OBJECT ORIENTED PROGRAMMING

Branch: S6 ECE

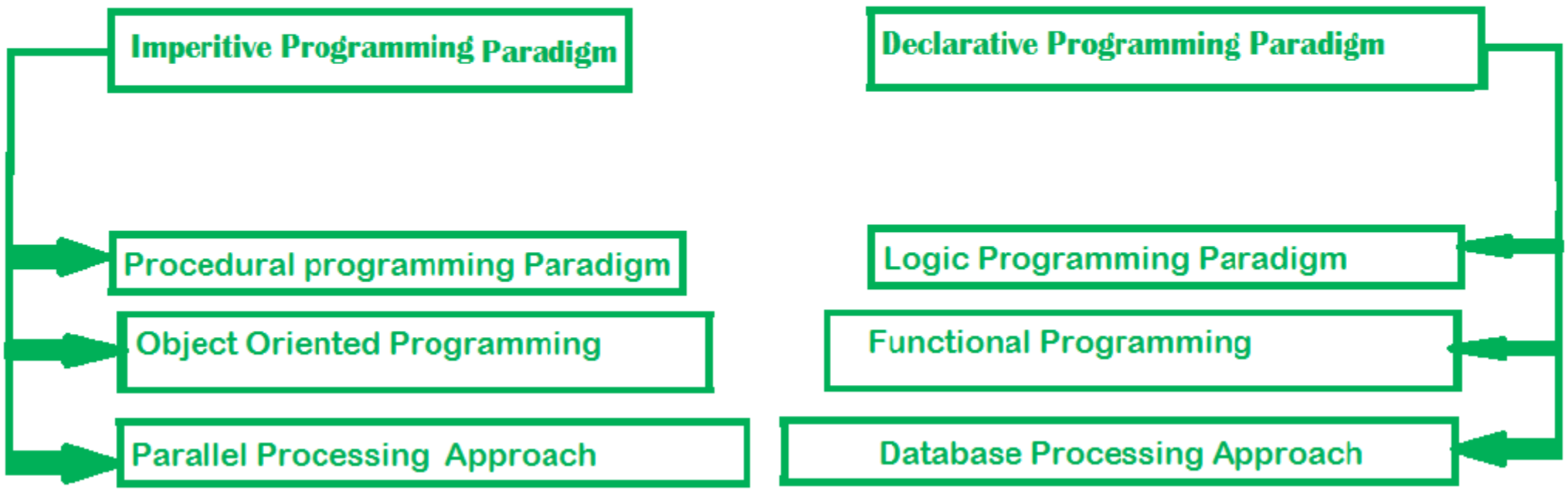
Faculty: SREEDIVYA I

PRINCIPLES OF OOP

PROGRAMMING PARADIGMS

- Paradigm-method to solve a problem
- Programming Paradigm- method to solve problem using some programming language

Programming Paradigms



A. Imperative Programming Paradigm

- Oldest programming paradigm
- Its features are closely related to machine language
- Based on Von Neumann architecture
- Assignment statements are used for changing program state
- Performs step by step task by changing state
- Three categories:

1. Procedural Programming
2. Object oriented Programming
3. Parallel Processing Approach

Advantages

- Very simple to implement
- It contains loops, variables etc.

Disadvantages

- Complex problem cannot be solved
- Less efficient and less productive
- Parallel programming is not possible

1.Procedural Programming Paradigm

- Emphasize on procedures i.e., how to do a particular task rather than what to do
- Ability to reuse the code
- Examples: C, Fortran, Pascal

2.Object Oriented Programming

- Consists of classes and objects
- More emphasis on data
- Able to solve all real world problems
- Examples: C++, Java, Python

Advantages

- Data security
- Inheritance
- Code reusability
- Flexible and abstraction is also present

3.Parallel Processing Approach

- Parallel processing is the processing of program instructions by dividing them among multiple processors.
- Objective - running a program in less time by dividing them
- Example - NESL

B. Declarative Programming Paradigm

- It expresses logic of computation without mentioning its control flow
- Considers programs as theories of some logic
- Emphasis is on what to do and just declares the result
- Three categories

1. Logic
2. Functional
3. Database Processing

1.Logic Programming

- abstract model of computation
- solve logical problems like puzzles, series
- Here we have a knowledge base which we know before and along with the question and knowledge base which is given to machine, it produces result.
- Emphasize is on knowledge base and the problem
- The execution of the program is very much like proof of mathematical statement
- Example: Prolog

2.Functional Programming

- The key principle is the execution of series of mathematical functions
- Functions hide the implementation details
- Function can be replaced with their values without changing the meaning of the program
- Example: Perl, JavaScript

3.Database Programming

- Based on data and its movement
- Program statements are defined by data
- Provides file creation, data entry, update, query and reporting functions
- Example: SQL

Difference between POP and OOP

Procedure Oriented Programming	Object Oriented Programming
Follows Top down approach	Follows Bottom up approach
Programs are divided into small parts called functions	Programs are divided into small parts called objects
Data is less secure	Data is highly secured
Deals with algorithm	Deals with data
Very less memory is required	Memory requirement is more than POP
No access specifier	Has Access specifier(public,private,protected)
Cannot perform overloading	Can perform overloading
Data hiding is not possible	Data hiding is possible
Examples: C, Fortran, Cobol	Examples: C++, Java, Python

Basic Concepts of OOP

Branch: S6 EC

Faculty: Sreedivya I

- Objects
- Classes
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message Passing

Objects

- Basic runtime entities in an object oriented system
- They may represent a person, a bank account , a table of data or any item that the program has to handle
- They represent a physical quantity
- During the execution of a program the objects interact by sending messages to each other
- Each object contains data and code to manipulate the data
- Example: Students in a classroom, Employees in a bank

Classes

- Collection of objects of similar type.
- User defined datatypes
- Class is a logical entity
- Objects are variables of the type class.
- Any number of objects can be created for a class.
- Example: Mango, Apple, Orange are members of class FRUITS

  Car, Bus, Train are members of class VEHICLE

Abstraction

- It refers to the act of representing essential features without including the background details or explanations
- That means only necessary details are exposed
- Example: When a key is pressed on the keyboard, the character appears on the screen, but the exact working may not be known to the user

Encapsulation

- Wrapping up of data and functions into a single unit
- Data is not accessible to the outside world
- The functions which are wrapped inside the class can access those data
- Data is made hidden
- Class is the best example of Encapsulation
- Example: To buy a medicine we go to medical store and ask chemist for the medicine. Only chemist has access to medicines. It reduces the risk of you taking any medicine that is not intended for you.

Inheritance

- It is the process by which objects of one class (derived class)acquire the properties of objects of another class(base class)
- Derived class has some common properties of the parent class
- Code reusability –additional features can be added to an existing class without modifying it
- Saves a lot of time in programming and reduce errors
- Increases the quality of work

- Example: Family tree

The features of children are inherited from Parents and grandparents. Similarly the parents inherit the properties from their parents. (Grandparents ,parents ,children)

- Types of Inheritance – Single, Hierarchical, Multiple, Multilevel and Hybrid

Polymorphism

- Ability to take more than one form
- An operation may exhibit different behaviors at different instances
- The behavior depends upon the types of data used in the operation
- The process of making a **single operator** to exhibit different behaviors on different instances is known as **operator overloading**
- The process of using a **single function name** to perform different types of tasks is known as **function overloading**
- **Overriding**- Both the parent class and derived class have the **same method name and arguments** but their functionality is different

- Example:
    1. Addition – If the operands are numbers then the result will be sum.
       If the operands are strings then the result is concatenation
       (Operator overloading)

    2. Drawing – If the problem is to draw then,
       for drawing a circle, function draw(int r) is used
       for drawing a rectangle, function draw(int l, int b) is used
       (function will be same, arguments will be different)

Dynamic Binding

- It refers to the linking of a procedure call to the code to be executed in response to the call.
- Also known as late binding as the code associated with a given procedure call is not known until the time of call at runtime

Message Passing

- Objects communicate with each other through message passing
- Objects can send or receive information
- Basic steps:

1. Creating classes that define objects and their behavior
2. Creating objects from class definition
3. Establishing communication between objects

Benefits of OOP

- Users can create new datatype by making class
- Code Reusability through Inheritance
- Data hiding which ensures data security through encapsulation
- Same functions or operators can be used for multitasking by overloading (Polymorphism)
- Can be used to build large or complex systems
- Message passing techniques makes communication easier
- It is easy to partition work for same project
- Software complexity can be easily managed

Application areas of OOP

- Real time systems
- Simulation and modelling
- Object oriented database
- Neural network and parallel programming
- Office automation system
- Client server system
- AI Expert system

Introduction to C++

Branch: S6 ECE

Faculty: Sreedivya I

History of C++

- Developed by Bjarne Stroustrup at Bell Laboratories in the early 1980’s
- It was standardized in 1998 by ANSI and ISO
- It was an extension to C
- It was called as C with classes

Features of C++

- General purpose programming language
- Compiled language
- Portable and machine independent
- Powerful and extensible
- Supports dynamic memory allocation
- Case sensitive
- Supports pointers
- Strongly typed language

Sample C++Program

```
#include<iostream>
using namespace std;
void main()
{
   cout<<“C++ is an object oriented programming language”;
}
```

Explanation of the program:

iostream – This is a header file which add the content of iostream file to the program. It should be added at the beginning of every program that use i/o statements

namespace – defines a scope for the identifiers. For using identifiers defined in the namespace scope it must include ‘using’ directive. Here ‘std’ is the namespace used where standard class libraries are defined

main() – every program has only one main() and the execution of a program starts from here

Tokens

Tokens are the smallest individual units in a program

- Keywords –reserved words(eg: int,class)
- Identifiers-variables or functions(eg: sum)
- Constants – variables with fixed value(eg: pi)
- Strings – characters(eg: “ABC”)
- Operators – used for performing some operations(eg: +,-,<<,>>)

Basic Datatypes

- Built in datatypes – int,char,float,double,void
- User Defined datatypes - class
- Derived datatypes – Arrays, Functions, Pointers

Operators

- Input Operator (>>)– extraction/get from operator.

  extracts value from keyboard and assigns it to the variable on the right

- Output Operator(<<)- insertion/put to operator

  inserts the contents of variable on its right to the objects on its left

Cascading of I/O operators

- Multiple usage of I/O operators in a single program statement.
- Eg: cout<<“Sum=“<<sum<<“\n”;

TYPE COMPATIBILTY(or Type Casting)

- Conversion from one datatype to another (like int to float, double to int etc)
- Two types

1. Implicit type casting(automatically done)
2. Explicit type casting(done by user)

Class

- A class in OOP means that it is a way to bind the data and functions together.
- A class specification has two parts
  1) Class declaration (describes the type & scope of its members)
  2) Class function definition (describes how class functions are implemented)

General form of class declaration

```
class class_name
{
 private:
    variable declarations;
    function declarations;
 public:
    variable declarations;
    function declarations;
};
```

Member functions can be defined in two places:
1) Outside the class definition
2) Inside the class definition (There will be no class declaration here)

Characteristics of member functions:

1) Several different classes can use the same function name. The membership label will resolve their scope.
2) Member functions can access the private data of the class.
3) A member function can call another member function directly without using the dot operator.

Private member function

A private member function can only be called by another function that is a member of its class.

Example

```
class sample
{
 int m;
 void read (void);          // private member function
 public:
    void update (void);
};
void sample :: update (void)
{ read ();                  // calling private member function by the public member function of same class
```

Example for class definition made inside the class

```
#include <iostream>
using namespace std
class person
{
  char name[30];
  int age;
  void getdata (void)
  {
    cout << " Enter name : " <<"\n";
    cin >> name;
    cout << " Enter age : " <<"\n";
    cin >> age;
  }
  void display (void)
  {
    cout <<"\n Name : " <<name;
    cout <<"\n Age : " << age;
  }
};
void main()
{
  person p;
  p.getdata();
  p.dispdata
  p.display();
}
```

Output

```
Enter name : JOHN
Enter age : 25

Name : JOHN
Age : 25
```

Example for class definition made outside the class

```
#include <iostream>
using namespace std;
class person
{
  char name[30];
  int age;
  public:
        void getdata(void);
        void display(void);
};
void person::getdata(void)
{
  cout << "Enter name: " << "\n";
  cin >> name;
  cout << "Enter age: " << "\n";
  cin >> age;
}
void person::display(void)
{
  cout << "\n Name: " << name;
  cout << "\n Age : " << age;
}
void main()
{
  person p;
  p.getdata();
  p.display();
}
```

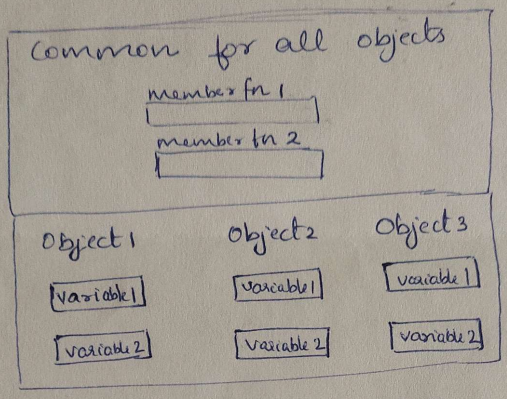
Output

```
Enter name : JOHN
Enter age : 25

Name : JOHN
Age  : 25
```

Memory allocation for objects

- Member functions are created and placed in the memory space only once when they are defined as part of class specification and during object creation.
- Only space for member variables are allocated separately for each object. This is essential because variables will hold different data values for different objects.



Static data members

A data member which is declared as 'static' has the following special characteristics:

1) It is initialised to zero when the first object of its class is created. No other initialization is permitted.
2) Only one copy of that member is created for the entire class and is shared by all the objects of that class.
3) It is visible only within the class but its lifetime is the entire program.

Static Member function

A member function which is declared as 'static' has the following properties:

1) A static function can have access to only other static members (variables/functions) declared in the same class.
2) A static member function can be called using classname

Syntax: classname :: function_name ;

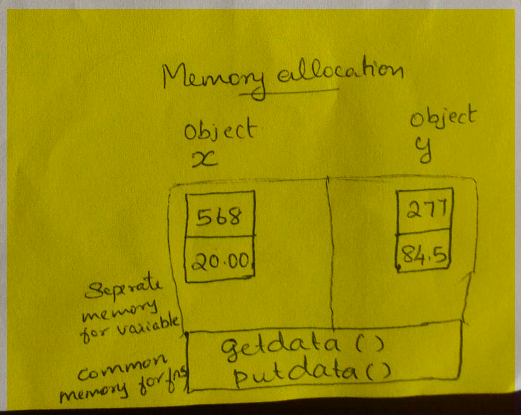
Example for memory allocation of objects

```
#include <iostream>
using namespace std;
Class item
{
  int number;
  float cost;
  public:
      void getdata (int a, float b);
      void putdata (void);
};
void item :: getdata (int a, float b)
  {
    number = a;
    cost = b;
  }
Void item :: putdata (void)
{
  cout << " number : " << number << "\n";
  cout << " cost : " << cost << "\n";
}
void main ()
{
  item x, y;
  cout << "\n Pencil " << "\n";
  x.getdata (568, 20.00);
  x.putdata();
  cout << "\n Pen " << "\n";
  y.getdata (277, 84.50);
  y.putdata ();
}
```

Output

Pencil
number : 568
cost : 20.00

Pen
number : 277
cost : 84.50



Scope Resolution Operator

Branch: S6 ECE

Faculty: SREEDIVYA

Local and global variable

- The scope of the variable extends from the point of its declaration till the end of the block containing the declaration.
- **Local** variable- a variable declared inside the block
- **Global** variable – a variable declared outside the block

SCOPE RESOLUTION OPERATOR

- **Scope resolution operator** is used to uncover a hidden variable
- This operator allows access to a **global** variable when there is a local variable with the same name
- Used to define a function outside a class
- It can be used to access **static members** when there is a local variable with same name
- Used in the case of multiple inheritance that is, if same variable name exists in the ancestor classes

Syntax:

::variable_name;

Simple example showing the usage of ::

```
#include<iostream>
using namespace std;
int m=10;

int main()
{
  int m = 20;
        {
                int k=m;
                int m=30;
                cout<<"\nINNER BLOCK\n";
                cout<<"k="<<k<<"\n";
                cout<<"m="<<m<<"\n";
                cout<<"::m="<<::m<<"\n";
        }
  cout<<"\nOUTER BLOCK\n";
  cout<<"m="<<m<<"\n";
  cout<<"::m="<<::m<<"\n";

  return 0;
}
```

OUTPUT

INNER BLOCK

k=20

m=30

::m=10

OUTER BLOCK

m=20

::m=10

1. To access a global variable when there is a local variable with same name

```
#include<iostream>
using namespace std;
int x=5; // Global x

int main()
{
int x = 10; // Local x
cout << "Value of global x is " << ::x;
cout << "\nValue of local x is " << x;
return 0;
}
```

Output

```
Value of global x is 5
Value of local x is 10
```

2. To define a function outside a class

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

class A
{
    public:
       void fun();
};

void A::fun()
{
    cout << "fun() called";
}

int main()
{
    A a;
    a.fun();
    return 0;
}
```

OUTPUT

```
fun() called
```

3. To access a class's static variables

```
#include<iostream>
using namespace std;
class Test
{
    static int x;
public:
    static int y;

    void func(int x)
    {
    cout << "Value of static x is " << Test::x;
    cout << "\nValue of local x is " << x;
    }
};
int Test::x = 1; //In C++, static members are explicitly defined like this
int Test::y = 2;
int main()
{
    Test obj;
    int x = 3 ;
    obj.func(x);
    cout << "\nTest::y = " << Test::y;
    return 0;
}
```

OUTPUT

Value of static x is 1

Value of local x is 3

Test ::y = 2

4. In case of Multiple Inheritance

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

class A
{
protected:
    int x;
public:
    void A()
      {
        x = 10;
      }
};

class B
{
protected:
    int x;
public:
    void B()
    {
      x = 20;
    }
};
```

```
class C: public A, public B
{
public:
void fun()
{
    cout << "A's x is " << A::x;
    cout << "\nB's x is " << B::x;
}
};

int main()
{
    C c;
    c.fun();
    return 0;
}
```

OUTPUT

```
A's x is 10
B's x is 20
```

ARRAY OF OBJECTS

Branch: S6 ECE

Faculty: SREEDIVYA

Example 1:Program to read and display the name and price of items

```
#include<iostream>
using namespace std;
class item
{
  char name[30];
  int price;
  public:
  void getitem();
  void printitem();
};

void item::getitem()
{
  cout << "Item Name = ";
  cin >> name;
  cout << "Price = ";
  cin >> price;
}
```

```
void item ::printitem()
{
  cout << "Name : " << name <<
          "\n";
  cout << "Price : " << price <<
          "\n";
}

const int size = 3;
```

```
int main()
{
  item t[size];
  for(int i = 0; i < size; i++)
  {
    cout << "Item  : " <<(i + 1) << "\n";
    t[i].getitem();
  }

  for(int i = 0; i < size; i++)
  {
    cout << "Item Details : " <<(i + 1) << "\n";
    t[i].printitem();
  }
}
```

Example 2: Program to read and display the details of 5 students and also calculate the total marks obtained

```
#include<iostream>
using namespace std;
class student
{
  char name[30];
  int rollno;
  int total;
  int m1,m2,m3;
  public:
  void getdata();
  void dispdata();
};

void student ::getdata()
{
  cout << “Enter Name : ";
  cin >> name;
  cout << “Enter Rollno: ";
  cin >> rollno;
  cout << “Enter Marks : ";
  cin>>m1>>m2>m3;
  total=m1+m2+m3;

}
```

```
void student ::dispdata()
{
        cout<<"Name : " <<name<<"\n";
        cout<<“Rollno: " <<rollno<<"\n";
        cout<<“Marks: " <<m1<<m2<<m3<<"\n";
        cout<<“Total: " <<total<<"\n";

}
void main()
{
        student st[5];
        for (i=0;i<5;i++)
        st[i].getdata();
        for (i=0;i<5;i++)
        st[i].dispdata();
}
```

Functions in C++

Branch:S6 ECE

Faculty: SREEDIVYA

Functions

- A function is a block of code that performs some operation
- Dividing a program into functions is one of the major principles of top – down, structured programming.
- It is possible to reduce the size of the program by calling and using them at different places in the program
- C++ has added many new features to functions to make them more reliable and flexible
- C++ functions can be overloaded to make it perform different tasks depending on the arguments passed to it

Structure of a function

```
void display();                    /*Function Declaration*/
void main()
{
        -----
    display();                     /*Function call*/
        --------
}
void display()                     /*Function Definition*/
{
    -----
    -----
}
```

Function Prototyping/Declaration

- Provides the compiler with details such as the number and type of arguments and the type of return values.
- Syntax:

```
type function_name(argument_list);
```

eg: float volume(int x, float y, float z);

Function Definition

- The actual operation of the function is specified in the function definition.
- The arguments inside function definition are known as **formal parameters**
- Formal parameters are just a reference or copy of the actual values

Function call

- It is an expression that passes control and arguments to a function
- The arguments(or variables) inside a function call are known as **actual parameters**
- Actual parameters represent the real values that are to be passed on to the function.
- There are two ways to call a function

1. Call by value
2. Call by reference

Example(Sample program illustrating use of Function)

```
#include <iostream>
using namespace std;
int add(int, int);
int main() {
        int x, y, sum;
        cout << "Enter two numbers: ";
        cin >> x >> y;
        sum = add(x, y);
        cout <<"The sum of "<<x<< " and " <<y<<" is: "<<sum;
        return 0;
}
int add(int num1, int num2) {
        int add;
        add = num1 + num2;
        return add;
}
```

Call by value

- Actual values in the call are passed to the function definition
- Disadvantages: Copying overhead and no alteration can be made

Call by reference

- Instead of passing values, the memory location is passed.

  Function definition be,

```
void swap(int &a, int &b)
{
    int t=a;
    a=b;
    b=t;
}
```

  Function call be, swap(m,n);

Return by reference

- A function can also return a reference
- Function definition be

```
int &max(int &x,int &y)
{
        if(x>y)
          return x;
        else
           return y;
}
```

Inline functions

- These functions are expanded in line when it is invoked. i.e., compiler replaces the function call with the corresponding code.

Syntax:

```
inline function header
{
function body;
}
```

Example:

```
inline double cube(double a)
{
return (a*a*a);
}
This can be invoked by c=cube(3.0);
```

Features of inline functions

- Inline functions must be defined before they are called.
- Functions are made inline when they are small enough to be defined in one or two lines
- Consumes more memory
- Inline functions may not work,

a) If they are recursive
b) If they contain static variables
c) For function returning values; if a loop, a switch or a goto exists.
d) For functions not returning values; if a return statement exists

Default Arguments

- C++ allows to call a function without specifying all its arguments.
- In that case the function assigns a default value to the parameter which does not have a matching argument in the function call.
- Default values are specified when the function is declared.

Eg:

```
int date(int day, int month, int year=2021);
```

Function call: calendar= set(20,12);

calendar= set(20,12,2023)

Function overloading in C++

- Two or more functions can have the same name but different parameters
- The number and type of arguments will be different
- When a function name is overloaded with different jobs it is called function overloading
- Function overloading can be considered as an example of polymorphism in C++
- This improves the readability of the program

Example 1

```
#include<iostream>
using namespace std;
class A
{
        int num1=20,num2=10;
        public:
                void fun()
                        {
                            int sum=num1+num2;
                            cout<<“Addition   ”<<sum<<“\n”;
                        }
```

```
        void fun(int a, int b)
    {
        int sub=a-b;
        cout<<“Subtraction   ”<<sub<<“\n”;
        }
};
int main()
{
   A obj;
   obj.fun();
   obj.fun(100,50);
   return 0;
}
```

OUTPUT

Addition 30

Subtraction 50

Example 2

```
#include<iostream>
using namespace std;
class Addition
{
   public:
                    int sum(int a, int b)
                    {
                       return (a+b);
                    }
```

```
                    int sum(int a, int b, int c)
                    {
                       return (a+b+c);
                    }
};
int main()
{
  Addition obj;
  cout<<obj.sum(20,15)<<"\n";
  cout<<obj.sum(81,100,10);
  return 0;
}
```

OUTPUT

35

191

Pointers in C++

Branch: S6 ECE

Faculty: SREEDIVYA

Pointers

- A pointer is a variable that holds the memory address of another variable of same type
- Derived data type
- A pointer that stores the address of some variable x is said to point to x.
- A pointer is declared by putting a star (or '*') before the variable name.
- To access the value of x, we need to dereference the pointer.

Reference Variable

- A reference is an alias(another name) for another variable.

- In C++, the address of a variable is obtained using **reference operator '&'(or address operator)**

Syntax

```
type &reference_name=variable_name;
```

Dereference operator/Indirection operator(*)

-It is possible to obtain the value of a variable directly from a pointer variable

-The expression *pointer_variable gives the value of the variable

-This evaluation is called dereferencing the pointer.

Need for pointers

- To return more than one value from a function.
- To pass arrays and strings more conveniently from one function to another.
- To manipulate arrays more easily by moving pointers to them.
- To create complex data structures, such as linked lists and binary trees
- For dynamic memory allocation and de-allocation

Sample program using pointers Example1

```
#include<iostream>
using namespace std;
void main()
{
        int a,*ptr;          //declaration of variable a and pointer variable ptr
        ptr=&a;              //initialization of ptr
        cout<<“The address of a:”<<ptr<<“\n”;
}
```

OUTPUT

The address of a: 0x8fb6fff4

Example 2

```
#include<iostream>
using namespace std;
int main()
{
 int n=44,*rn;
 int &rn=n;
 cout<<“Value of variable n=“<<n<<endl;
 cout<<“Value of pointer variable rn=“<<&rn;
}
```

OUTPUT

```
Value of variable n=44
Value of pointer variable rn=0x8fb6fff4
```

Pointer Expressions and Pointer Arithmetic

- A pointer can be incremented(++) or decremented(--)

  int ptr++; or int ++ptr;

  int ptr--; or int - -ptr;
- Any integer can be added or subtracted from a pointer
- One pointer can be subtracted from another

Example

```
#include<iostream>
using namespace std;
void main()
{
        int num[]={56,75,22};
        int *ptr,i;
        cout<<“Array values: “<<“\n”;
```

```
        for(i=0;i<3;i++)
                cout<<num[i]<<"\n";
        ptr=num;
        cout<<"Value of ptr: "<<*ptr<<"\n";
        ptr++;
        cout<<"Value of ptr++: "<<*ptr<<"\n";
        ptr--;
        cout<<"Value of ptr--: "<<*ptr<<"\n";
        ptr=ptr+2;
        cout<<"Value of ptr+2: "<<*ptr<<"\n";
        ptr=ptr-1;
        cout<<"Value of ptr-1: "<<*ptr<<"\n";
}
```

OUTPUT

Array values are:

56

75

22

Value of ptr: 56

Value of ptr++:75

Value of ptr--:56

Value of ptr+2:22

Value of ptr-1:75

Pointers with Arrays and Strings

Differences between pointers and arrays

- Arrays refer to a block of memory space whereas pointers do not refer to any section of memory
- Memory address of array cannot be changed whereas content of pointer variable such as memory addresses can be changed.
- We can declare the pointers to arrays as follows:

```
int *nptr;
nptr=number[0];
```

Program to print the sum of even numbers using pointers to arrays

```
#include<iostream>
using namespace std;
void main()
{
        int numbers[50],*ptr,i,n;
        cout<<“Enter the count: “;
        cin>>n;
        cout<<“\nEnter the numbers ”<<“\n”;
        for(i=0;i<n;i++)
        cin>>numbers[i];
```

```
        ptr=numbers;
        int sum=0;
        for(i=0;i<n;i++)
        {
                if(*ptr%2==0)
                        sum=sum+*ptr;
                ptr++;
        }
        cout<<“\nSum of even numbers : ”<<sum;
}
```

OUTPUT

Enter the count : 5

Enter the numbers

10

16

23

45

34

Sum of even numbers: 60

Array of Pointers

- Represents a collection of addresses
- Save a substantial amount of memory space
- Array of pointers point to an array of data items
- Each element of the pointer array points to an item of the data array
- Data items can be accessed either directly or by dereferencing the elements of pointer array

Example

```
int *inarray[10];
```

Example

```
#include<iostream>
using namespace std;
void main()
{
    int *p = new int[5];
    for (int i = 0; i < 5; i++)
            p[i] = 10 * (i + 1);
    cout << *p << endl;
    cout << *p + 1 << endl;
    cout << *(p + 1) << endl; // similar to p++
    cout << p[2] << endl;
}
```

OUTPUT

10

11

20

30

Pointers to objects

- A pointer can point to an object created by a class.

  eg: item x; // *item* is the class and *x* is the object

  Then,

  item *it_ptr; // *it_ptr* is pointer of type *item*

- Object pointers are useful in creating objects at run time
- Object pointers can be used to access public members of an object

Example

```
#include<iostream>
using namespace std;
class item
{
        int code;
        float price;
 public:
        void getdata( int a, float b)
        {
                code = a;
                price = b;
        }
        void show(void)
        {
                cout<<“Code: ”<<code<<“\n”;
                cout<<“Price: ”<<price<<“\n”;
        }
};
const int size = 2;
```

```
int main()
{
        item *p = new item[size];
        item *d = p;
        int i,x;
        float y;
        for(i=0;i<size;i++)
        {
                cout<<“Enter code and price for item  “ <<i+1;
                cin>>x>>y;
                p->getdata(x,y);
                p++;
        }
```

```
                for(i=0;i<size;i++)
                {
                        cout<<“Item: “<<i+1<<“\n”;
                        d->show();
                        d++;
                }
        return 0;
}
```

OUTPUT

```
Enter code and price for item1 40 500
Enter code and price for item2 50 600
Item: 1
Code: 40
Price: 500
Item: 2
Code: 50
Price: 600
```

this pointer

- C++ uses a unique keyword called **this** to represent an object that invokes a member function
- **this** is a pointer that points to the object for which this function was called.
- **this** pointer is automatically passed to a member function when it is called
- **this** pointer acts as an implicit argument to all the member functions

eg:

```
class ABC
{
        int a;
        - -- -
        ---- -
};
```

The private variable ‘a’ can be used directly inside a member function like,

```
a=123;
```

or

```
this->a = 123;
```

Used in:

- Overloading the operators using member function
- Returning the object it points to (eg: return *this;)
- When local variable's name is same as member's name

Example (name of local variable and member name are same)

```
#include<iostream>
#include<string>
using namespace std;
class Test
{
  int x;
```

```
public:
        void setX (int x)
        {
           this->x = x;
        }
        void print()
         {
           cout << "x = " << x << endl;
        }
};
```

```
int main()
{
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
  return 0;
}
```

OUTPUT

x =20

Dynamic Allocation operators

- Dynamic memory allocation is the process of allocating memory during runtime
- C++ defines two unary operators '**new**' and '**delete**' for this.
- An object can be created using 'new' and destroyed using 'delete' operator
- So a data object created inside a program using 'new' will remain in existence until it is explicitly destroyed by using 'delete'
- If sufficient memory is not available for allocation, 'new' operator returns a null pointer

Dynamic objects

- For object creation using ‘new’, syntax will be,

  datatype pointer_variable = new datatype;

  The **new** operator allocates sufficient **memory** to hold a data object of that datatype and returns the address of the object . The **pointer_variable** holds the **address** of the memory space allocated.

  eg: int *p = new int;

  float *q = new float;

- We can also initialize a memory using new operator

  Syntax is:

  pointer_variable = new datatype(value);

  Example:

  int *p = new int(25);

- For arrays,

  Syntax is:

  pointer_variable = new datatype[size];

  Example:

  int *p= new int[10];

- When a data object is no longer needed it is destroyed to release the memory space for reuse and it is done using '**delete**' operator.

Syntax:

```
delete pointer_variable;
delete [size] pointer_variable;      //For array
```

Example:

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

where p and are pointer variables

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

Advantages of new over malloc

- Automatically computes the size of data objects. No need to use the operator sizeof()
- Automatically returns the correct pointer type. No need to use typecasting
- It is possible to initialize the object while creating the memory space
- Both ‘new’ and ‘delete’ operator can be overloaded.

Dynamic initialization of variables

→ Initialisation of variables at run time

→ In C++, a variable can be initialised at run time using expressions at the place of declaration.

eg:-
```
int n = strlen(string);

float area = 3.14159 * rad * rad;
```

→ Thus both declaration and initialization of a variable can be done simultaneously at the place where the variable is used for the first time.

→ Dynamic initialization is extensively used in OOP.

→ We can create exactly the type of object needed, using information that is known only at the run time.

Dynamic initialization of Objects

→ Objects of a class can be initialized dynamically.

→ That is, the initial value of an object may be provided during run time.

→ Advantage: We can provide various initialisation formats using overloaded constructors.

eg:-
```
class deposit
{
 public:
 float fixed_deposit (int p, int y, float r = 0.12);
};
int main()
{
 deposit D;
 int p, y;
 cout << " Enter amount and year: ";
 cin >> p >> y;
 D.fixed_deposit (p, y, r);
 return 0;
}
```

Const Arguments

→ In C++, an argument to a function can be declared as const.

eg:- int strlen (const char *p);
int length (const string &s);

→ The qualifier 'const' tells the compiler that the function should not modify the argument.

→ It will generate error otherwise.

→ This type of declaration is significant only when we pass arguments by references and pointers.

Const Member functions

→ If a member function does not alter any data in the class, then we may declare it as a 'const' member function

eg:- void mul (int, int) const ;
double get_balance() const ;

→ The qualifier 'const' is appended to the function prototypes (in both declaration and definition).

→ If we try to alter values, then compiler will generate errors.

const Objects

→ It is possible to create and use constant objects using 'const' keyword before object declaration.

for example, let the class be 'matrix' and constant object be 'X' then, it is as shown below,

const matrix X(m,n); // Here object X is constant

→ Any attempt to modify the values of 'm' and 'n' will generate compile-time error.

→ Also, a constant object can call only 'const' member functions.

→ Whenever 'const' objects try to invoke non-const member functions, the compiler generates errors.

1. List out the basic concepts in OOP (2)
2. List a few areas of application of OOP technology (2)
3. What is the output of this program? (3)

```
class Test
{
  int x;
};

int main()
{
  Test t;
  cout << t.x;
  return 0;
}
```

(a) 0
(b) Garbage value
(c) Runtime error
(d) Compiler error

4. What is the correct syntax of accessing a static member of a class in C++? (3)

```
class A
{
        public:
                static int value;
}
```

(a) A->value
(b) A^value
(c) A.value
(d) A::value

4. Write a C++ program using classes to implement basic arithmetic functions calculator (5)
5. Explain the use of Scope Resolution Operator in detail with examples (10)

Question Paper (MINOR) for OBJECT ORIENTED PROGRAMMING S6 ECE

Total Marks: 20 Total Questions: 7 Time: 45 Minutes

Note: For optional questions, you have to answer only one.

1. Fill in the blanks:

   (a) C++ is a case _____ language. _____ refer to the names of variables, functions, arrays, classes
   (b) _____ datatypes are those that are not composed of other datatypes
   (c) A this pointer refers to ____ that currently invokes a member function (4)

2. What is the output of the following program?

```
#include <iostream>
using namespace std;
void main()
{
        int a = 1, b=2;
        cout << (a++ + b);
}
```

   (a) 3 (b) 4 (c) Syntax error (d) 2 (2)

3. Explain the following concepts with example:

   (a) Class
   (b) Default Arguments
   (c) Polymorphism
   (d) Inline functions (8)

4. Write a C++ program to convert the temperature from Celsius to Fahrenheit using class (C to F is (temperature * 1.8 +32))

(OR)

5. Write a C++ program showing the basic arithmetic functions of a calculator using class (3)

6. Write a C++ program showing the concept of pointers to objects

(OR)

7. Write a C++ program showing the concept of array of objects (3)

Question Paper (MINOR) for OBJECT ORIENTED PROGRAMMING S6 ECE

Total Marks: 20 Total Questions: 8 Time: 1 hour

Note: For optional questions, you have to answer only one.

1. By default members of a class are

   (a) public (b) private (c) virtual (d) static (2)

2. How many objects can be created for a class?

   (a) 1 (b) 2 (c) 3 (d) any (2)

3. Write a C++ program to print the message “C++ is an extension to C” (2)

4. List any 2 application areas of OOP (1)

5. Explain new and delete operators. What happens if sufficient memory is not available for allocation? (3)

6. What are objects? How are they created? (2)

7. (a) What are the applications of scope resolution operator in C++? (2)

(b) Explain any 2 scenarios in which scope resolution operator is used with the help of programming examples (6)

(OR)

8. Explain the following OOP concepts with an example.

(a) Abstraction (b) Encapsulation (c) Polymorphism (d) Inheritance (8)