VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

BCSE102L- Structured and object-oriented programming

Dr. P.Keerthika
Associate Professor
School of Computer Science & Engineering
VIT, Vellore.

BCSE102L- Structured and object-oriented programming

Module:1	**C Programming Fundamentals**	**2 hours**
Variables - Reserved words – Data Types – Operators – Operator Precedence - Expressions - Type Conversions - I/O statements - Branching and Looping: if, if-else, nested if, if-else ladder, switch statement, goto statement - Loops: for, while and do…while – break and continue statements.		
Module:2	**Arrays and Functions**	**4 hours**
Arrays: One Dimensional array - Two-Dimensional Array – Strings and its operations. User Defined Functions: Declaration – Definition – call by value and call by reference - Types of Functions - Recursive functions - Storage Classes - Scope, Visibility and Lifetime of Variables.		
Module:3	**Pointers**	**4 hours**
Declaration and Access of Pointer Variables, Pointer arithmetic – Dynamic memory allocation – Pointers and arrays - Pointers and functions.		
Module:4	**Structure and Union**	**2 hours**
Declaration, Initialization, Access of Structure Variables - Arrays of Structure - Arrays within Structure - Structure within Structures - Structures and Functions – Pointers to Structure -		
Module:5	**Overview of Object-Oriented Programming**	**5 hours**
Features of OOP - Classes and Objects - "this" pointer - Constructors and Destructors - Static Data Members, Static Member Functions and Objects - Inline Functions – Call by reference - Functions with default Arguments - Functions with Objects as Arguments - Friend Functions and Friend Classes.		
Module:6	**Inheritance**	**5 hours**
Inheritance - Types of Inheritance: Single inheritance, Multiple Inheritance, Multi-level Inheritance, Hierarchical Inheritance - Multipath Inheritance - Inheritance and constructors.		
Module:7	**Polymorphism**	**4 hours**
Function Overloading - Operator Overloading – Dynamic Polymorphism - Virtual Functions - Pure virtual Functions - Abstract Classes.		
Module:8	**Generic Programming**	**4 hours**
Function templates and class templates, Standard Template Library.		
	Total Lecture hours:	**30 hours**

BCSE102L- Structured and object-oriented programming – Text Books and Reference Books

Text Book(s)	
1.	Herbert Schildt, C: The Complete Reference, 4th Edition, McGraw Hill Education, 2017
2.	Herbert Schildt, C++: The Complete Reference, 4th Edition, McGraw Hill Education, 2017.
Reference Books	
1.	Yashavant Kanetkar, Let Us C: 17th Edition, BPB Publicaitons, 2020.
2.	Stanley Lippman and Josee Lajoie, C++ Primer, 5th Edition, Addison-Wesley publishers, 2012.

VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

BCSE102P- Structured and object-oriented programming Laboratory

	Indicative Experiments
1.	Programs using basic control structures, branching and looping
2.	Experiment the use of 1-D, 2-D arrays and strings and Functions
3.	Demonstrate the application of pointers
4.	Experiment structures and unions
5.	Programs on basic Object-Oriented Programming constructs.
6.	Demonstrate various categories of inheritance
7.	Program to apply kinds of polymorphism.
8.	Develop generic templates and Standard Template Libraries.

Text Book(s)	
1.	Robert C. Seacord, Effective C: An Introduction to Professional C Programming, 1st Edition, No Starch Press, 2020.
Reference Book(s)	
1.	Vardan Grigoryan and Shunguang Wu, Expert C++: Become a proficient programmer by learning coding best practices with C++17 and C++20's latest features, 1st Edition, Packt Publishing Limited, 2020.

BCSE102L- Structured and Object-Oriented Programming

- **Module-7: POLYMORPHISM**
  - **Function Overloading**
  - **Operator Overloading**
  - **Dynamic Polymorphism**
  - **Virtual functions**
  - **Pure Virtual Functions**
  - **Abstract Classes**

Dynamic Polymorphism

- Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time.
- Dynamic binding or late binding.
- C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of **object that invokes the function**.
- This type of polymorphism is executed through **virtual functions and function overriding.**
- All the methods of runtime polymorphism get invoked during the run time.

Function Overriding

- Function overriding takes place when your derived class and base class both contain a function having the same name.
- Along with the same name, both the functions should have the same number of arguments as well as the same return type.
- The derived class inherits the member functions and data members from its base class.
- To override a certain functionality, you must perform function overriding.

Function Overriding

```
class Base
{
        public:
        void print()
        {
        cout << "Base Function" << endl;
        }
};
class Derived : public Base
{
        public:
        void print()
        {
        cout << "Derived Function" << endl;
        }
};
```

Function Overriding- Example 1

```
int main()
{
        Derived derived1;
        derived1.print();
        return 0;
}
```

OUTPUT

Derived Function

- Same function print() is defined in both Base and Derived classes
- So, when we call print() from the Derived object derived1, the print() from Derived is executed by overriding the function in Base.

Accessing overridden function in base class

```
class Base
{
        public:
        void print()
        {
        cout << "Base Function" << endl;
        }
};
class Derived : public Base
{
        public:
        void print()
        {
        cout << "Derived Function" << endl;
        }
};
```

Accessing overridden function in base class

```
int main()
{
        Derived derived1, derived2;
        derived1.print();
        derived2.Base::print();
        return 0;
}
```

OUTPUT

Derived Function
Base Function

Calling overridden function from derived class

```
class Base
{
        public:
        void print()
        {
        cout << "Base Function" << endl;
        }
};
class Derived : public Base
{
        public:
        void print()
        {
        cout << "Derived Function" << endl;
        base::print() // called the overridden function inside
                      the derived class itself
        }
};
```

Accessing overridden function in base class

```
int main()
{
        Derived derived1;
        derived1.print();
        return 0;
}
```

OUTPUT

Derived Function
Base Function

Calling overridden function using Pointer

```
class Base
{
        public:
        void print()
        {
        cout << "Base Function" << endl;
        }
};
class Derived : public Base
{
        public:
        void print()
        {
        cout << "Derived Function" << endl;
        }
};
```

Accessing overridden function in base class

```
int main()
{
        Derived derived1;
        base *ptr = &derived1// pointer of base type that points to derived 1 (object of derived class)
        ptr->print(); // call function of Base class using ptr
        return 0;
}
```

Member function of base class is not overridden- how to avoid this??

OUTPUT

Base Function

- Created a pointer of Base type named ptr that points to the Derived object derived1.
- When we call the print() function using ptr, it calls the overridden function from Base.
- This is because even though ptr points to a Derived object, it is actually of Base type. So, it calls the member function of Base

Virtual Functions

- A virtual function is a member function in the base class that we expect to redefine in derived classes.
- Basically, a virtual function is used in the base class in order to ensure that the function is **overridden**. This especially applies to cases where a pointer of base class points to an object of a derived class.

```
class Base
{
public:
        virtual void print()
        { // code }
};
```

Virtual Function- Example

```
class Base
{
        public:
        virtual void print()
        {
        cout << "Base Function" << endl;
        }
};
class Derived : public Base
{
        public:
        void print()
        {
        cout << "Derived Function" << endl;
        }
};
```

Virtual Function- Example

```
int main()
{
        Derived derived1;
        base *ptr = &derived1// pointer of base type that points to derived 1(object of derived class)
        ptr->print(); // calls member function of derived class
        return 0;
}
```

OUTPUT

Derived Function

Virtual Function- Example

```
class shape
{
        public:
        double base,height;
        shape(double a, double b)
        {
        base=a;
        height=b;
        }
        virtual double area()
        {
                cout<<" base class area";
                return 0;
        }
};
```

Virtual Function- Example

```
class triangle: public shape
{
        triangle(double a, double b) : shape (a,b){ }
        double area()
        {
                cout<<" area of triangle";
                return (base *height)/2;
        }
};
class rectangle: public shape
{
        rectangle(double a, double b) : shape (a,b){ }
        double area()
        {
                cout<<" area of rectangle";
                return (base *height);
        }
};
```

Virtual Function- Example

```
int main()
{
        triangle t(10.0,20.0);
        cout<<t.area(); // static binding or early binding
        rectangle r(10.0,20.0);
        cout<<r.area(); // static binding or early binding
}
```

OUTPUT

Area of Triangle
100
Area of rectangle
200

Virtual Function- Example

```
int main()
{
        shape *s;
        triangle t(10.0,20.0);
        s=&t;
        cout<<s->area(); // dynamic binding or late binding
        rectangle r(10.0,20.0);
        s=&r;
        cout<<s->area();
}
```

OUTPUT

Area of Triangle
100
Area of rectangle
200