CSC127 – Classes and Objects in C++

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

Data Members or Properties of Student Class

Members Functions or Behaviours of Student Class

- **class** Student
- {
  - int st_id;
  - char st_name[];
  - void read_data();
  - void print_data();
- };

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

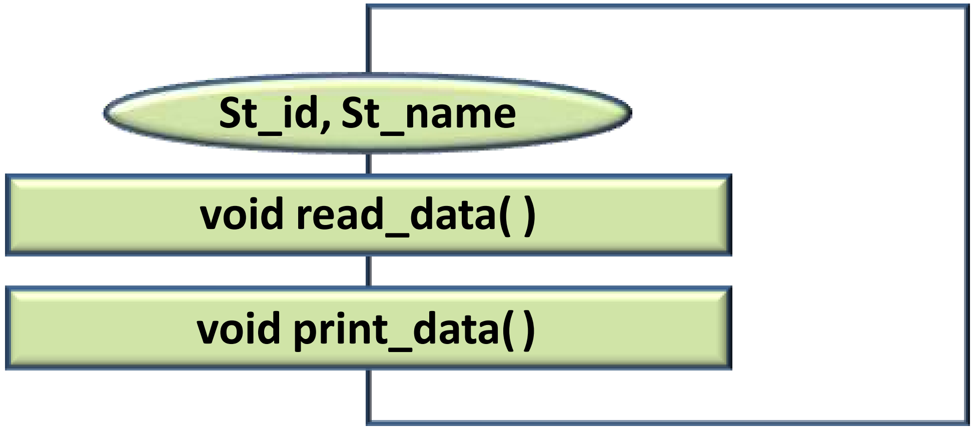
- **Object Instantiation:**

  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!



St_id, St_name

void read_data( )

void print_data( )

Class Object

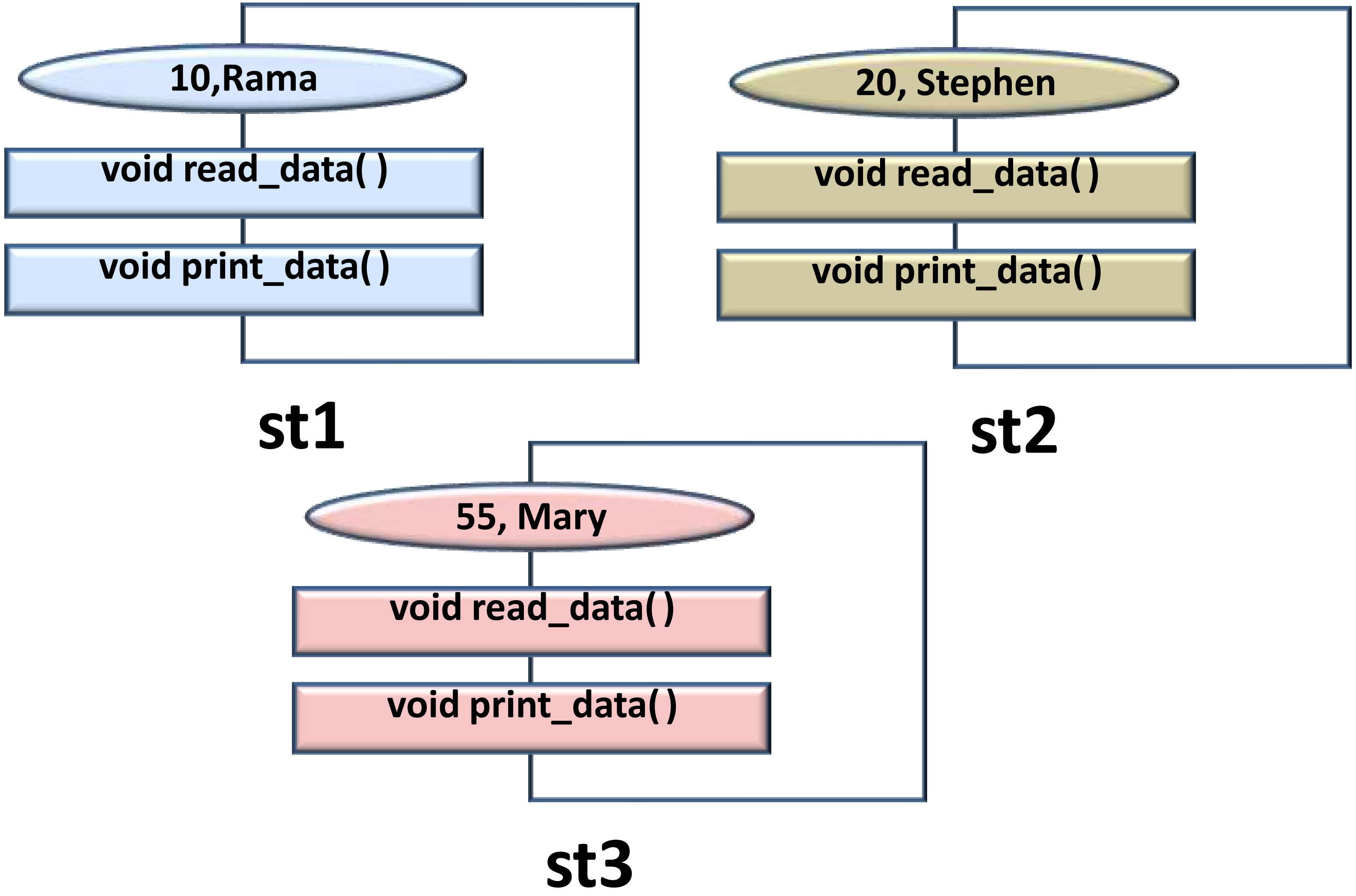
- **More of Objects**

  ex: Student st1;

  Student st2;

  Student st3;

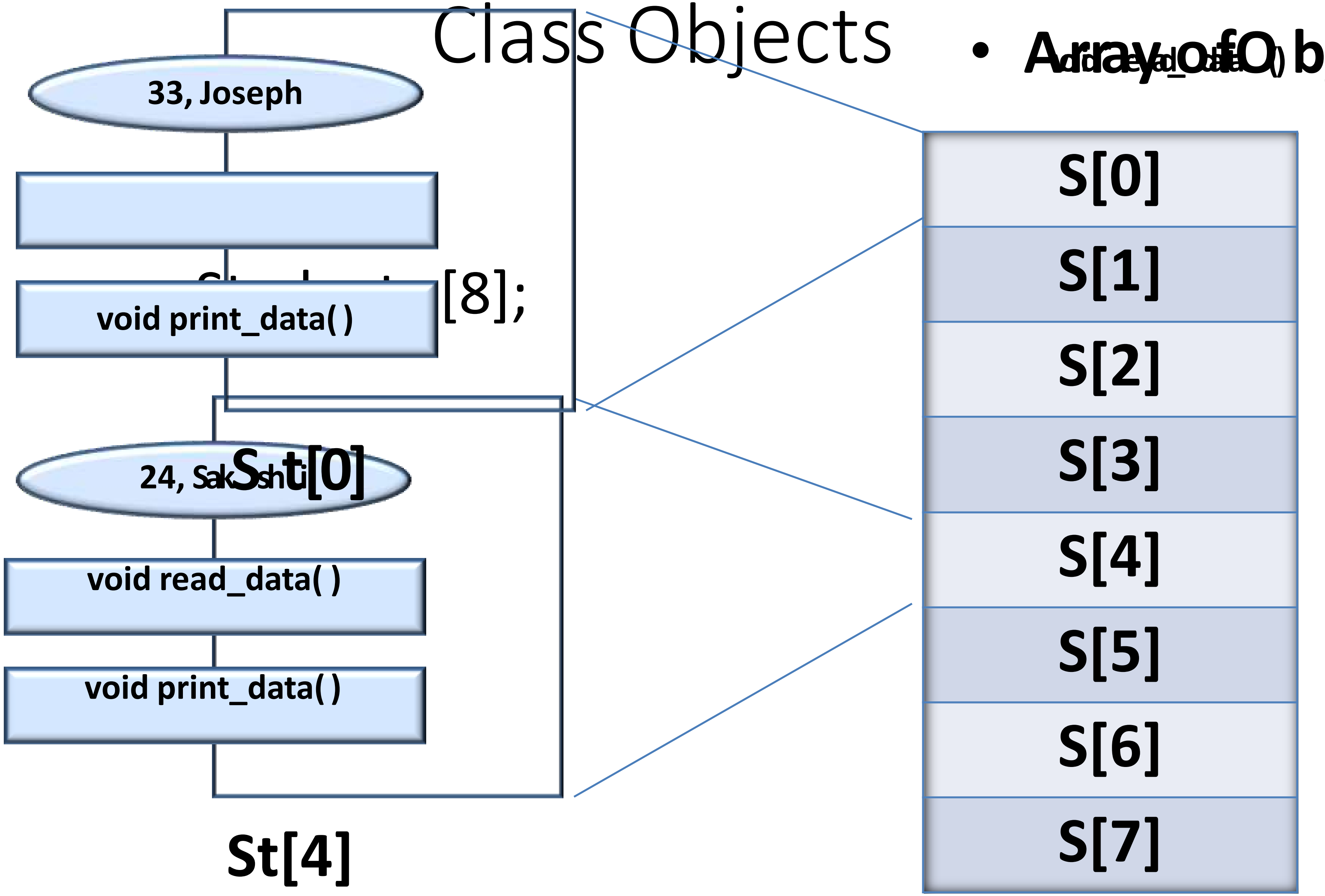
Class Objects



Class Objects

- **Array of Objects**

Student s[8];



Accessing Data Members

(outside the class)

- **Syntax: (single object)**

  obj_name . datamember;

  ex: Student st;

  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);

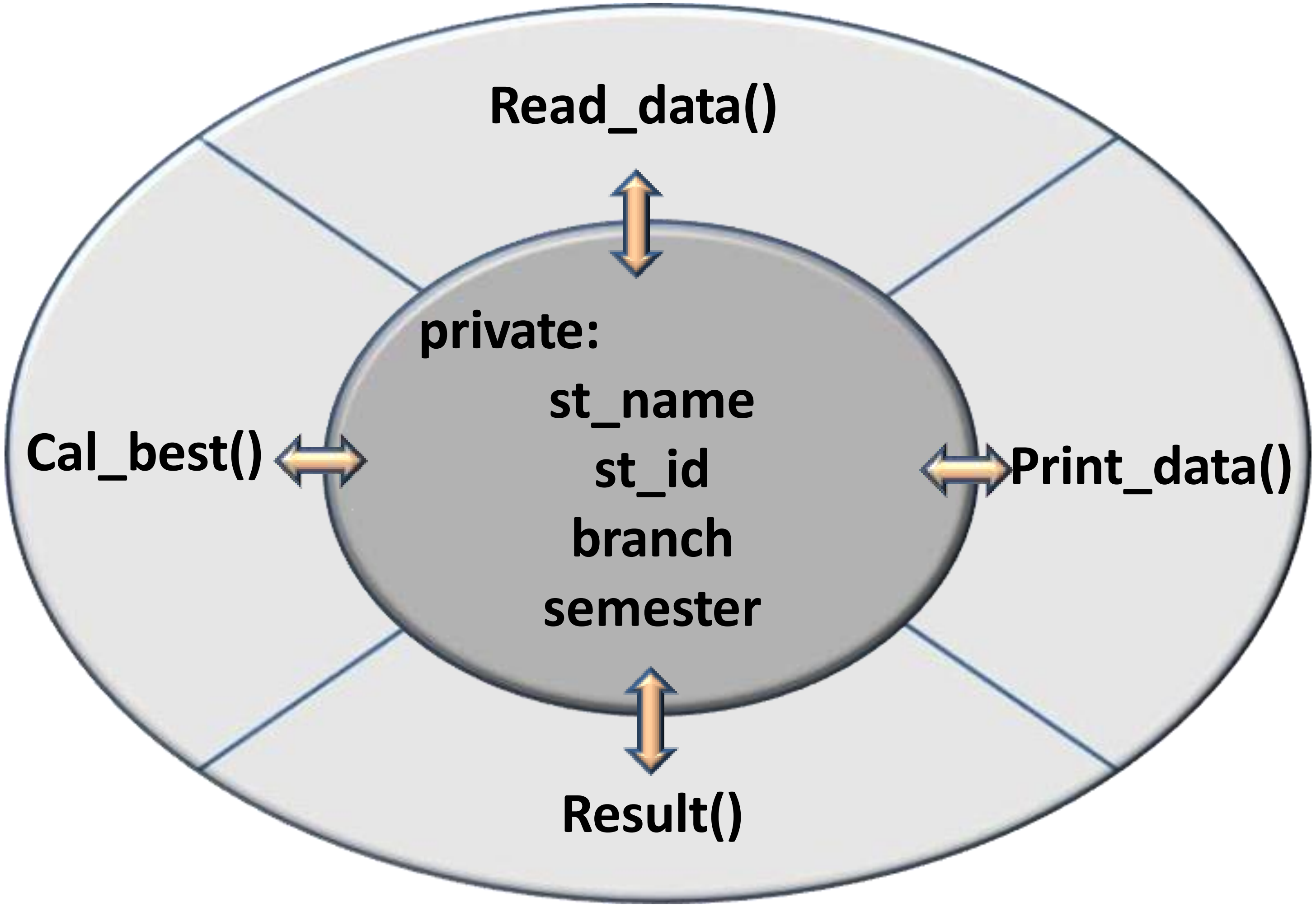
  ex: st[i].read( );

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:

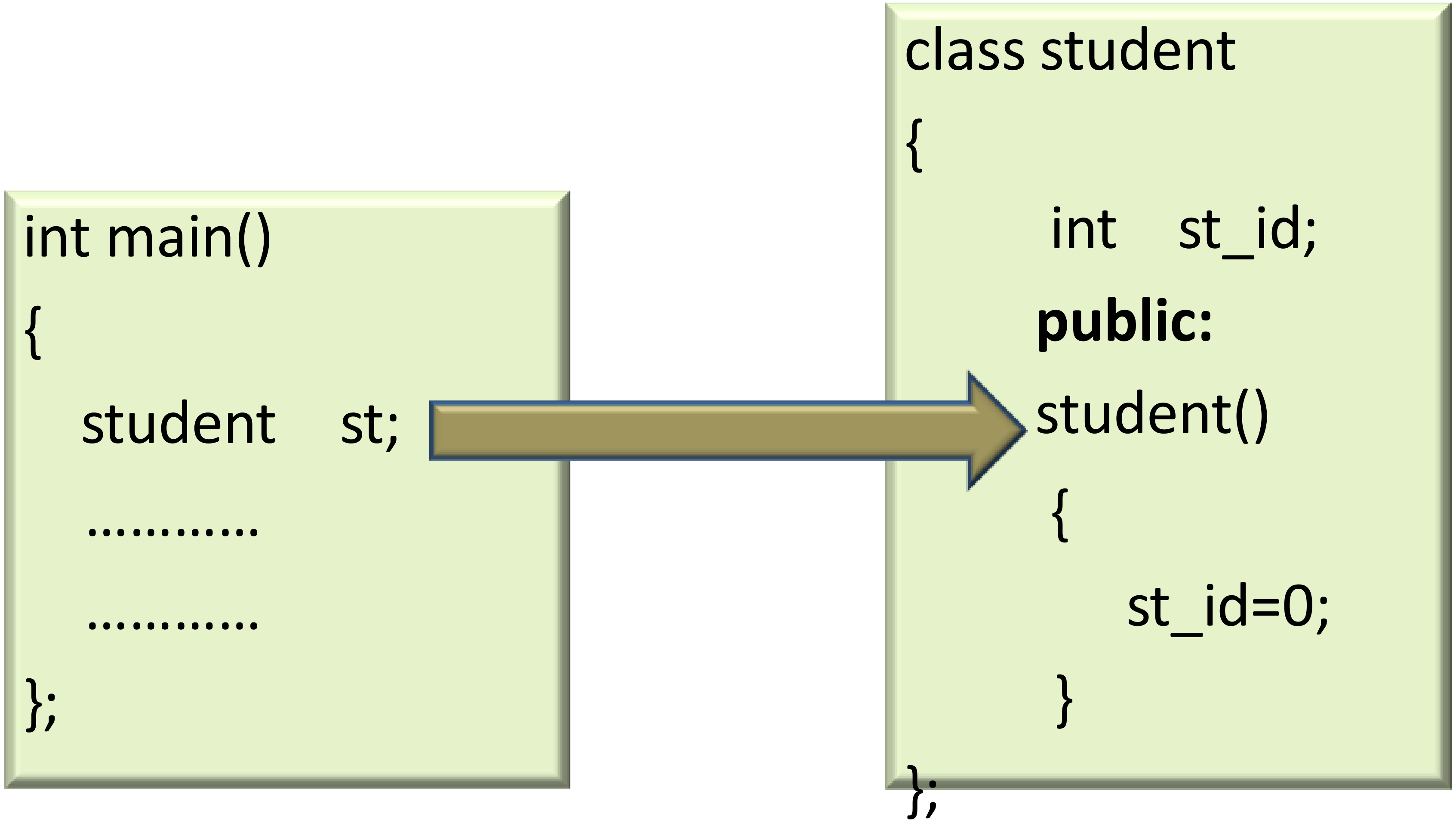
```
class  class_name
{
public:
class_name();
};
```

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

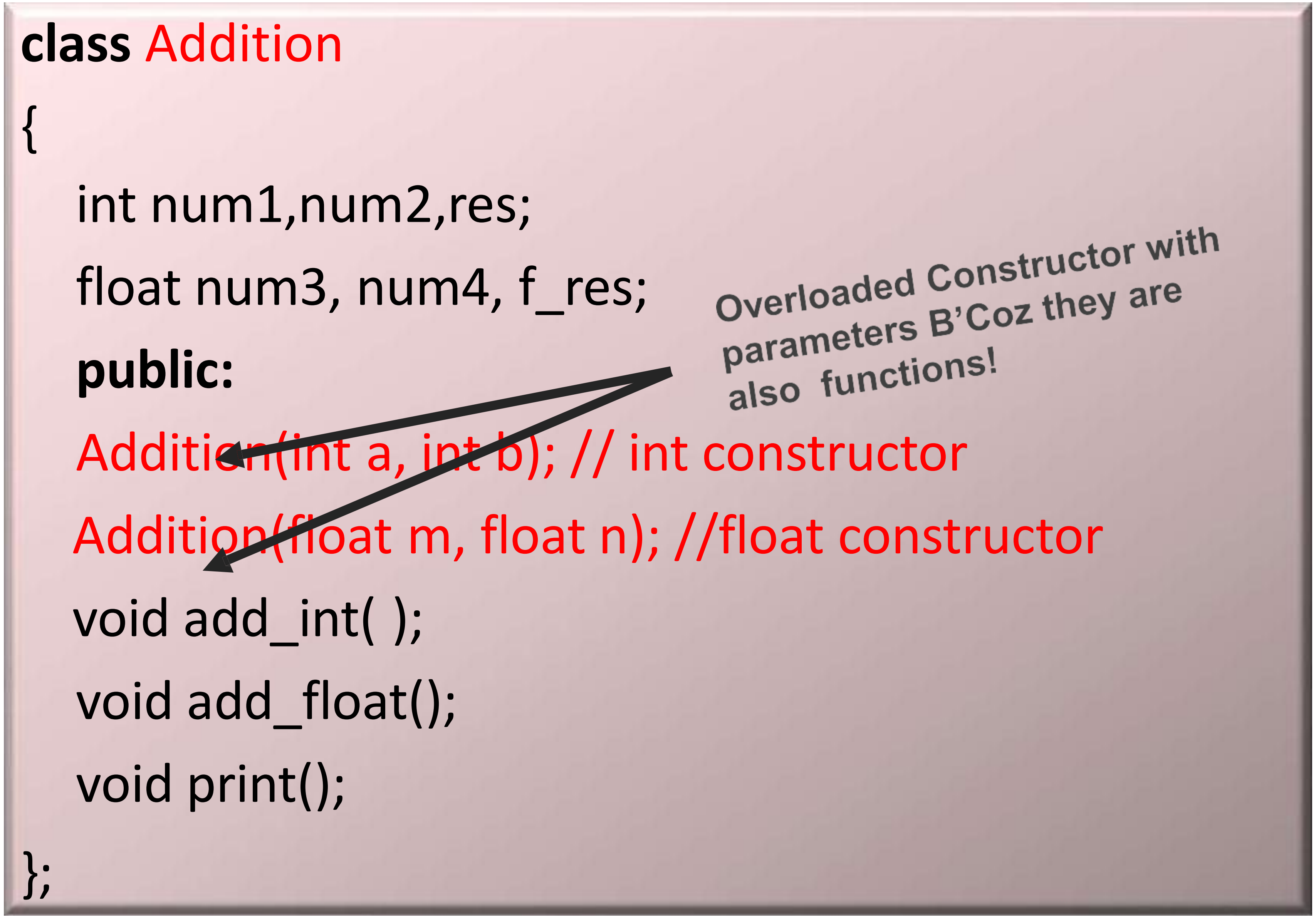
- Parameterized constructors
- Overloaded constructors
- Constructors with default argument
- Copy constructors
- Dynamic constructors

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

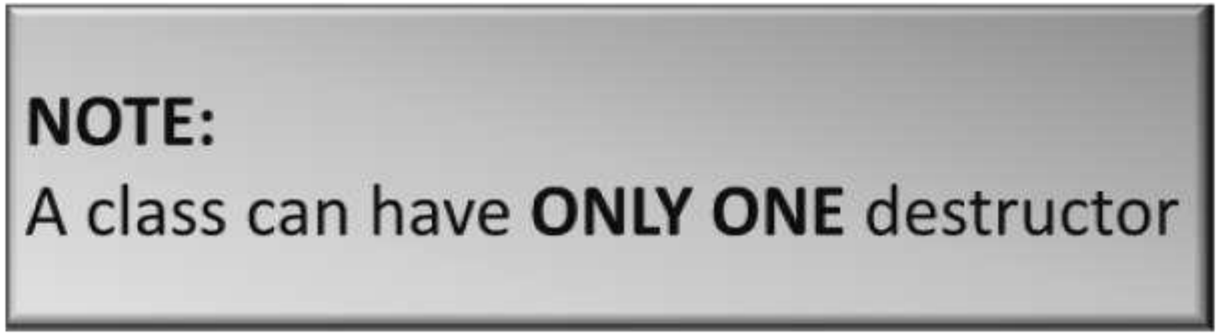
```
class code
{
        int id;
        public:
        code()    //constructor
        { id=100;}
        code(code &obj)   // constructor
        {
        id=obj.id;
        }
};
```

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;}
};
```

Static Data Members

- **Static data members** of a class are also known as **"class variables“.**

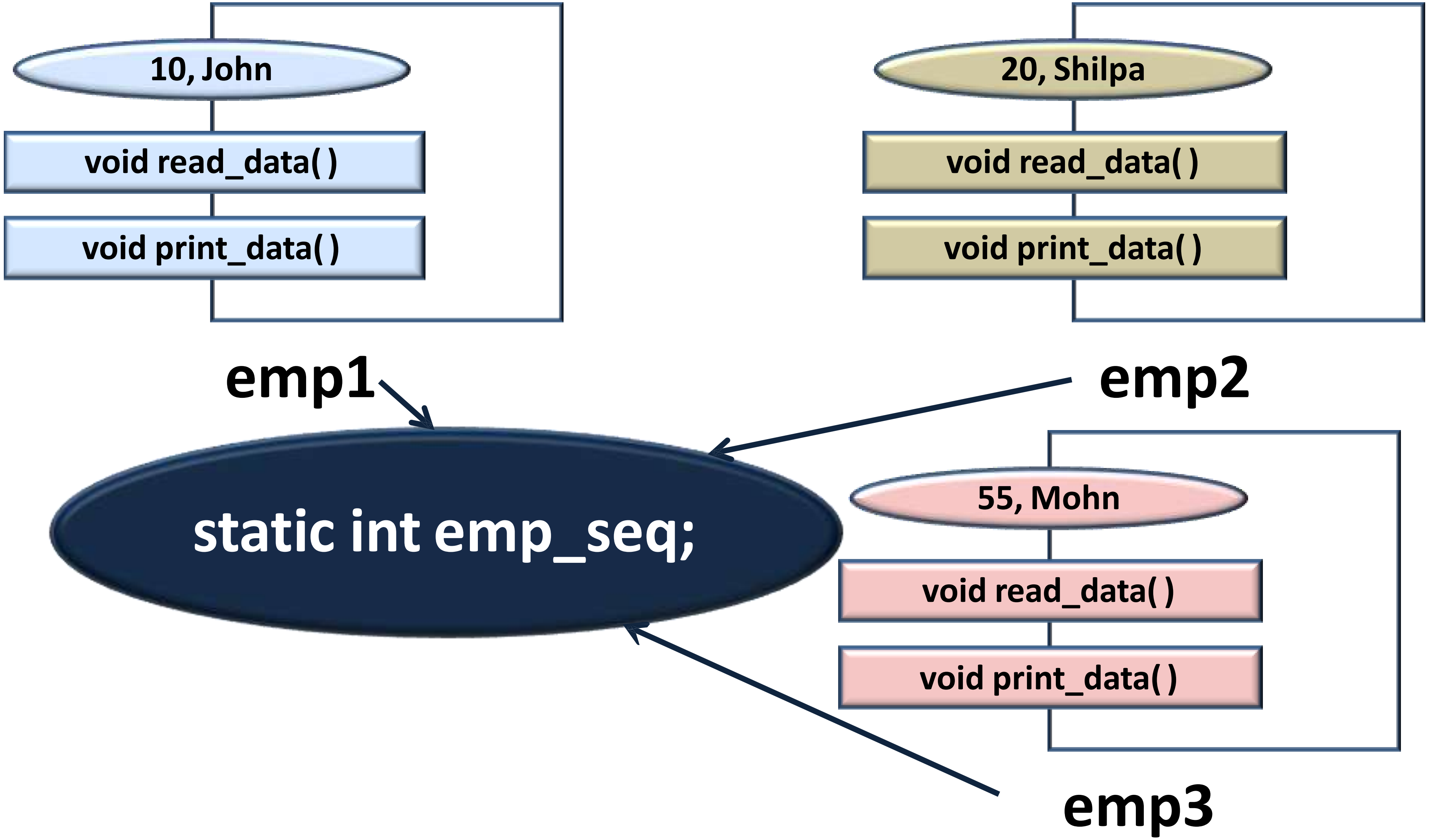
- Because their **content** does **not depend** on **any object**.

- 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
.
.
.
}
```

Solution.....?

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

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.