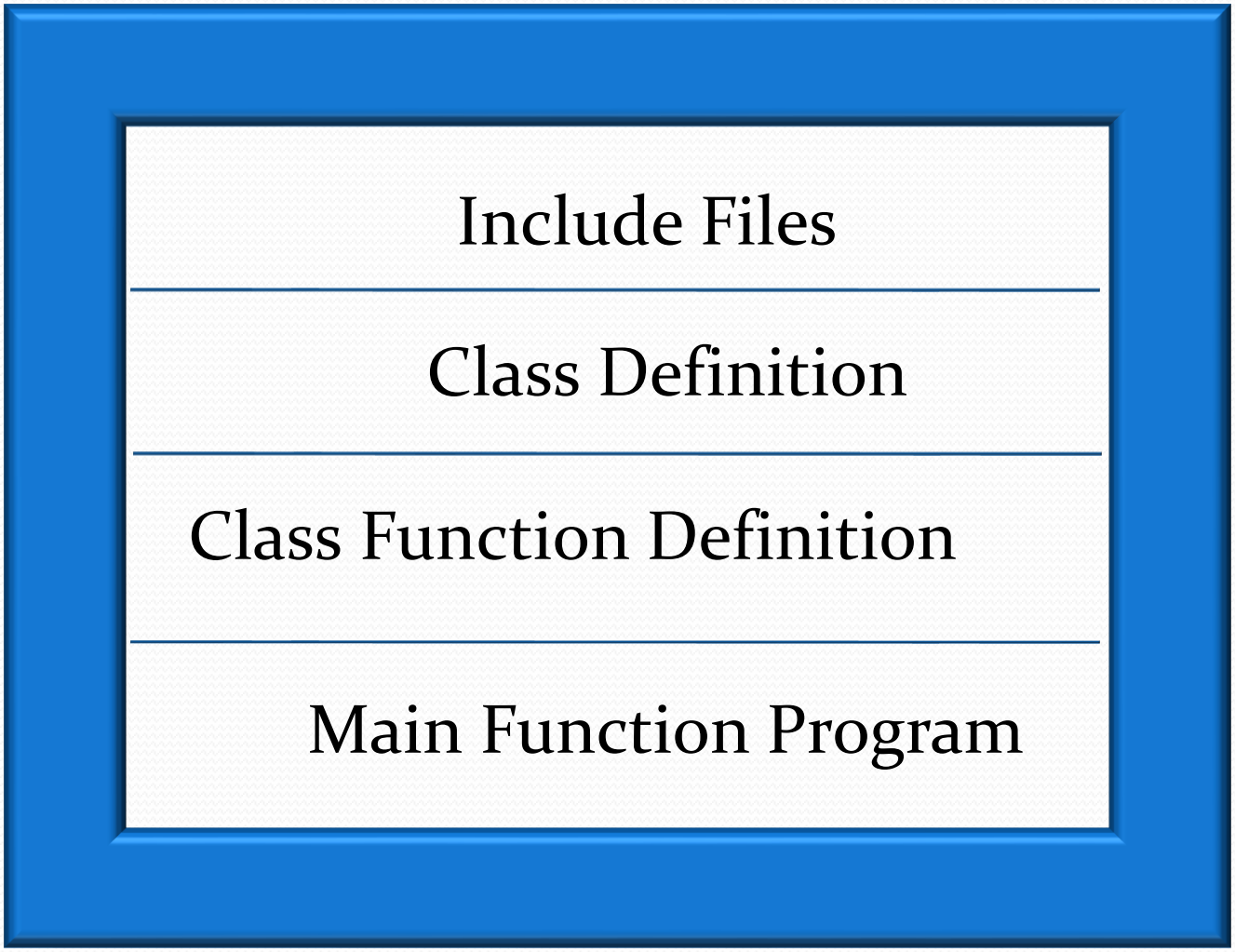
C++ Overview

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)

Defining Class

Class Specification

- **Syntax:**

class class_name

{

Data members

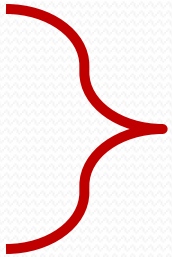
Members functions

};

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

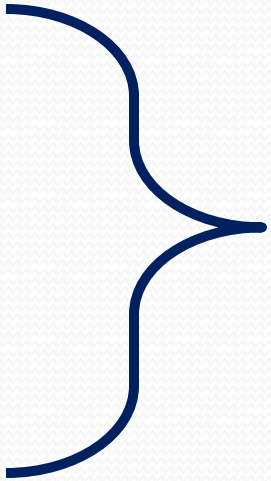
Default –

All items defined in the class are private.

Class Specification

- **class** Student

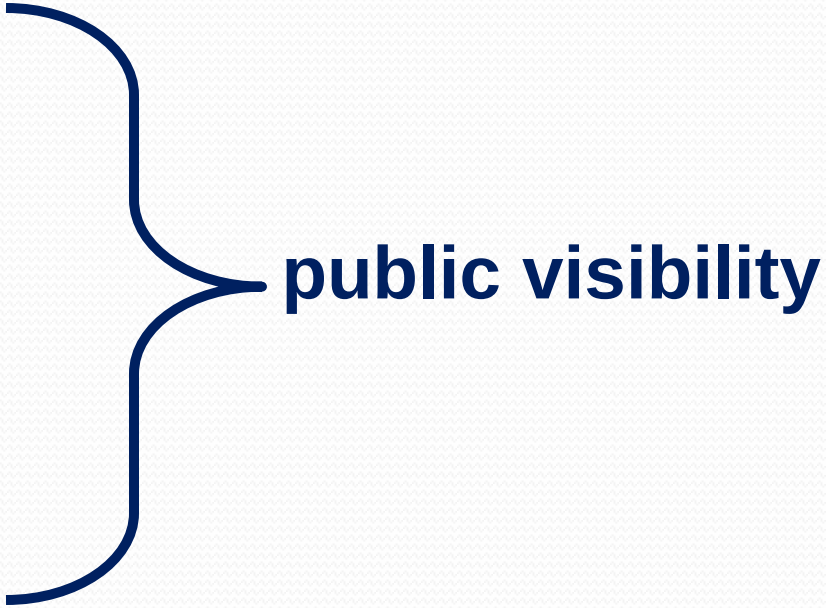
```
{
   int st_id;
  char st_name[];
  void read_data();
  void print_data();
};
```



private / default visibility

Class Specification

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



Class Objects

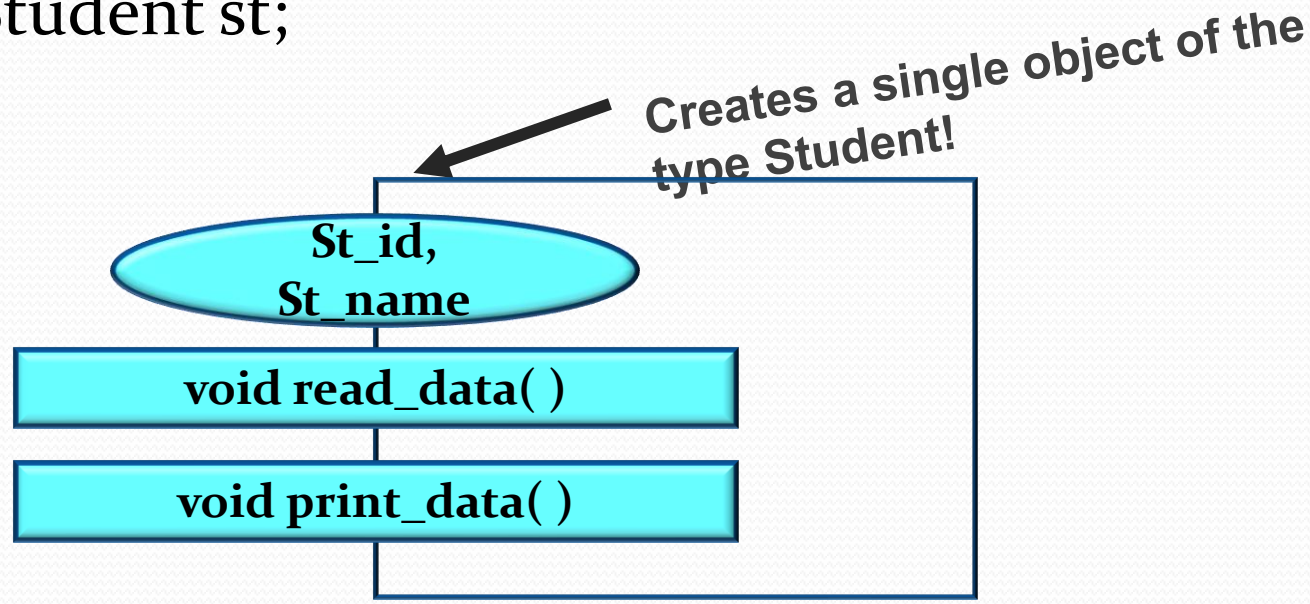
- **Object Instantiation:**

  The process of creating object of the type class

- **Syntax:**

  class_name obj_name;

  ex: Student st;



Class Object

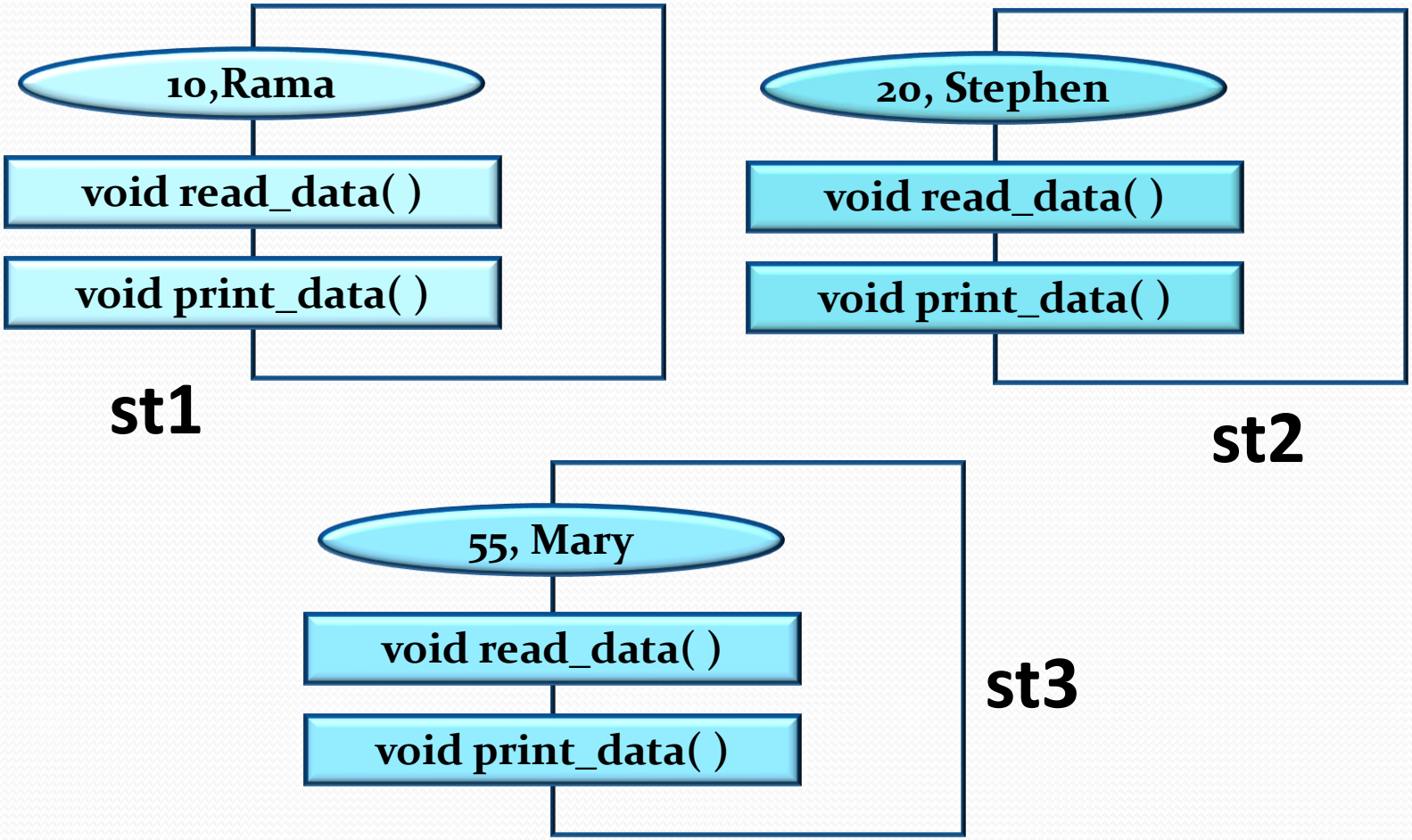
- **More of Objects**

  ex: Student st1;

  Student st2;

  Student st3;

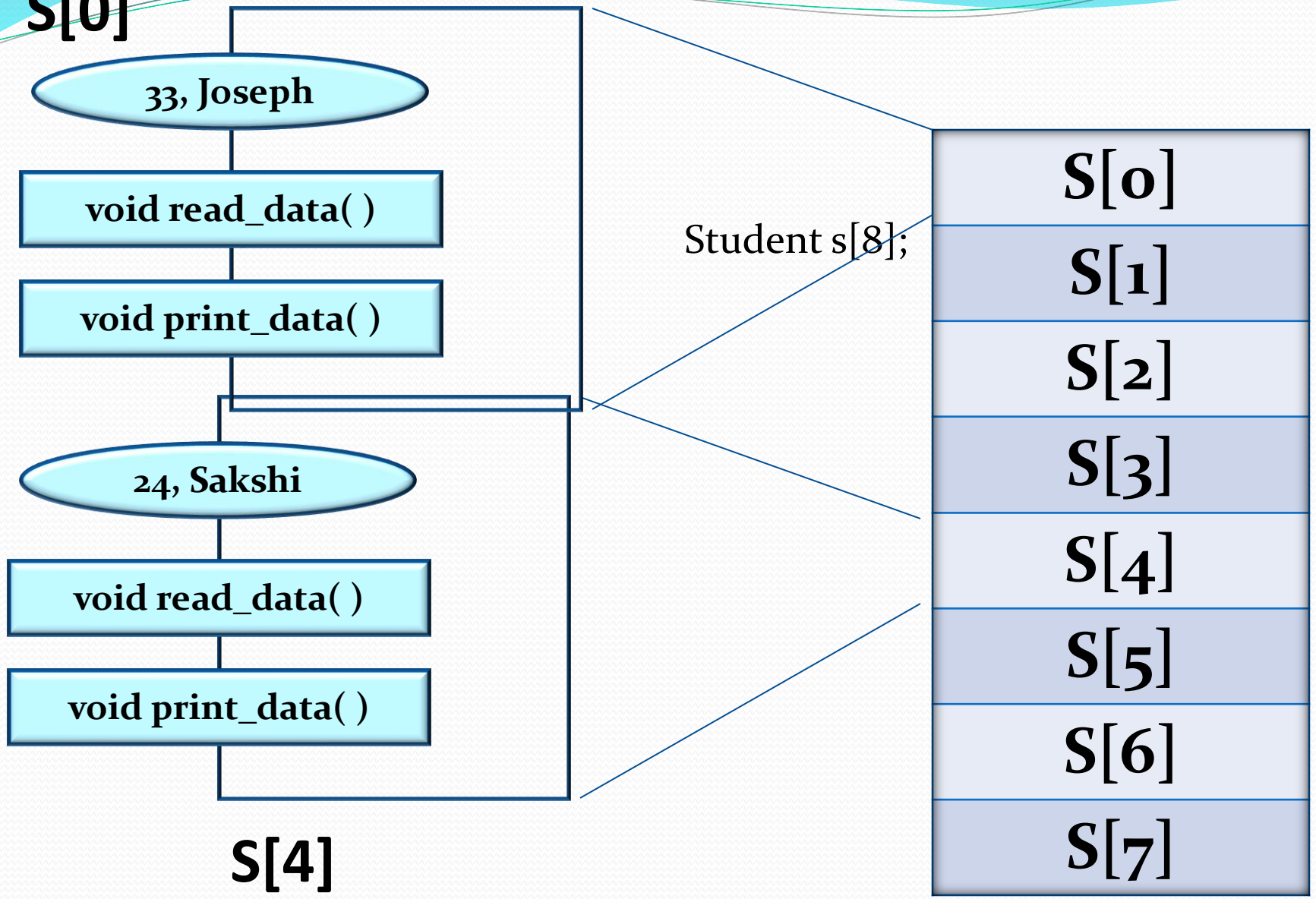
Class Objects



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.


S[0]
33, Joseph
void read_data( )
void print_data( )
24, Sakshi
void read_data( )
void print_data( )
S[4]
Student s[8];
S[0]
S[1]
S[2]
S[3]
S[4]
S[5]
S[6]
S[7]

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;

Defining Member Functions

(Inside the class definition)

- **Syntax**

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

Defining Member Functions

(Outside the class definition)

- **Syntax**

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( );

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

  }

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 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 can be called using class name like..
  - **Class name::function name;**

```
#include < iostream>
Class test
{
Int code;
Static int count;
Public:
Void setcode(void)
{  code=++count;}
Void showcode(void)
{ cout<<“object no.:”<<code;
Static void showcount(void)
{ cout<<“count:”<<count; }
};
Int test::count;
```

```
int main()
{
test t1,t2;
t1.setcode();
t2.setcode();
test::showcount();
test t3;
t3.setcode();
test::showcount();
t1.showcode();
t2.showcode();
t3.showcode();
Return 0;
}
```

Output

```
Count:2
Count :3
Object number:1
Object number:2
Object number:3
```

Constructors

- A **constructor** function is a special member 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.
- Invoked automatically when objects are created

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 functio**

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

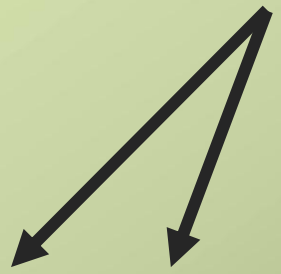
```
class student
{
        int    st_id;
        public:
        student()
        {
            st_id=0;
        }
};
```

Types of Constructors

- Parameterized constructors
- Constructors with default argument
- Overloaded Constructor
- 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!

Constructor that can take arguments is called parameterized constructor.

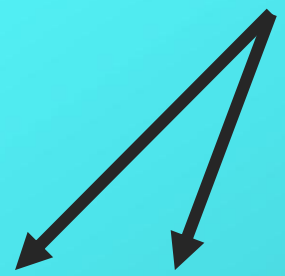
Overloaded Constructors

```
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) // Copy constructor
        {
        id=obj.id;
        }
        void display()
        {
                cout<<id;
         }
};
```

```
int main()
{   code A(100);
     code B(A);
     code C=A;
     code D;
     D=A;   // wrong syntax
cout<<" id of A:";
 A.display();
     cout<<" id of B:";
B.display();
     cout<<" id of C:";
C.display();
    cout<<" id of D:";
D.display();
```

Copy constructor is used to declare and initialize an object from another object

Dynamic Constructors

Used to allocate memory at the time of object creation

```
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”;
        }
};
```

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

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 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;}
};
```

Friend Functions

- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.
- To declare a **friend function**, its **prototype** should be included within the class, preceding it with the keyword **friend.**

Friend Function

Characteristics:

- Not in the scope of class to which it has been declared as friend
- It can not be called using object of class
- Invoked like normal function
- Can not access data members directly, has to use object and dot operator
- Can be declare in private or public section.
- It has objects as arguments.

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);
};
```

```
#include <iostream>
using namespace std;
class Box
{
 double width;
public:
        friend void printWidth( Box box );
        void setWidth( double wid );
};
 // Member function definition
void Box::setWidth( double wid )
{
 width = wid; }
void printWidth( Box box )          // Note: printWidth() is not a member function of any class.
 {
    /* Because printWidth() is a friend of Box, it can directly access any member of this class */
cout << "Width of box : " << box.width <<endl; }
int main()
 {
        Box box;                                // set box width without member function
        box.setWidth(10.0);
        printWidth( box );                      // Use friend function to print the width.
return 0; }
```

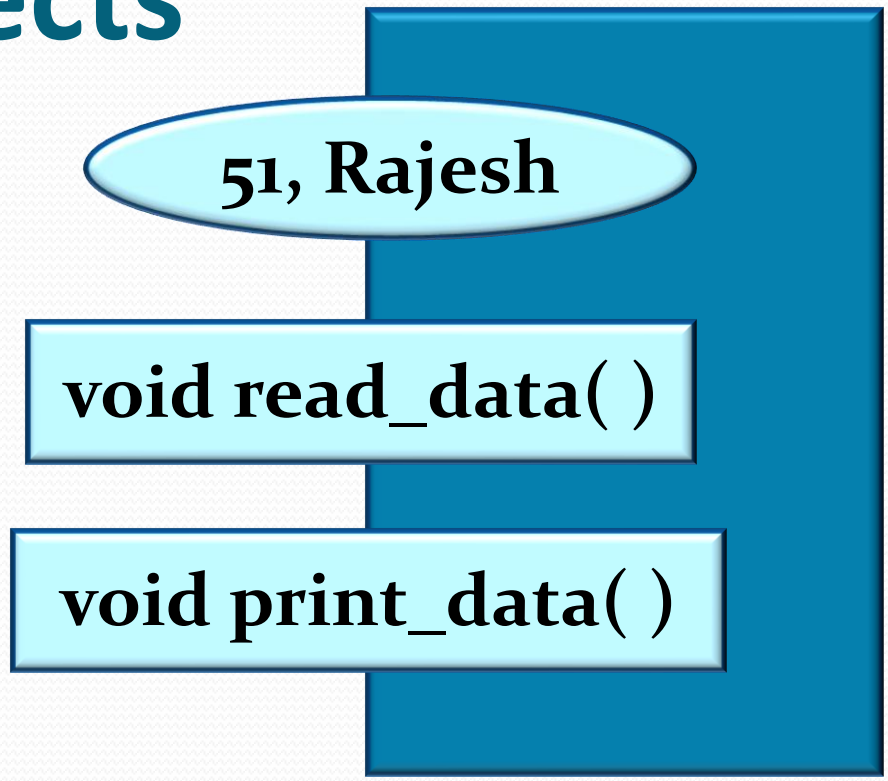
Width of box : 10

Pointers to Objects

```
student   st;

student  *ptr;

ptr= &st;
```

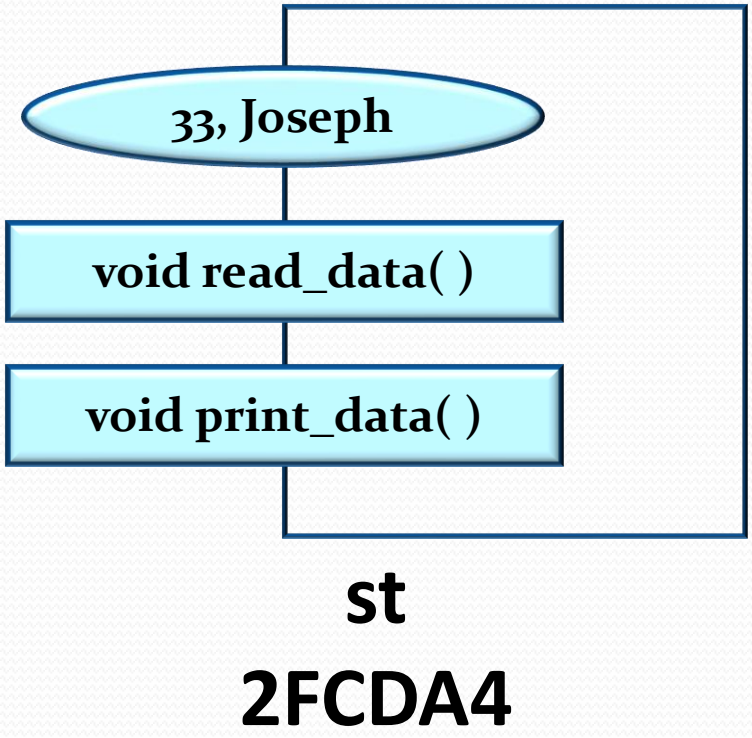


st

2FCD54

Pointers to Objects

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



Statically created object:

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

Dynamically created object:

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

Pointers to Objects

- **Accessing Members of objects:**

  Syntax:

  ptr_obj→member_name;

  ptr_obj→memberfunction_name( );

  Example:

  stp→st_name;

  stp→read_data( );

1. Each object gets its own copy of the data member.
2. All access the same function definition as present in the code segment

- Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?
  Compiler supplies an implicit pointer along with the functions names as ‘this’.

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 ( );

When local variable's name is same as member's name

```
#include<iostream>
using namespace std;
 /* local variable is same as a member's name */
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
      // The 'this' pointer is used to retrieve the object's x
      // hidden by the local variable 'x'
      this->x = x;
   }
   void print() { cout << "x = " << x << endl; }
};
int main()
{
   Test obj;
   int x = 20;
   obj.setX(x);
   obj.print();
   return 0;
}
```

To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
   // Some processing
   return *this;
}
```

- When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};
 int main()
{
  Test obj(5, 5);
   // Chained function calls.  All calls modify the same object as the same object is returned by
reference
  obj.setX(10).setY(20);
   obj.print();
  return 0;
}
```

Pointer to Class Member

- Just like pointers to normal variables and functions, we can have pointers to class member functions and member variables.
- 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.**
- A pointer to a member is **not** the same as a normal **C++ pointer**.

Pointer to Data Members of class

- **Syntax for Declaration :**
  - datatype class_name :: *pointer_name;
  - int student::*d_ptr;

- **Syntax for Assignment :**
  - pointer_name = &class_name :: datamember_name ;
  - **To access a member of a class:**

  Special pointer-to-member operators

  1) .* (Object.*pointerToMember)
  2) –>* (ObjectPointer->*pointerToMember)

```
class Data
{
 public:
 int a;
void print()
{ cout << "a is "<< a; }
 };
 int main()
{
Data d, *dp;
dp = &d; // pointer to object
 int Data::*ptr=&Data::a;
 // pointer to data member 'a'
d.*ptr=10;
d.print();
dp->*ptr=20;
dp->print(); }
```

a is 10 a is 20

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)();

Thank you