PROGRAMMING IN C++

UNIT – 2

|  |
| --- |
| #include <iostream.h>  class XYZ {  private:  int num=100;  char ch='Z';  public:  friend void disp(XYZ obj);  };  //Global Function  void disp(XYZ obj)  {  cout<<obj.num<<endl;  cout<<obj.ch<<endl;  }  int main() {  XYZ obj;  disp(obj);  return 0;  }  **CLASSES AND OBJECTS**  **CLASS**   * A class is a way to bind the data and its associated functions together. * It allows the data (and functions) to be hidden. If necessary , from external use. * When defining a class, we are creating a new abstract data type that can be treated like any other built-in data type. * Generally , a class specification has two parts: * Class declaration * Class function definitions * The class declaration describes the type and scope of its members. * The class function definitions describe how the class functions are implemented.   The general form of a class declaration is :  C:\Users\admin\Documents\Bluetooth\New Doc 2020-01-20 11.10.21.jpg   * The body of a class is enclosed within braces and terminated by a semicolon. * The class body contains the declaration of variables and functions. These function and variables are collectively called class members. * They are usually grouped under two sections, namely, private and public to denote which of the members are private and which of them are public. * The keywords private and public are known as visibility labels. * The variables declared inside the class are known as data members and the functions are known as member functions. * The binding of data and functions together into a single class-type variable is referred to as encapsulation. * Class's member functions can be defined inside the class definition or outside the class definition. * All the features of OOPS, revolve around classes in C++. Inheritance, Encapsulation, Abstraction etc. * Objects of class holds separate copies of data members. We can create as many objects of a class as we need.  Objects of Classes  * Class is mere a blueprint or a template. * No storage is assigned when we define a class. * Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects. * Each object has different data variables.   class Abc  {  int x;  void display()  {  // some statement  }  };  int main()  {  Abc obj; // Object of class Abc created  } |

|  |  |
| --- | --- |
|  |  |

**Creating Objects**

* Once a class has been declared, we can create variables of that type by using class name.
* In C++ , the class variables are known as objects.

Class item

{

….

….

….

}

Item x,y,z;

**ACCESSING CLASS MEMBERS**

* **The private data of a class can be accessed only through the member functions of that class.**
* **The main cannot contain statements that access number and cost directly.**
* **The following is the format for calling a member function.**

**Object-name.function-name(actual-arguments);**

* **For example, the function call statement**

**x.getdata(100,75.5);**

**DEFINING MEMBER FUNCTIONS**

* **Member functions can be defined in two places**
  + - **Outside the class definition**
    - **Inside the class definition**
* **It is obvious that, irrespective of the place of definition, the function should perform the same task.**

**Outside the class definition**

* **Member functions that are declared inside a class have to be defined separately outside the class.**
* **Their definitions are very much like the normal functions.**
* **They should have a function header and a function body.**
* **The general form of a member function definition is:**

**return-type class-name :: function-name (argument declaration)**

**{**

**function body**

**}**

* **The membership label class-name :: tells the compiler that the function function-name belongs to the class class-name.**
* **That is, the scope of the function is restricted to the class-name specified in the header line.**
* **The symbol :: is called the scope resolution operator.**

**Example:**

**Void item :: getdata(int a,float b)**

**{**

**Number=a;**

**Cost=b;**

**}**

**INSIDE THE CLASS DEFINITION**

**Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.**

**Class item**

**{**

**Int number;**

**Float cost;**

**Public:**

**Void getdata(int a,float b);**

**Void putdata(void)**

**{**

**Cout<<number<<”\n”;**

**Cout<<cost<<”\n”;**

**}**

**};**

**PRIVATE MEMBER FUNCTIONS:**

* A private member function can only be called by another function that is a member of its class.
* Even an object cannot invoke a private function using the dot operator.

class sample

{

int m;

void read(void);// private member function

public:

void update(void);

void write(void);

};

if s1 is an object of sample,then

s1.read(); // wont work; objects cannot access

//private members

void sample :: update(void)

{

read();

}

**STATIC DATA MEMBERS**

* A data member of a class can be qualified as static.
* The properties of a static member variable are similar to that of a C static variable.
* A static member variable has certain special characteristics.
  + It is initialized to zero when the first object of its class is created
  + Only one copy of that member is created for the entire class and is shared by all the objects of that class, matter how many objects are created.
  + It is visible only within the class but its lifetime is the entire program.
* Static int a=10;

Sample::a=30;

**STATIC MEMBER FUNCTION**

* A member function that is declared static has the following properties.
* A static function can have access to only other static members (functions or variables) declared in the same class.
* A static member function can be called using the class name (instead of its objects) as follows.

**Arrays of objects**

* An array of objects is stored inside the memory in the same way as a multi-dimensional array.
* Member functions are stored separately and will be used by all the objects.

Class employee

{

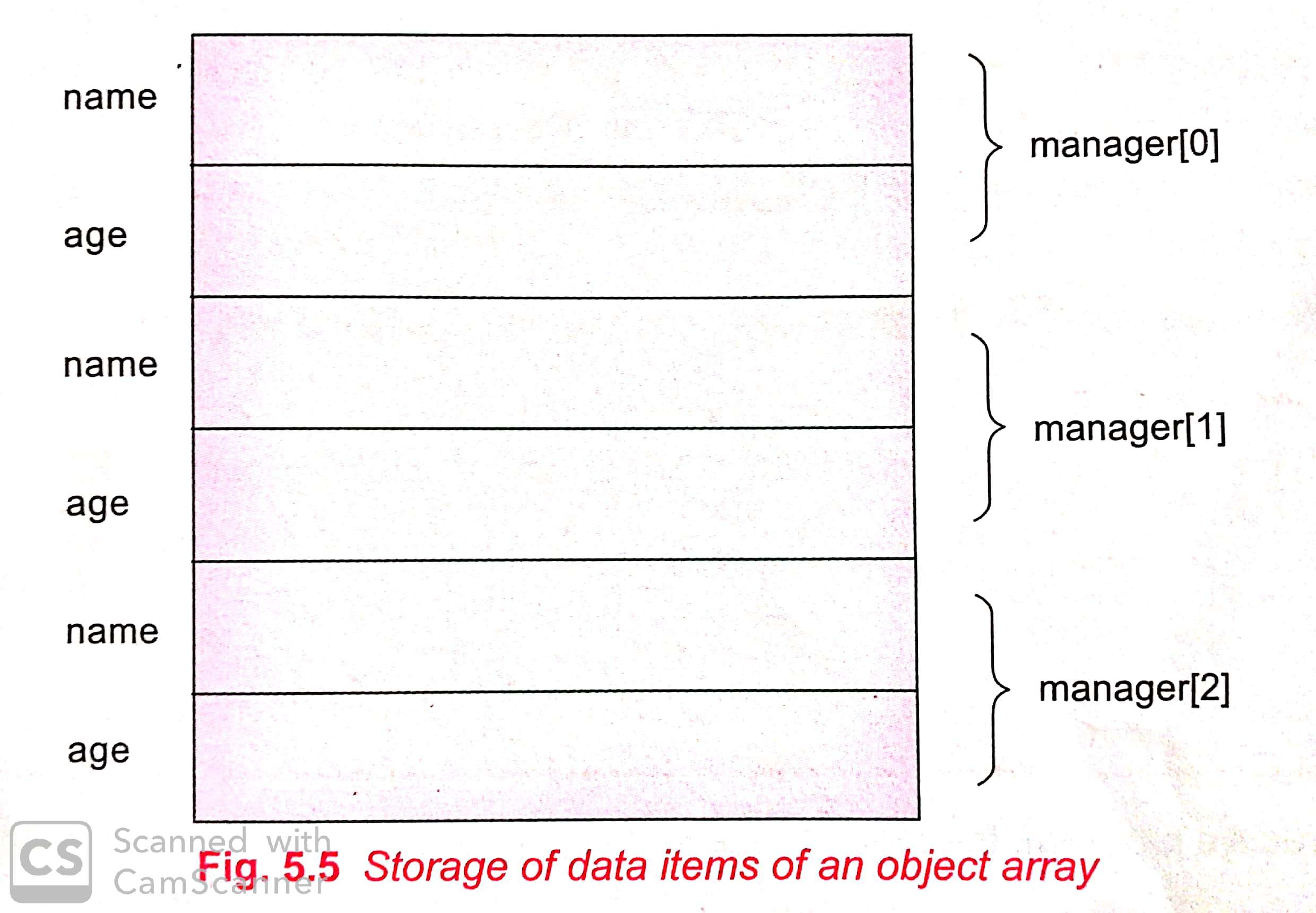
….

};

Employee manager[3];

Manager[0].getdata();

Manager[1].getdata();



Exercise programs.

* Write a program to take the input of 10 faculty details (ID,name,post,qualification,address) and display it.

**CHAPTER -6**

**CONSTRUCTOR**

* A constructor is a special member function whose task is to initialize the objects of its class.
* It is special because its name is the same as the class name.
* The constructor is invoked whenever an object of its associated class is created.
* It is called constructor because it constructs the values of data members of the class.
* A constructor is declared and defined as follows.

class integer

{

int m,n;

public:

integer()

{

……………….

}

……..

……

};

* A constructor that accepts no parameters is called the default constructor.
* The default constructor for class A is A::A().
* If n such constructor is defined, then the compiler supplies a default constructor.
* Therefore a statement such as

A a;

* The constructor functions have some special characteristics. These are
  + They should be declared in the public section.
  + They are invoked automatically when the objects are created.
  + They do not have return types, not even void and therefore, and they cannot return values.
  + They cannot be inherited, though a derived class can call the base class constructor.

Class simple

{

Int m,n;

Public:

simple ();

……

……

};

simple:: simple ()

{

M=0;

N=0;

}

simple s1;

Example program for Default Constructor

#include <iostream>

  class construct

{

public:

    int a, b;

    construct()// Default Constructor

    {

        a = 10;

        b = 20;

    }

};

  int main()

{

    // Default constructor called automatically

    // when the object is created

    construct c;

    cout << "a: " << c.a << endl

         << "b: " << c.b;

    return 0;

}

**PARAMETERIZED CONSTRUCTORS**

* C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created.
* The constructors that can take arguments are called parameterized constructors.
* The constructor simple() may be modified to take arguments as shown below.

class simple

{

int m,n;

public:

simple(int x,int y); //parameterized constructor

……..

………

};

simple::simple(int x,int y)

{

m=x;

n=y;

}

Simple s1(12,65);

**// parameterized constructors**

**#include <iostream>**

**class Point**

**{**

**private:**

**int x, y;**

**public:**

**// Parameterized Constructor**

**Point(int x1, int y1)**

**{**

**x = x1;**

**y = y1;**

**}**

**int getX()**

**{**

**return x;**

**}**

**int getY()**

**{**

**return y;**

**}**

**};**

**int main()**

**{**

**// Constructor called**

**Point p1(10, 15);**

**// Access values assigned by constructor**

**cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();**

**return 0;**

**}**

Multiple constructors in a class

* The constructor itself supplies the data values and no values are passed by the calling program.
* In the second case, the function call passes the appropriate values from main().
* C++ permits us to use both these constructors in the same class.
* In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading.

**Example Program.**

**#include <iostream.h>**

**class cons**

**{**

**public:**

**float area;**

**// Constructor with no parameters**

**cons()**

**{**

**area = 0;**

**}**

**// Constructor with two parameters**

**cons(int a, int b)**

**{**

**area = a \* b;**

**}**

**void disp()**

**{**

**cout<< area<< endl;**

**}**

**};**

**void main()**

**{**

**cons o;**

**cons o2( 10, 20);**

**o.disp();**

**o2.disp();**

**}**

**DYNAMIC INITIALIZATION OF OBJECTS**

* Dynamic initialization of object refers to initializing the objects at run time i.e. the initial value of an object is to be provided during run time
* Dynamic initialization can be achieved using constructors and passing parameters values to the constructors.
* This type of initialization is required to initialize the class variables during run time.
* Dynamic initialization of objects is needed as
  + - It utilizes memory efficiently.
    - Various initialization formats can be provided using overloaded constructors.
    - It has the flexibility of using different formats of data at run time considering the situation.

Example program:-

#include <iostream.h>

class simple\_interest

{

   float principle , time, rate ,interest;

   public:

      simple\_interest (float a, float b, float c)

{

         principle = a;

         time =b;

         rate = c;

      }

      void display ( )

{

         interest =(principle\* rate\* time)/100;

         cout<<"interest ="<<interest ;

      }

};

void main()

{

   float p,r,t;

   cout<<"principle amount, time and rate"<<endl;

   cout<<"2000 7.5 2"<<endl;

   simple\_interest s1(2000,7.5,2);//dynamic initialization

   s1.display();

   }

**COPY CONSTRUCTOR:-**

* A copy constructor is a member function which initializes an object using another object of the same class.

sample i2=i1;

* The process of initializing through a copy constructor is known as copy initialization.
* A copy constructor takes a reference to an object of the same class as itself as an arguments.

EXAMPLE PROGRAM:-

#include<iostream.h>

class code

{

int id;

public:

code

{

}

code(int a)

{

id=a;

}

void display()

{

cout<<id;

}

};

void main()

{

code a(100);

code b=a;

cout<<”\n id of a: “; a.display();

cout<<”\n id of b: “; b.display();

}

**DESTRUCTORS**

* A destructor as the name implies, is used to destroy the objects that have been created by a constructor.
* The destructor is a member function whose name is the same as the class name but is preceded by a tilde.

~simple()

* A destructor never takes any argument nor does it return any value.

#include<iostream.h>

int count=0;

class test

{

public:

test()

{

count++;

cout<<”\n\n constructor msg: object number “<<count<<”created..”;

}

~test()

{

cout<<”\n\n destructor msg: object number “<<count<<”destroyed..”;

count--;

}

};

void main()

{

cout<<”inside the main block…”;

cout<<”\n\n creating first object t1..”;

test t1;

{

cout<<”\n\n inside block 1…”;

cout<<”\n\n creating two more objects t2 and t3..”;

test t2,t3;

cout<<”\n\n leaving block 1…”;

}

cout<<”\n\n back inside the main block..”;

}

**OPERATOR OVERLOADING AND TYPE CONVERSIONS**

* The mechanism of giving such special meanings to an operator is known as operator overloading.
* Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators.
* We can overload all the C++ operators except following.
  + - Class member access operators(.\*)
    - Scope resolution operator(::)
    - Size operator(sizeof)
    - Conditional operator(?:)

**DEFINING OPERATOR OVERLOADING**

* Define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied.
* This is done with the help of a special function called operator function, which describes the task.
* The general form of an operator function is:

**returntype classname :: operator op(arglist)**

**{**

**function body// task defined**

**}**

* Where return type is the type of value returned by the specified operation and op is the operator being overloaded.
* Operator op is the function name, where operator is a keyword.

RULES FOR OVERLOADING OPERATORS

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
4. There are some operators that cannot be overloaded.
5. We cannot use **friend** functions to overload certain operators. However,member functions can be used to overload them.

**TYPE CONVERSIONS**

* C applies automatic type conversion to the operands as per certain rules.
* Similarly an assignment operation also causes the automatic type conversion.
* The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left.

int m;

float x=3.14159;

m=x;

* convert **x** to an integer before its value is assigned to **m**.
* Three types of situations might arise in the data conversion between uncompatible types.

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.