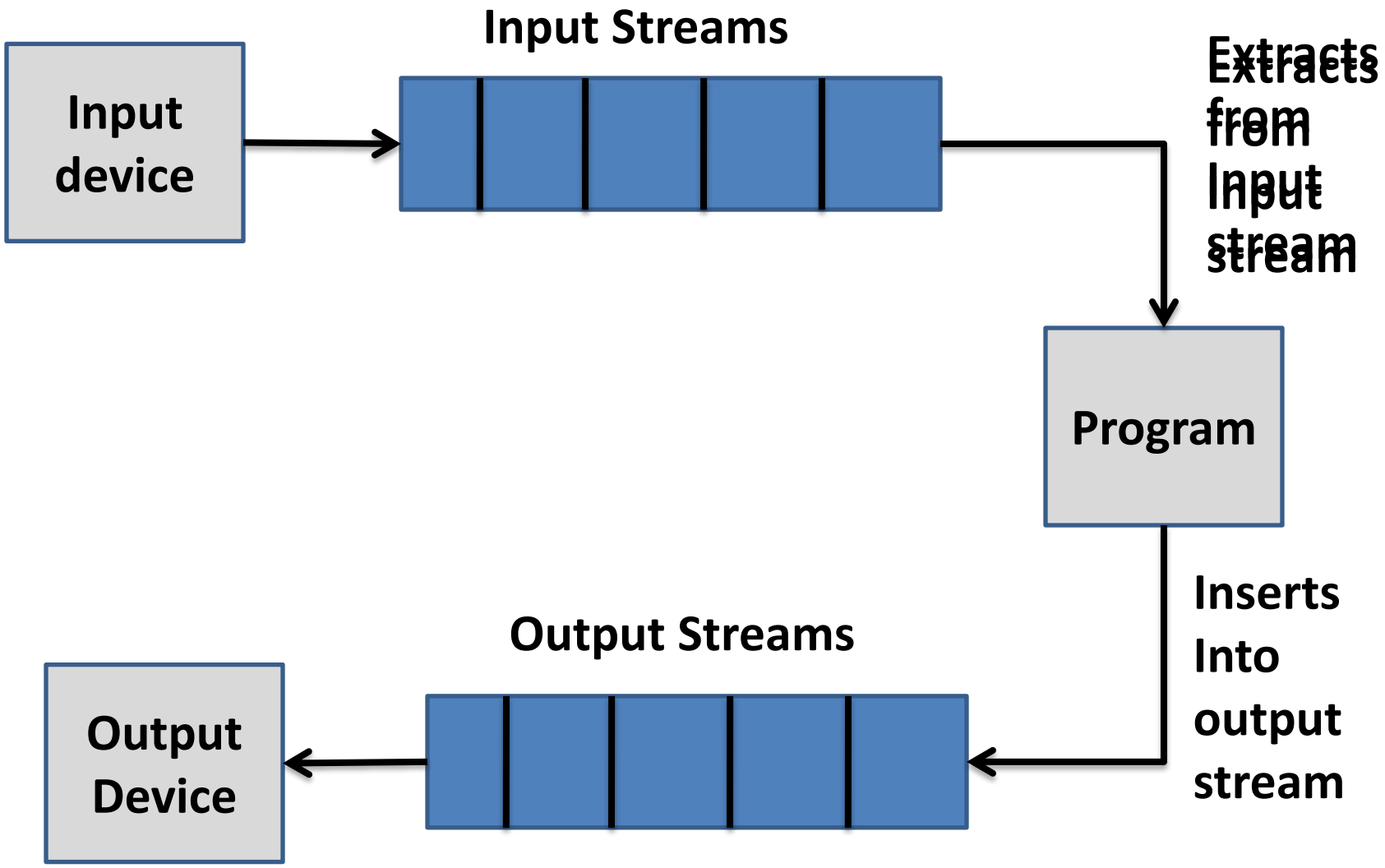
Introduction

- 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.***

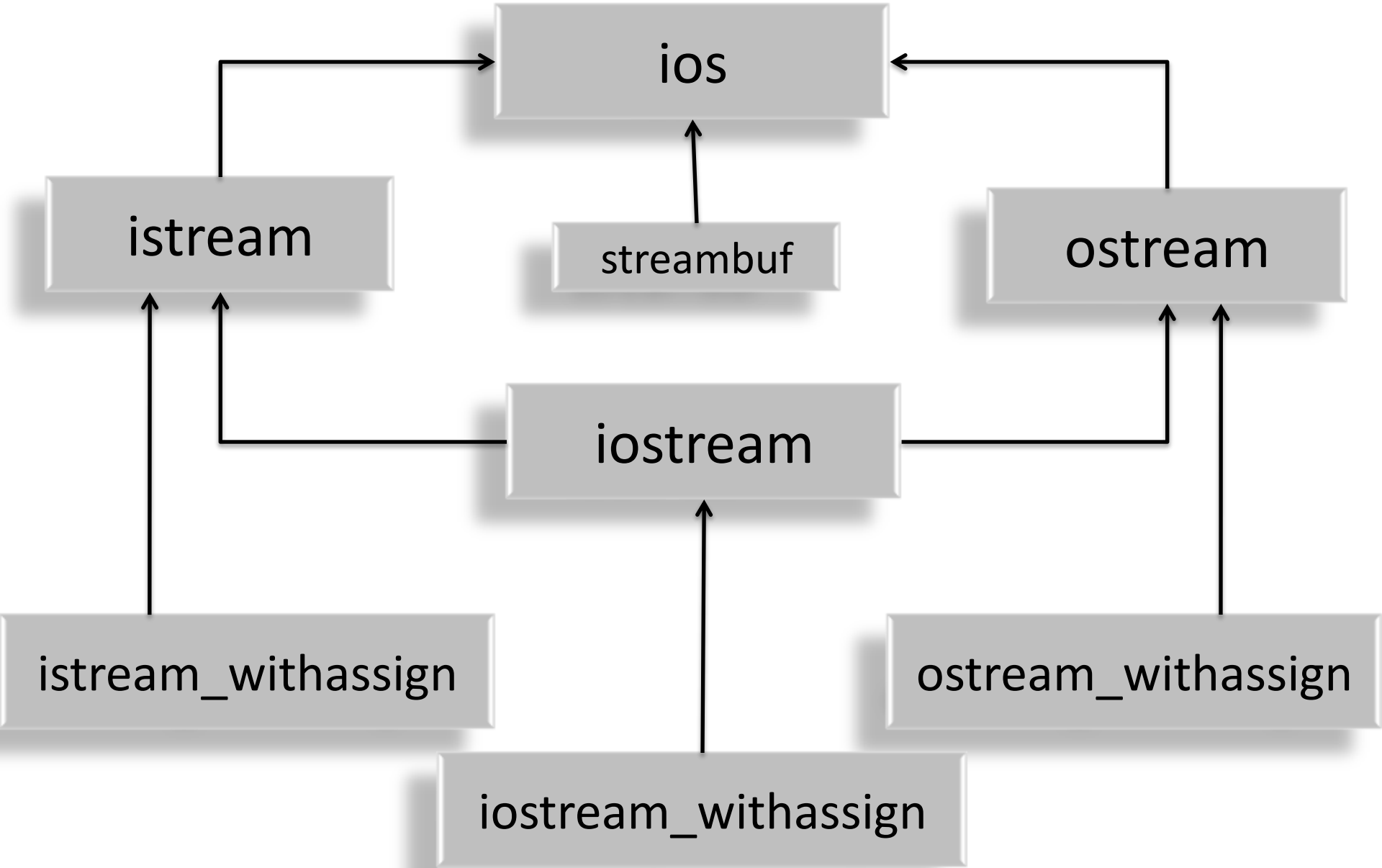
Introduction

- 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


Input Streams
Input device
Extracts from Input stream
Program
Inserts Into output stream
Output Streams
Output Device

I/O Stream Classes for console Operations



C++'s Predefined Streams

When a C++ program begins execution, four 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()

Overloading >> operator

- **Prototype:**

  friend istream& operator >>(istream&, Matrix&);

- **Example:**

```
istream& operator >> (istream& in, Matrix& m)
{
    for (int i=0; i<row*col; i++)
    {
        in >> m[i];
    }
    return in;
}
```

```
void main()
{
  :
Cin>>mobj;
  :
}
```

Overloading << operator

- **Prototype:**

friend ostream& operator <<(ostream&, Matrix&);

- **Example:**

```
ostream& operator << (ostream& out, Matrix& m)
{
    for (int i=0; i<row; i++)
    {
        for (int j=0; j<col; j++)
        {   out<< m.a[i][j] << "\t";
        }
        out << endl;
    }
```

```
void main()
{
    :
cout<<mobj;
    :
}
```

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

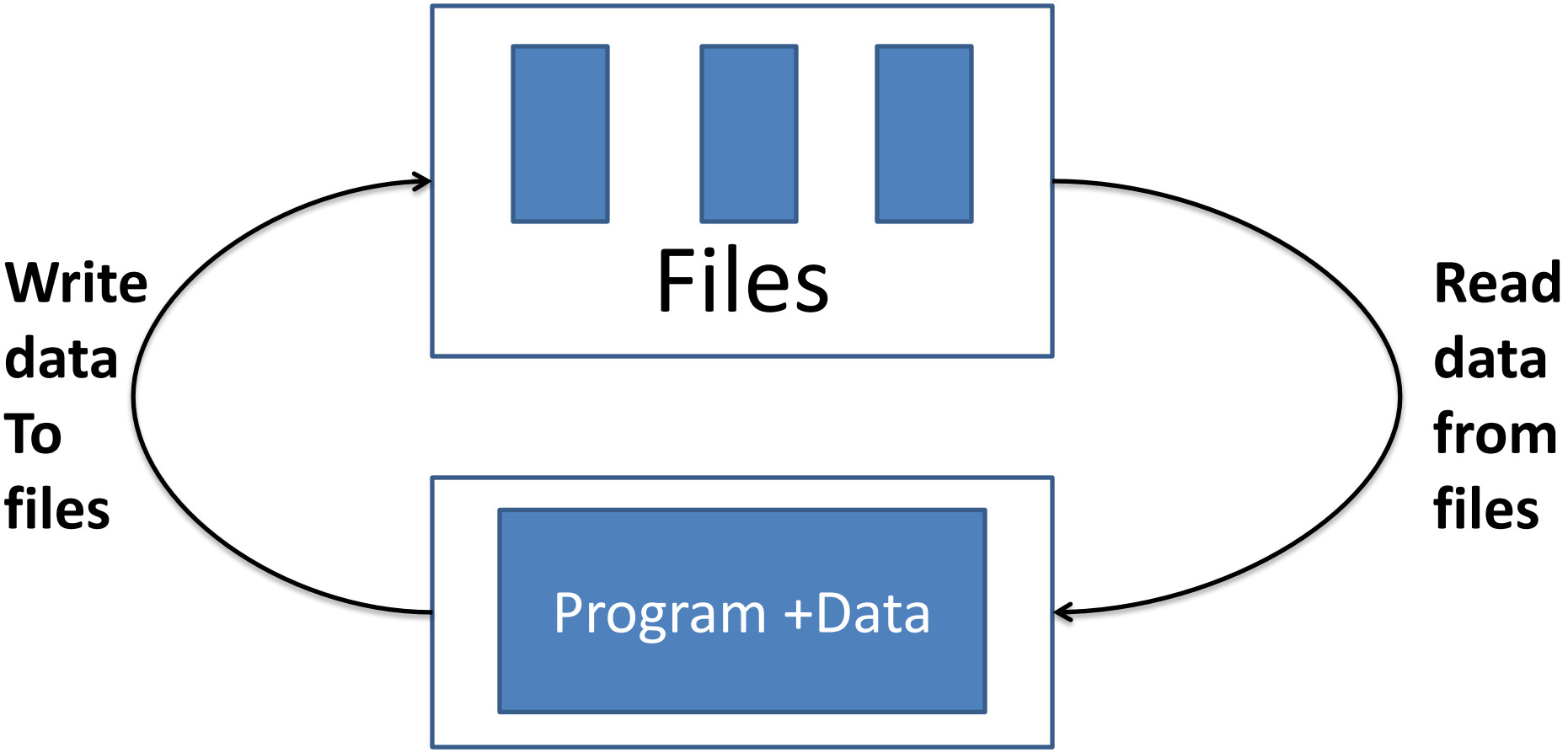
Using Manipulators to Format I/O

Manipulators	Meaning
boolalpha	Turns on **boolapha 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.**

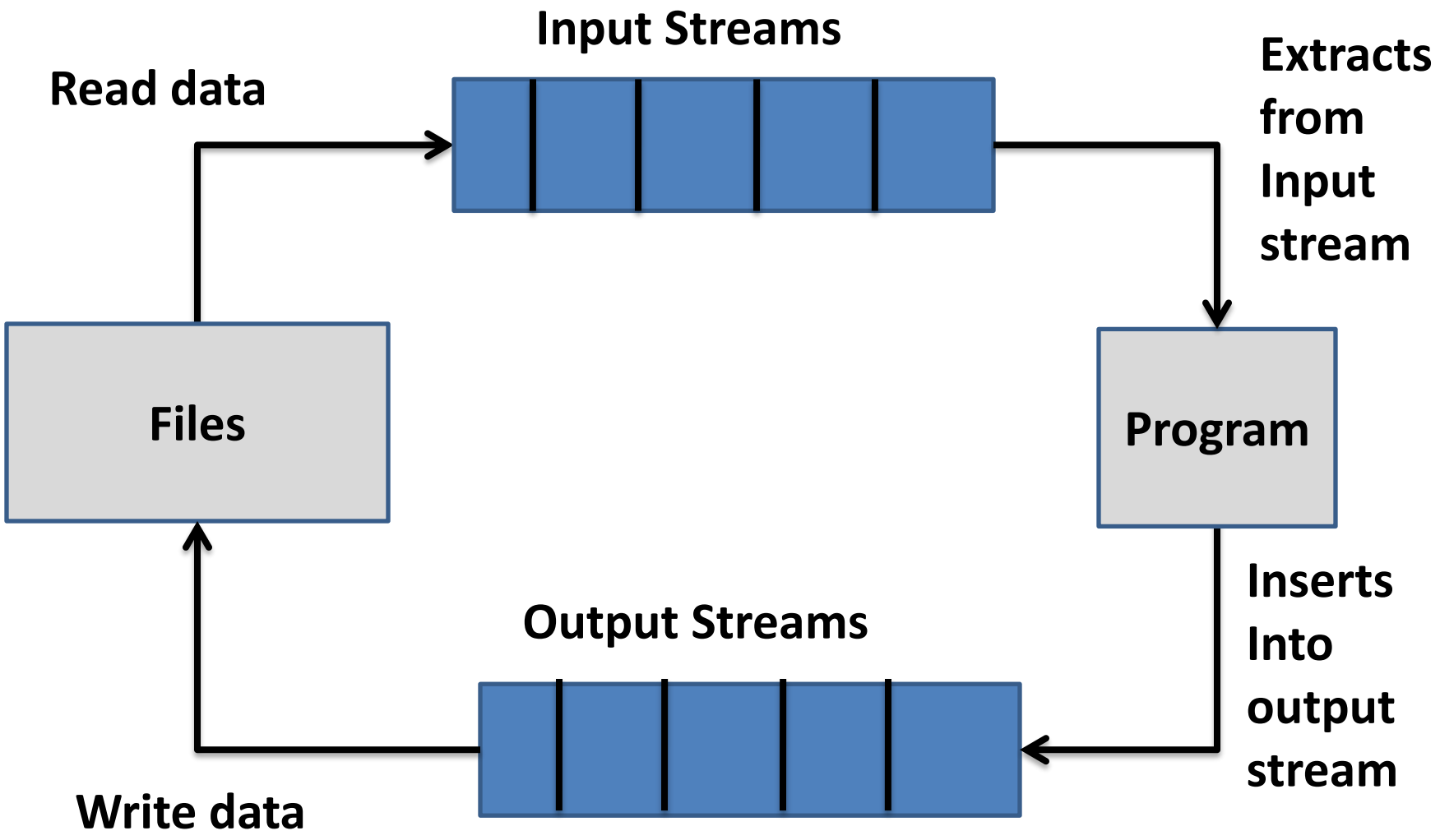
Manipulators	Meaning
noshowbase	Turns off **showbase flag.**
noshowpoint	Turns off **showpoint flag.**
no showpos	Turns off **showpos flag.**
noskipws	Turns off **skipws flag.**
nounitbuf	Turns off **unitbuf flag.**
nouppercase	Turns off **uppercase flag.**
oct	Turns on **oct flag.**
right	Turns on **right flag.**
scientfic	Turns on **scientific flag.**
setbase(int base)	**Set the number base to *base.***

Manipulators	Meaning
setfill(int ch)	Set the fill character to *ch.*
setiosflags(fmtflags f)	Turn on the flags specified in *f.*
setprecision(int p)	Set the number of digits of precision.
setw(int w)	Set the field width to *w.*
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.**

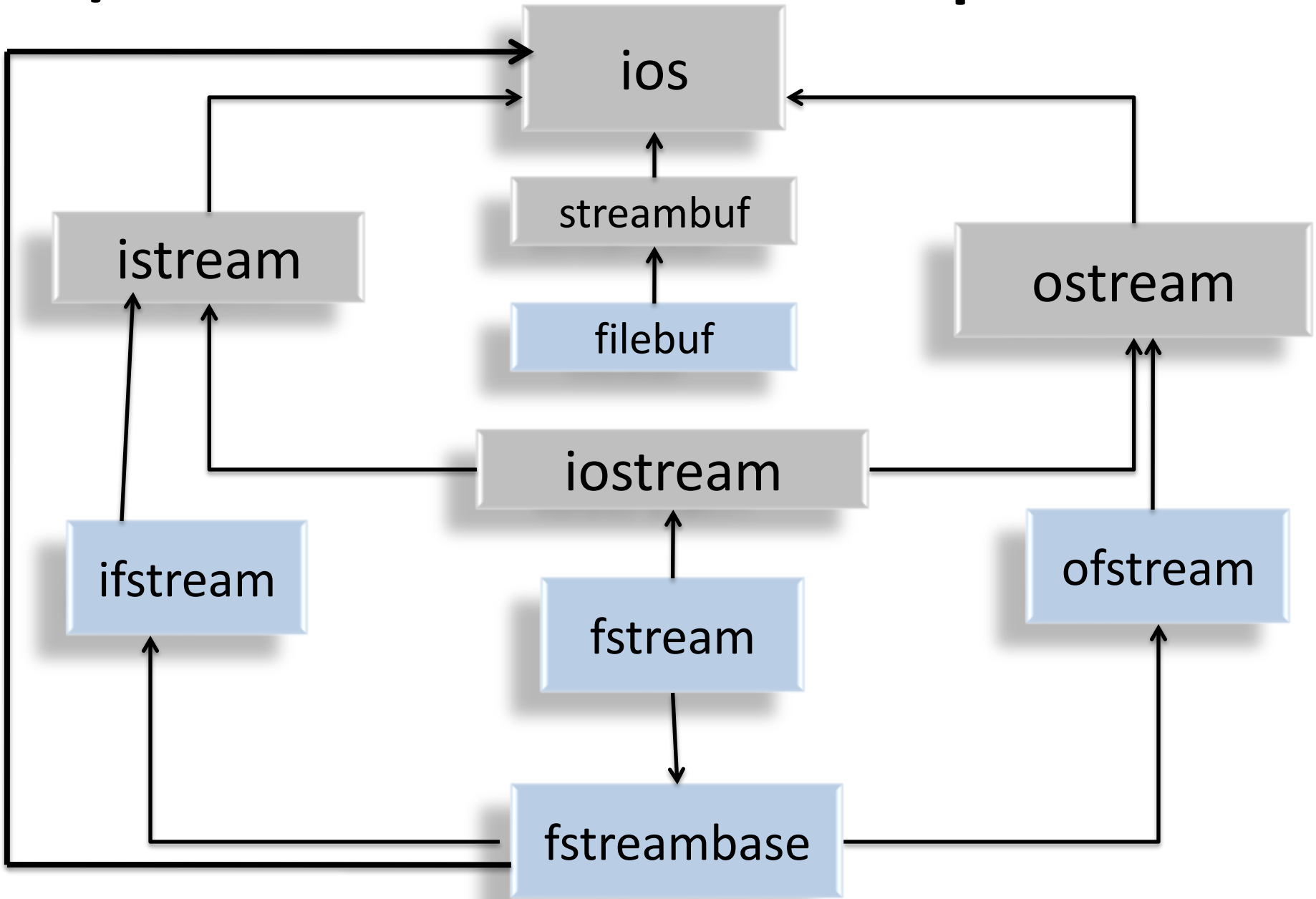
File I/O Operations



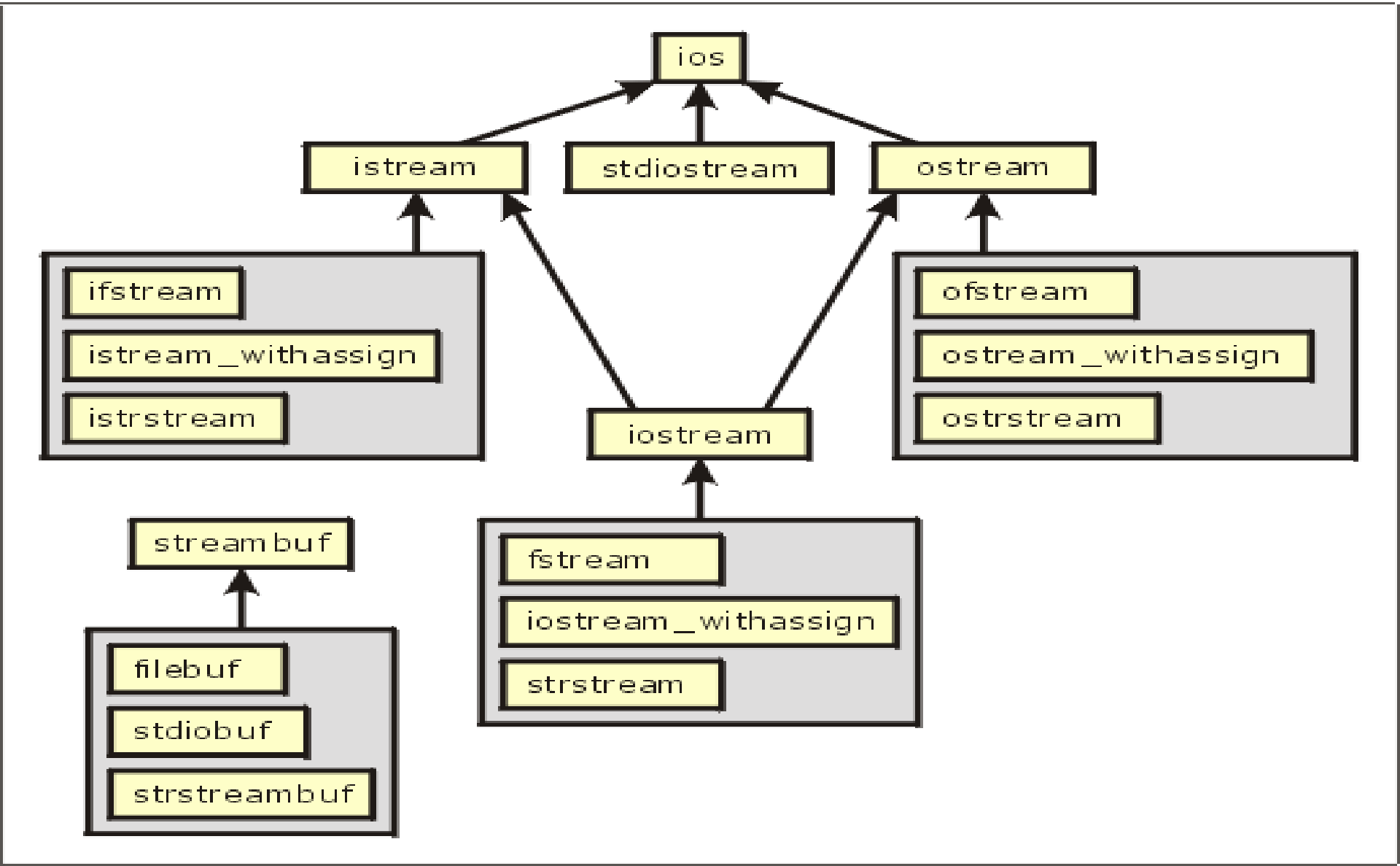
File Input & Output streams



I/O Stream classes for File operations



I/O Stream Class Hierarchy

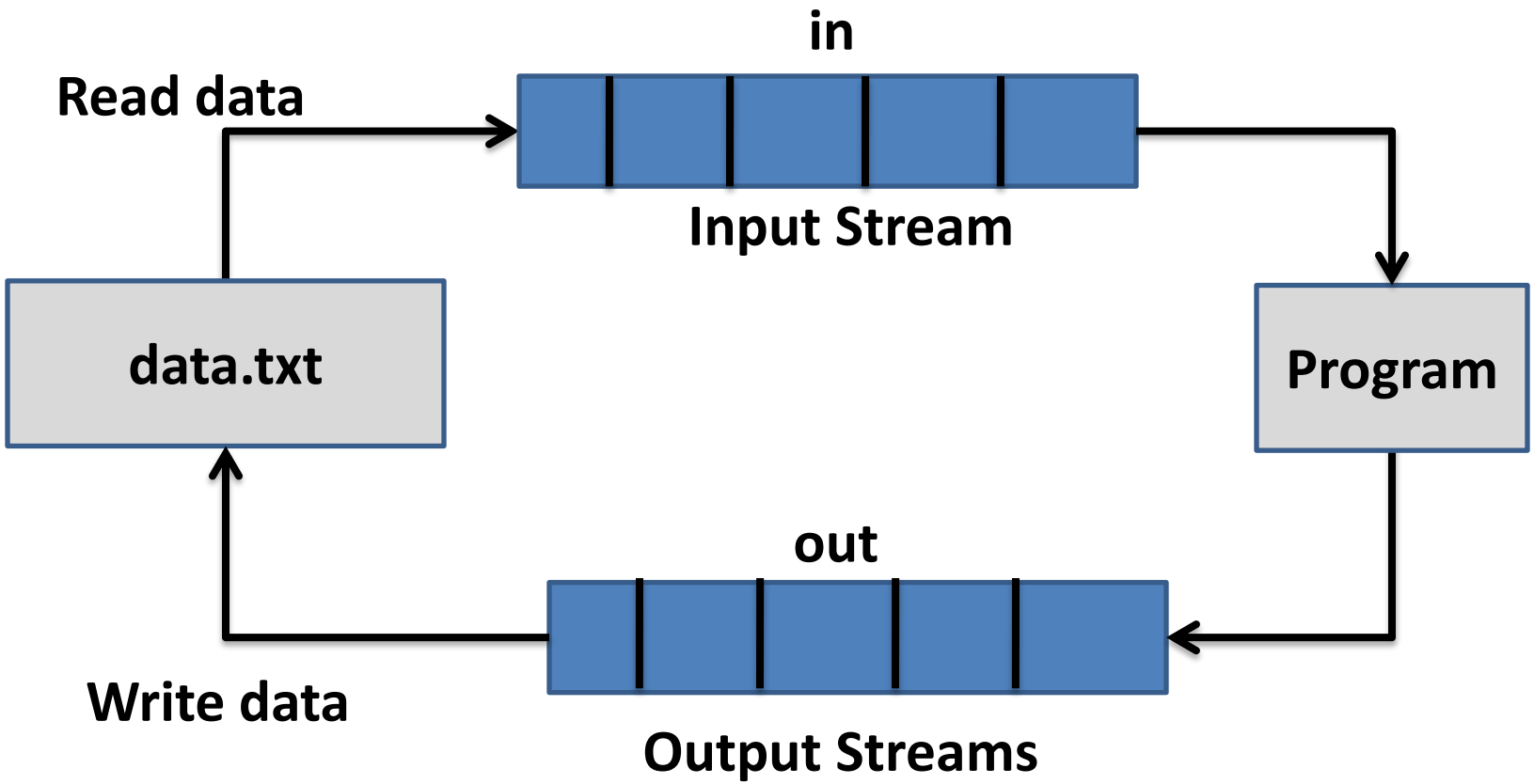


Opening & Closing a File

- **Opening (default mode):**
  - Create a file stream
  - Link it to the filename
  - Two method 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( );

Modes of File Opening

Parameter	Meaning
ios::in	opens file for reading only
ios::out	opens file for writing only
ios::app	opens file for appending at the end only
ios::binary	opens file in binary mode
ios::trunc	Deletes the content of the file if it exists
ios::ate	opens file for appending but at anywhere

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);
- **repostion 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 encounterd while reading otherwise false(zero)
fail()	returns true when an iput of output operation has failed
bad()	returns true if an invalid operation is attempted of any unrecoverable error as occurred. if it false it may possible to recover from any other error reported and continue operation
good()	returns true if no error has occurred, if it false , no further operations can be carried out.

8. Exception Handling

Introduction

- **Exception:** "An abnormal condition that arises in a code sequence at run time".

- An exception is a run-time error.

- Exception handling allows us to manage run-time errors in an orderly fashion.

Introduction

- Using exception handling, our program can automatically invoke an error-handling routine when an error occurs.

- C++ exception handling is built upon three keywords: **try, catch, and throw.**

Exception Handling Fundamentals

- **Error prone program statements** that we may want to monitor for generation of exceptions are contained in a **try block.**
- **Syntax:**

```
try {
    // try block
}
```

Exception Handling Fundamentals

- If an exception (i.e., an error) occurs within the **try block,** then that exception is thrown using **throw.**
- **Syntax:**

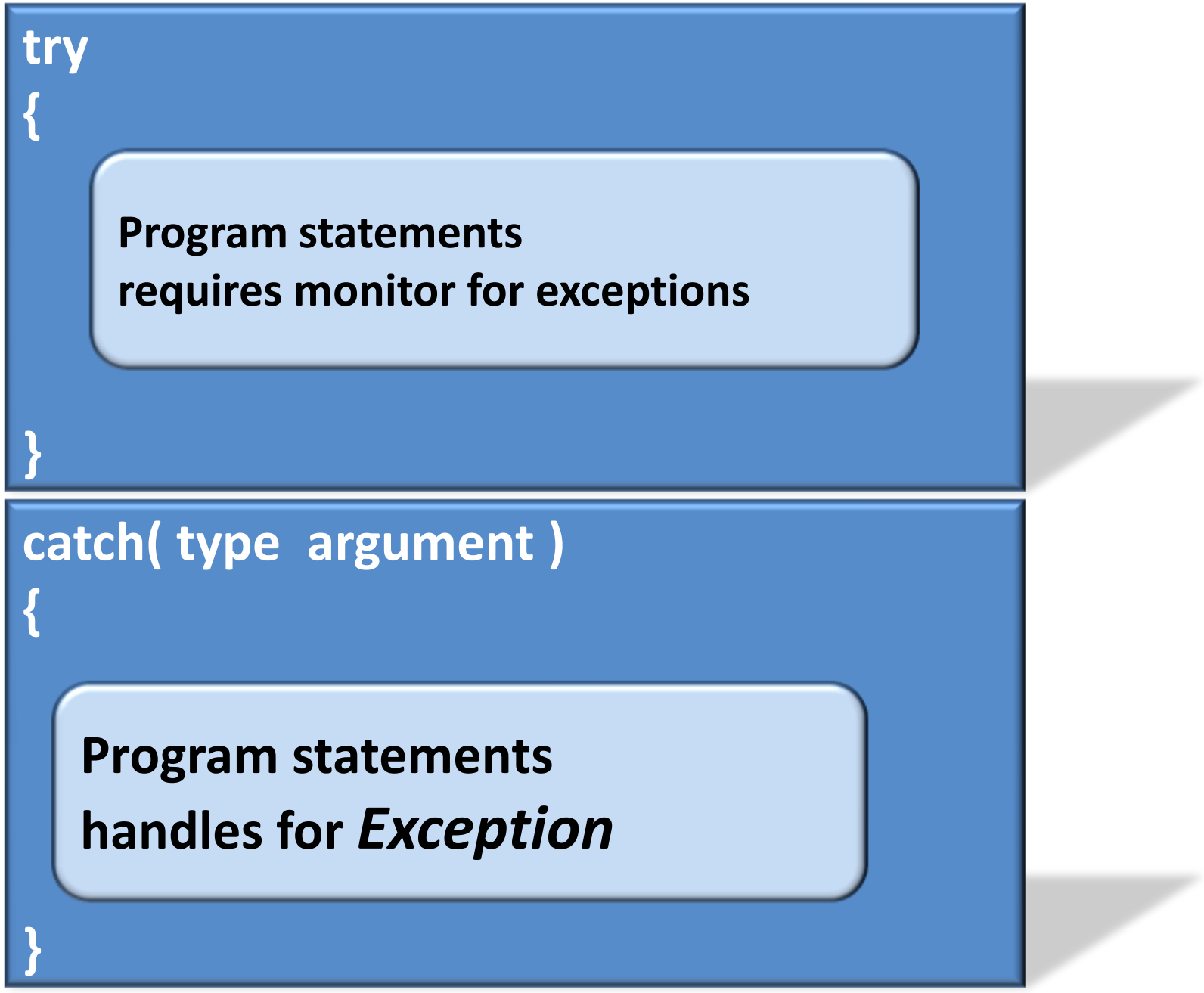
  throw exception;
- If an exception is to be caught, then throw must be executed either from **within a try block** or from any function called from within the **try block (directly or indirectly).**

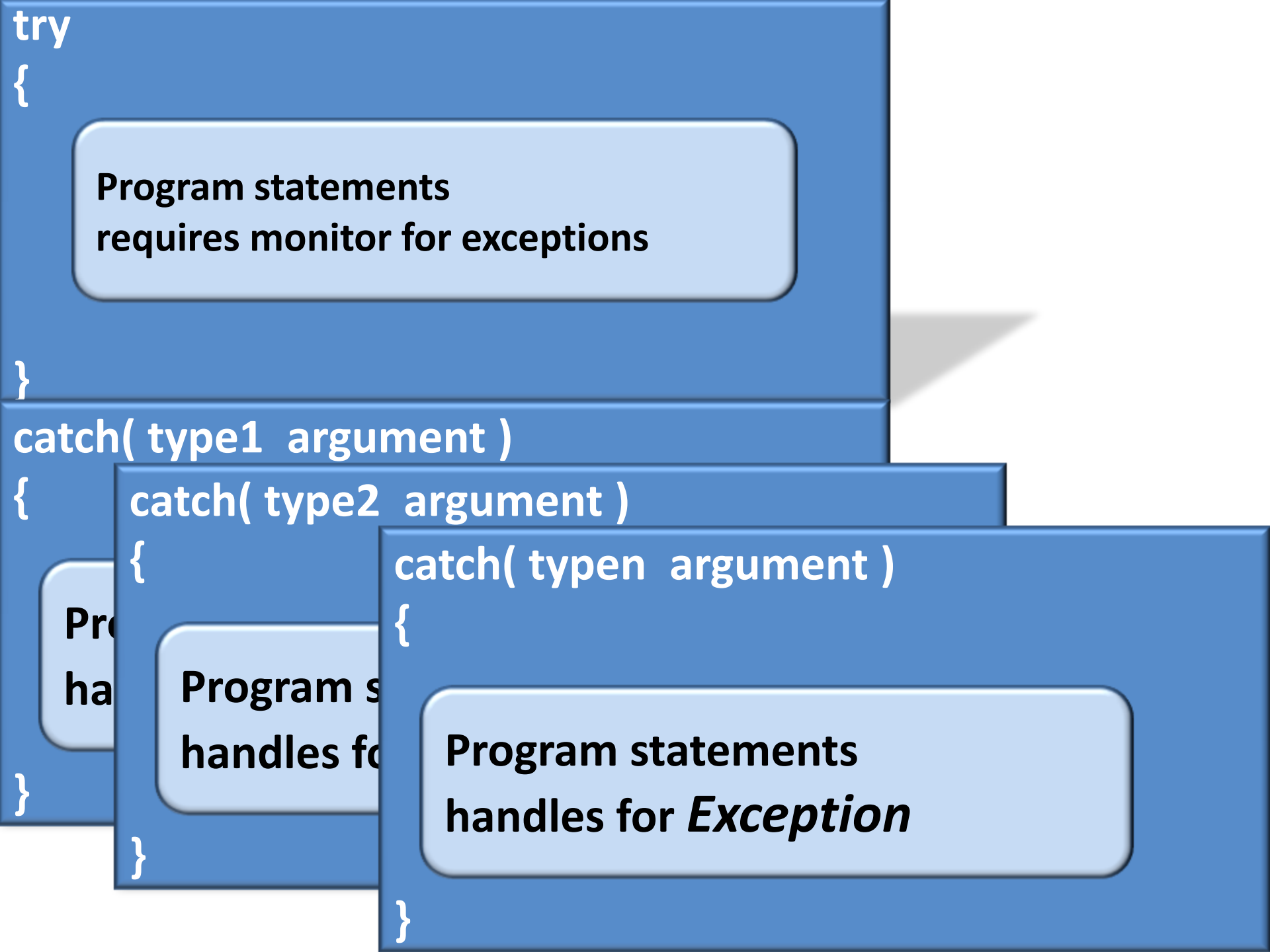
Exception Handling Fundamentals

- The thrown exception is caught, using **catch block** and processed.

- **Syntax:**

```
catch (type  argument)
{
  // catch block
}
```


try
{
Program statements
requires monitor for exceptions
}
catch(type argument)
{
Program statements
handles for *Exception*
}


try
{
Program statements
requires monitor for exceptions
}
catch(type1 argument)
{
Pr
ha
}
catch(type2 argument)
{
Program s
handles fo
}
catch(typen argument)
{
Program statements
handles for *Exception*
}

Using try & catch

```
try
{
    int d = 0;
    int a = 30 / d;
}
```

throws

Arithmetic Exception

```
catch(int e )
{
    printf("Division by zero.");
}
```

Exception Handling Fundamentals

NOTE:

Throwing an unhandled exception causes the standard library function **terminate()** to be invoked**.** By default, **terminate()** calls **abort()** to stop your program.