https://shikshamentor.com/object-oriented-programming-using-c-for-msbte-3k-scheme/

312304 - OOP Using C++ (Sem III)

As per MSBTE's K Scheme

CO / CM / IF

Unit 4:Pointers and Polymorphism in C++ Mark - 16

Sr.No	Questions	Marks	Year
1.	**What are the rules of virtual function(Write any two)**	**2**	**S-24**
	State any four rules for virtual functions.	**4**	**S-23**
	1. Function Prototype Matching: The prototype (including return type, name, and parameter list) of a virtual function must be identical in the base class and all derived classes that override it. This ensures that the compiler can identify the function correctly even when using a base class pointer or reference. 2. Access through Base Class Pointer or Reference: Virtual functions are typically accessed through pointers or references of the base class type. This allows for runtime polymorphism, where the actual function called depends on the object's dynamic type at runtime, not the pointer or reference type. 3. Declaration: Virtual functions are declared using the virtual keyword in the base class. 4. Member Function: Virtual functions must be member functions of a class. They cannot be static members. 5. Access through Pointers or References: Virtual functions are typically accessed through pointers or references of the base class type. This enables runtime polymorphism, where the specific function to be called is determined at runtime based on the actual object type. 6. Optional Override: A derived class can optionally choose to override (redefine) the behavior of an inherited virtual function. This allows for specialization of the function in the derived class.	1M each (Any 2 correct Rules)	

	7. Base Class Definition: A virtual function must be defined (have a body) in the base class, even if it's not overridden in any derived classes. 8. Friend Functions: Virtual functions can be declared as friend functions of another class. This allows the friend function to access the virtual function directly, even if it's called through a base class pointer. 9. No Virtual Constructors: Class cannot have virtual constructors, but can have virtual destructors.		
2.	**Explain virtual function with suitable example.Give the rules of virtual function.**	**4**	**S-24**

	Virtual Function:
	A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.
	Example:
	`#include<iostream>`
	`using namespace std;`
	`class base`
	`{`
	`public:`
	` virtual void print()`
	`{`
	`cout << "print base class\n";`
	`}`
	` void show()`
	`{`
	`cout << "show base class\n";`
	`}`
	`};`
	`class derived : public base`
	`{`
	`public:`
	` void print()`
	`{`
	`cout << "print derived class\n";`
	`}`
	` void show()`
	`{`
	`cout << "show derived class\n";`
	`}`
	`};`
	`int main()`
	`{`
	` base* bptr;`
	` derived d;`		

```
 bptr = &d;
 bptr->print();
 bptr->show();
}
```

Rules for virtual functions:
1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class it is incremented or decremented only relative to its base type.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class.

3.	**Describe function overloading and function overriding with suitable example**	**4**	**S-24**
	Describe Function overloading with suitable program.(Repeat)	**6**	**W-22**

Function Overloading

Function overloading allows multiple functions to have the same name within the same scope, as long as their parameter lists are different. This is a form of compile-time polymorphism.

```
#include <iostream>
using namespace std;
void add(int a, int b)
 {
   cout << "Sum of integers: " << a + b << endl;
}
void add(double a, double b) {
   cout << "Sum of doubles: " << a + b << endl;
}
int main() {
   add(2, 3); // Calls the first add function
   add(2.5, 3.5); // Calls the second add function
   return 0;
}
```

Function Overriding:

Function overriding occurs when a derived class redefines a function that is already defined in its base class. The function signature (name and parameters) must be identical in both classes. This is a form of runtime polymorphism.

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual void display() {
        cout << "Base class display" << endl;
    }
};
class Derived : public Base {
public:
    void display() override {
        cout << "Derived class display" << endl;
    }
};
int main() {
    Base *basePtr = new Derived();
    basePtr->display(); // Calls the derived class's display
    return 0;
}
```

4.	**Develop a program to declare class book containing data members as title, authorname, publication,price.Accept and display the information for one object using pointer to that objects**	**6**	**S-24**

```
#include<iostream>
using namespace std;
class book
{
```

```
char title[30];
char authorname[30];
char publication[30];
int price;
public:
void accept()
{
cout<<"\n Enter the title of Book:";
cin>>title;
cout<<"\n Enter the Book Author Name:";
cin>>authorname;
cout<<"\n Enter the Publication details:";
cin>>publication;
cout<<"\n Enter the Price of Book:";
cin>>price;
}
void display()
{
cout<<"\n Title of the Book:"<<title;
cout<<"\n The Name of Author is:"<<authorname;
cout<<"\n The Publication company is:"<<publication;
cout<<"\n The Price of the Book is:"<<price;
}
};
int main()
{
book b,*p;
p=&b;
p->accept();
p->display();
return 0;
}
```

5.	**Write a C++ program to overload "+" operator so that it will perform concatenation of two strings.(Use class getdata function to accept two strings)**	**6**	**S-24**

```
#include<iostream>
#include<string.h>
using namespace std;
class String
{
 char str[20]; //member variable for string input
public:
 void input( ) //member function
 {
cout<<"Enter String: ";
cin>>str;
 }
 void display( ) //member function for output
 {
cout<<"Concatenation of two Strings: "<<str;
 }
 String operator+(String s) //operator overloading
 {
String obj;
strcat(str,s.str);
strcpy(obj.str,str);
return obj;
 }
};
int main( )
{
 String str1,str2,str3; //creating three object
 str1.input( );
str2.input( );
 str3=str1+str2;
 str3.display( ); //displaying
}
```

OR

```
#include<iostream>
#include<string.h>
using namespace std;
class string_concat
{
char s1[30];
char s2[30];
public:
void getdata( )
{
cout<<"\n Enter the First String :";
cin>>s1;
cout<<"\n Enter the Second String :";
cin>>s2;
}
void operator+( )
{
cout<<"\n The concatenation of two String
is:"<<strcat(s1,s2);
}
};
int main( )
{
string_concat s;
s.getdata( );
+s;
}
```

6.	**Explain reference and deference operators with respect to pointers.**	**2**	**W-23**
	Reference and Dereference Operators with Pointers In programming, particularly in languages like C and C++, pointers are used to indirectly access memory locations. The reference and dereference operators are essential for working with pointers.		

	Reference Operator (&)
	
	• **Purpose:** Gets the address of a variable.
	• **Syntax:** &variable_name
	• **Explanation:** When you apply the reference operator to a variable, it returns the memory address where that variable is stored. This address is then typically assigned to a pointer.
	
	Example:
	
	C++
	
	int x = 10;int *ptr = &x; // ptr now holds the address of x
	
	Dereference Operator (*)
	
	• **Purpose:** Accesses the value stored at the memory location pointed to by a pointer.
	• **Syntax:** *pointer_name
	• **Explanation:** When you apply the dereference operator to a pointer, it gives you the value stored at the memory address that the pointer points to.
	
	Example:
	
	C++
	
	int x = 10;int *ptr = &x;int y = *ptr; // y now holds the value of x, which is 10
	In this example, *ptr dereferences the pointer ptr to get the value at the address it points to (which is the value of x).		

7.	**Differentiate between compile time polymorphism and run time polymorphism. (Any four points)**	**4**	**W-23**
	Differentiate between compile time and run time polymorphism.	**4**	**S-22**
	Compile Time Polymorphism (Static Polymorphism) • **Determined at compile time:** The compiler decides which method to call based on the static type of the object. • **Achieved through method overloading:** Multiple methods with the same name but different parameter lists. • **Example:** Overloaded add methods in a class to handle different data types (int, double, etc.). • **Less flexible:** The decision is made before the program runs. **Run Time Polymorphism (Dynamic Polymorphism)**		

	• **Determined at runtime:** The method to be called is decided based on the actual object type at runtime. • **Achieved through method overriding:** A subclass provides a specific implementation of a method inherited from a superclass. • **Example:** A Shape class has a draw() method, and subclasses like Circle and Rectangle override it with their specific implementations. • **More flexible:** Allows for dynamic behavior based on object types.		
8.	**Describe ‘this’ pointer in C++ with an example**	**6**	**W-23**
	Illustrate this pointer with example.	**4**	**S-23**

The this Pointer in C++

The this pointer is a special constant pointer available within the non-static member functions of a class. It points to the object whose member function is being called.

How it works:

When an object calls a non-static member function, the compiler implicitly passes a hidden argument to the function: a pointer to that object itself.

This pointer is stored in the this pointer within the function.

You can use this to access the object's members from within its own member functions.

```
#include <iostream>

class MyClass {

public:

   int x;

   void setX(int value) {

      x = value; // Equivalent to (*this).x = value;

   }

   void print() {

      std::cout << "x: " << x << std::endl;

   }

};
```

	int main() {
	    MyClass obj;
	    obj.setX(10);
	    obj.print();
	    return 0;
	}
	Explanation:
	In the setX function, this->x is equivalent to x. However, using this explicitly can be helpful when there's a name conflict between a member variable and a local variable.
	The print function demonstrates how to access and print the object's data member using this.
	this is a constant pointer, meaning its value cannot be changed.
	this is only available within non-static member functions.
	It can be used to access the object's members and return a reference to the object itself.
	It's often used in constructors, operator overloading, and when there's a name conflict between a member and a local variable.		

9.	**Explain the rules of virtual function.**	**6**	**W-23**
	● Must be a member of a class: Virtual functions cannot exist outside a class. ● Cannot be static: Static members belong to the class, not objects, so they can't be overridden. ● Accessed through object pointers: To achieve runtime polymorphism, virtual functions must be called through a base class pointer pointing to a derived class object. ● Can be a friend of another class: This doesn't affect their virtual behavior. ● Must be defined in the base class: Even if not used, a virtual function must have a definition in the base class. ● Prototype must match: The prototype (return type, name, and parameters) of a virtual function must be identical in both base and derived classes. ● Overriding is optional: Derived classes can choose to override or not. If not		

overridden, the base class version is called.

- Virtual destructors: Destructors can be virtual to ensure proper cleanup of derived class objects.
- No virtual constructors: Constructors are called during object creation, so runtime polymorphism doesn't apply.
- Pointer arithmetic: Incrementing or decrementing a base class pointer pointing to a derived class object is based on the base class size, not the derived class size.

```
#include <iostream>
class Base {
public:
   virtual void display() {
       std::cout << "Base class display" << std::endl;
   }
};
class Derived : public Base {
public:
   void display() override {
       std::cout << "Derived class display" << std::endl;
   }
};
int main() {
   Base *basePtr = new Derived();
   basePtr->display(); // Calls Derived's display() due to virtual function
   delete basePtr;
   return 0;
}
```

10.	**Write a program to declare a class ‘item’ containing data members as ‘item_name’, ‘code’, ‘price’. Accept and display the information for one**	**6**	**W-23**

	object using pointer to that object.		

```
#include <iostream>
#include <string>
using namespace std;
class Item {
public:
    string item_name;
    int code;
    double price;
};
int main() {
    Item item;
    Item *ptr = &item;
    cout << "Enter item name: ";
    getline(cin, ptr->item_name);
    cout << "Enter item code: ";
    cin >> ptr->code;
    cout << "Enter item price: ";
    cin >> ptr->price;
    cout << "\nItem Information:\n";
    cout << "Item Name: " << ptr->item_name << endl;
    cout << "Item Code: " << ptr->code << endl;
    cout << "Item Price: " << ptr->price << endl;
    return 0;
}
```

11.	**Define polymorphism with it's types.**	**2**	**S-23**
	Polymorphism is a core concept in object-oriented programming (OOP) that allows		

	objects of different types to be treated as if they were of the same type. It essentially means "many forms," reflecting the ability of an object to take on multiple forms. Types of Polymorphism 1. Compile-time Polymorphism (Static Polymorphism) 2. Runtime Polymorphism (Dynamic Polymorphism)		
12.	**State the advantages of pointer**	**4**	**S-23**
	Efficiency Direct memory access: Pointers provide direct access to memory locations, which can lead to faster program execution compared to accessing data through variables. Efficient data manipulation: Operations on data through pointers can be performed more efficiently in certain cases. **Flexibility** Dynamic memory allocation: Pointers are essential for allocating memory dynamically at runtime, allowing for flexible memory management based on program needs. **Data structures:** Pointers are fundamental for implementing complex data structures like linked lists, trees, and graphs. **Function parameters:** Pointers can be used to pass arguments by reference, allowing functions to modify the original data. **Low-level programming** **Memory management:** Pointers provide a way to interact with the underlying hardware and memory system, enabling low-level programming tasks. **System programming:** Pointers are crucial for operating system development and other system-level programming.		
13.	**Write C++ program to overload binary operator ‘+’ to concatenate two strings.**	**6**	**S-23**

	#include <iostream>
	
	#include <string>
	
	using namespace std;
	
	class String {		

```
public:
    string str;
    String(const string& s) : str(s) {}
    String operator+(const String& other) {
        return String(str + other.str);
    }
    void display() {
        cout << str << endl;
    }
};
int main() {
    String s1("Hello");
    String s2(" World");
    String s3 = s1 + s2;
    s3.display();
    return 0;
}
```

14.	**Explain concept of pointer with example.**	**2**	**W-22**
	Pointers in C++ A pointer in C++ is a variable that stores the memory address of another variable. It acts like a locator or indicator pointing to the location where data is stored. `int x = 10; int *ptr = &x;`		
15.	**Differentiate between compile time polymorphism and Runtime polymorphism.**	**4**	**W-22**

Compile-Time Polymorphism	Run-Time Polymorphism
It is also called Static Polymorphism.	It is also known as Dynamic Polymorphism.
In compile-time polymorphism, the compiler determines which function or operation to call based on the number, types, and order of arguments.	In run-time polymorphism, the decision of which function to call is determined at runtime based on the actual object type rather than the reference or pointer type.
Function calls are statically binded.	Function calls are dynamically binded.
Compile-time Polymorphism can be exhibited by: 1. Function Overloading 2. Operator Overloading	Run-time Polymorphism can be exhibited by Function Overriding.
Faster execution rate.	Comparatively slower execution rate.
Inheritance in not involved.	Involves inheritance.

16.	**Develop a c++ program to perform arithmetic operation using pointer.**	**4**	**W-22**

```
#include <iostream>
using namespace std;
int main() {
   int num1 = 10, num2 = 20;
   int *ptr1 = &num1, *ptr2 = &num2;
   // Addition
   int sum = *ptr1 + *ptr2;
   cout << "Sum: " << sum << endl;
   // Subtraction
   int difference = *ptr1 - *ptr2;
   cout << "Difference: " << difference << endl;
   // Multiplication
   int product = *ptr1 * *ptr2;
```

```
    cout << "Product: " << product << endl;
    // Division
    if (*ptr2 != 0) {
        double quotient = static_cast<double>(*ptr1) / *ptr2;
        cout << "Quotient: " << quotient << endl;
    } else {
        cout << "Division by zero error!" << endl;
    }
    return 0;
}
```

17.	**Explain rules of operator overloading and overload '+' operator to concatenate two string.**	**6**	**W-22**

Operator Overloading

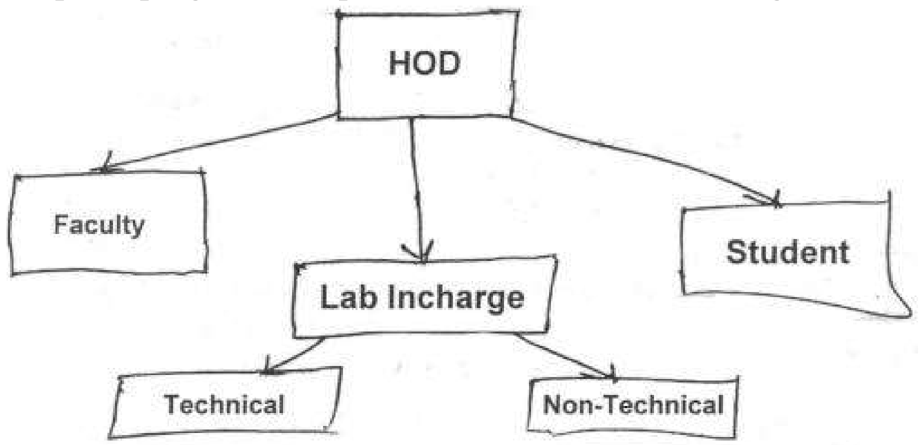
Operator overloading is a feature in C++ that allows you to redefine the behavior of built-in operators for user-defined data types. This enhances code readability and expressiveness.

Rules of Operator Overloading

- ✓ Only built-in operators can be overloaded.
- ✓ The arity of the operator cannot be changed. For instance, you cannot make a unary operator binary or vice versa.
- ✓ Operator precedence and associativity cannot be changed.
- ✓ Overloaded operators follow the same rules of precedence and associativity as their built-in counterparts.
- ✓ Overloaded operators cannot be new operators. They must have existing definitions.
- ✓ The return type of an overloaded operator should be intuitive. For example, overloading + for string concatenation should return a string.

Overloading the + Operator for String Concatenation

```
#include <iostream>
#include <string>
using namespace std;
class String {
public:
    string str;
    String(const string& s) : str(s) {}
    String operator+(const String& other) {
        return String(str + other.str);
    }
    void display() {
        cout << str << endl;
    }
};
int main() {
    String s1("Hello");
    String s2(" World");
    String s3 = s1 + s2;
    s3.display();
    return 0;
}
```

18.	**Develop c++ program to implement inheritance shown in fig.**  HOD Faculty Lab Incharge Student Technical Non-Technical	**6**	**W-22**

```
#include <iostream>
using namespace std;
class HOD {
public:
    virtual void display() {
        cout << "HOD" << endl;
    }
};
class Faculty : public HOD {
```

```
public:
    void display() override {
        cout << "Faculty" << endl;
    }
};
class LabIncharge : public HOD {
public:
    virtual void display() {
        cout << "Lab Incharge" << endl;
    }
};
class Technical : public LabIncharge {
public:
    void display() override {
        cout << "Technical" << endl;
    }
};
class NonTechnical : public LabIncharge {
public:
    void display() override {
        cout << "Non-Technical" << endl;
    }
};
class Student : public HOD {
public:
    void display() override {
        cout << "Student" << endl;
    }
};
int main() {
    HOD *ptr;
    ptr = new Faculty();
    ptr->display();
```

	`ptr = new LabIncharge();`
	`ptr->display();`
	`ptr = new Technical();`
	`ptr->display();`
	`ptr = new NonTechnical();`
	`ptr->display();`
	`ptr = new Student();`
	`ptr->display();`
	`return 0;`
	`}`		

19.	**State the need of virtual function in C++.**	**2**	**S-22**
	Virtual functions are essential for achieving runtime polymorphism in C++. This means that the decision about which function to call is made at runtime based on the actual type of the object, rather than the declared type of the pointer or reference. Here's why virtual functions are crucial: Runtime polymorphism: Allows for different behaviors based on the object's type at runtime. Base class pointers: Enables using a base class pointer to refer to objects of derived classes and calling appropriate functions. Overriding functionality: Provides a mechanism to redefine behavior in derived classes without changing the base class interface. Generic programming: Facilitates writing generic code that can work with different derived classes. Design patterns: Supports design patterns like Template Method, Strategy, and Observer.		
20.	**Write a C++ program to overload add function to add two integer numbers and two float numbers.**	**4**	**S-22**
	`#include <iostream>`		

```
using namespace std;

int add(int a, int b) {

    return a + b;

}

float add(float a, float b) {

    return a + b;

}

int main() {

    int num1 = 10, num2 = 20;

    float f1 = 3.14, f2 = 2.71;

    cout << "Sum of integers: " << add(num1, num2) << endl;

    cout << "Sum of floats: " << add(f1, f2) << endl;

    return 0;

}
```

21.	**i) Define pointer operator and address operator with example.** **ii) Write a C++ program to declare a class train with members as train no and name. Accept and display data for one object of train. Use pointer to object to call functions of class**	**6**	**S-22**

i) Define pointer operator and address operator with example.

Pointer Operator (*)

The pointer operator, denoted by *, is used to dereference a pointer. This means it accesses the value stored at the memory location pointed to by the pointer

```
#include <iostream>

using namespace std;

int main() {

    int x = 10;

    int *ptr = &x; // ptr points to the address of x

    cout << *ptr; // Output: 10 (value at the address stored in ptr)
    return 0;
}
```

Address Operator (&)

The address operator, denoted by &, returns the memory address of a variable.

```
#include <iostream>
using namespace std;
int main() {
    int x = 10;
    int *ptr = &x; // ptr stores the address of x using the address operator
    cout << ptr; // Output: memory address of x
    return 0; }
```

ii) Write a C++ program to declare a class train with members as train no and name. Accept and display data for one object of train. Use pointer to object to call functions of class

```
#include <iostream>
#include <string>
using namespace std;
class Train {
public:
    int train_no;
    string name;
    void acceptData() {
        cout << "Enter train number: ";
        cin >> train_no;
        cin.ignore(); // Ignore newline character
        cout << "Enter train name: ";
        getline(cin, name);
    }
    void displayData() {
        cout << "Train Number: " << train_no << endl;
        cout << "Train Name: " << name << endl;
    }
};
int main() {
    Train train;
    Train *ptr = &train;
    ptr->acceptData();
    ptr->displayData();
    return 0;
}
```

22.	**Write a C++ program to overload "+" operator so that it will perform concatenation of two strings. (Use class get data function to accept two strings)**	**6**	**S-22**

```
#include <iostream>
#include <string>
using namespace std;
class String {
public:
    string str;
    void getData() {
        cout << "Enter a string: ";
        getline(cin, str);
    }
    String operator+(const String& other) {
        String temp;
```

	temp.str = str + other.str;
	return temp;
	}
	void display() {
	cout << str << endl;
	}
	};
	int main() {
	String s1, s2, s3;
	s1.getData();
	s2.getData();
	s3 = s1 + s2;
	s3.display();
	return 0;
	}		

23.	**Describe ‘this’ pointer with an example.**	**4**	**W-19**
	‘this’ pointer: C++ uses a unique keyword called „this" to represent an object that invokes a member function. This unique pointer is automatically passed to a member function when it is invoked. „this" is a pointer that always point to the object for which the member function was called. For example, the function call A.max ( ) will set the pointer „this" to the address of the object A. Then suppose we call B.max ( ), the pointer „this" will store address of object B. Example: #include<iostream.h> class sample {		

```
int a;
public:
void setdata(int x)
{
this ->a=x;
}
void putdata()
{
cout<<this ->a;
}
};
void main()
{
sample s;
s.setdata(100);
s.putdata( );
}
```

In the above example, this pointer is used to represent object s when

setdata ( ) and putdata ( ) functions are called.

24.	**Write a program to declare a class ‘book’ containing data members as ‘title’, ‘author-name’, ‘publication’, ‘price’. Accept and display the information for one object using pointer to that object.**	**6**	**W-19**

```
#include<iostream.h>
#include<conio.h>
class book
{
char author_name[20];
char title[20];
char publication[20];
float price;
public:
void Accept();
void Display();
};
void book::Accept()
{
cout<<"\n Enter book‟s title, author_name, publication and price \n:";
cin>> title >>author_name>> publication >> price;
}
void student::Display()
{
cout<<title <<”\t”<<author_name<<”\t”<<publication <<”\t”<<
price<<”\n”<<;
}
void main()
{
book b, *p;
clrscr();
p=&b;
p->Accept();
cout<<”title \t author_name \t publication \t price\n”;
p-> Display();
getch();
```

	}		
25.	**Write a program to overload the '—' unary operator to negate the values.**	**6**	**W-19**

```
#include<iostream.h>

#include<conio.h>

#include<string.h>

class Number

{

int x,y;

public:

Number (int a, int b)

{

a =x;

b =y;

}

void display()

{

cout<<"value of x=”<<x<<”\n Value of y= ”<<y;

}

void operator - ( )

{

x = - x;

y = - y;

}

};

void main ()

{
```

	Number N1(5,6);
	
	clrscr ();
	
	N1. display ();
	
	-N1;
	
	cout<<"\n After negation:";
	
	N1. display ();
	
	getch ();
	
	}		

26.	**Give meaning of following statements:** **int * ptr, a = 5; ptr = & a ;** **cout << * ptr ;** **cout << (* ptr) + 1;**	**2**	**S-19**

	int *ptr, a = 5;
	
	Declare pointer variable ptr and variable a with initial value 5.
	
	ptr = & a;
	
	initialize pointer variable with address of variable a (store address of variable a in ptr)
	
	cout<< * ptr;
	
	Displays value of a i.e. value at address stored inside ptr. It displays value 5.
	
	cout<< (* ptr) + 1;
	
	Displays value by adding 1 to the value at address stored inside ptr. It displays value 6		

27.	**(i) Write any three rules of operator overloading.** **(ii) Write a program in C++ to overload unary '_' operator to negate values of data members of class.**	**6**	**S-19**
	(i) Write any three rules of operator overloading.		

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 data type.

3. We can't change the basic meaning of an operator. That is to say, we can't redefine the plus(+) operator to subtract one value from other.

4. Overloaded operators follow the syntax rules of the original operators. They can't be overridden.

5. There are some operators that can't be overloaded.

6. We can't use friend functions to overload certain operators. However, member function scan be used to overload them.

7. Unary operators overloaded by means of member function take no

explicit arguments and return no explicit values, but, those overloaded by means of the friend function, take one reference argument (the object of the relevant class).

8. Binary operators overloaded through a member function, take one

explicit argument and those which are overloaded through a friend

function take two explicit arguments.

9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

10. Binary arithmetic operators such as +,-,* and / must explicitly

Return a value. They must not attempt to change their own arguments.

(ii) Write a program in C++ to overload unary '_' operator to negate values of data members of class.

(Note: Any other correct logic shall be considered)

```
#include<iostream.h>

#include<conio.h>

#include<string.h>
```

```
class Number
{
int x, y;
public:
Number (int a,int b)
{
a =x;
b =y;
}
void display()
{
cout<<"value of x=”<<x<<”\nValue of y= ”<<y;
}
void operator - ( )
{
x = - x;
y = - y;
}
};
void main()
{
Number N1(5,6);
clrscr();
N1.display();
-N1;
cout<<"\n After negation:";
```

	N1. display ();
	getch();
	}		

28.	**With suitable example, describe effect of ++ and – – operators used with pointer in pointer arithmetic.**	**4**	**W-18**

	++ Operator: - It is referred as increment operator that increments the value of variable. If ++ operator is used with pointer variable, then pointer variable points to next memory address that means pointer increment with respect to size of the data type used to declare pointer variable.
	Example:-
	int a[5]={10,20,30,40,50},*ptr;
	ptr=a[0];
	for(i=0;i<5;i++)
	{
	cout<<*ptr;
	ptr++;
	}
	In the above example, ptr points to memory location of a[0].
	Increment statement ptr++ increments ptr by memory size of int i.e 2 bytes and ptr points to a[1].
	- - Operator: - It is referred as decrement operator that decrements the value of variable. If - - operator is used with pointer variable, then pointer variable points to previous memory address that means pointer decrement with respect to size of the data type used to declare pointer variable.		

29.	**Write a C++ program to swap two integer numbers and swap two float**	**4**	**W-18**

	numbers using function overloading. (Hint : overload swap function)		

```
#include<iostream.h>
#include<conio.h>
void swap(int a,int b)
{
int temp;
temp=a;
 a=b;
 b=temp;
cout<<"\nInteger values after swapping are:"<<a<<" "<<b;
}
void swap(float x,float y)
{
float temp1=x;
 x=y;
 y=temp1;
cout<<"\nFloat values after swapping are:"<<x<<" "<<y;
}
void main()
{
clrscr();
swap(10,20);
swap(10.15f,20.25f);
getch();
}
```

30.	**Write a C++ program to overload binary operator '+' to concatenate two strings.**	**6**	**W-18**

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class opov
 {
char str1[10];
public:
void getdata()
{
cout<<"\nEnter a strings";
cin>>str1;
}
void operator +(opov o)
{
cout<<strcat(str1,o.str1);
}
 };
void main()
{
opov o1,o2;
clrscr();
o1.getdata();
o2.getdata();
 o1+o2;
getch();
 }
```