README.md

Object-Oriented Programming

▼ Class and Objects

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

/*
A class is just a blueprint for creating objects.
It defines a datatype by bundling data and methods that work on data into a single u
*/

class Car {
    public :
            string brand; // data member
            string model;
            int year;

            void displayInfo (){ // method call
                cout << "Brand : " << brand << "\nModel: " << model << "\nyear : " <
            }
};

int main ()
{
    cout << "hello world !!!" << endl;
    Car car1;
    car1.brand = "Toyota";
    car1.model = "Corolla";
    car1.year = 2000;

    car1.displayInfo();

    return 0;
}
```

▼ Constructor

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

/*
Class ::
- A class is just a blueprint for creating objects.
- It defines a datatype by bundling data and methods that work on data into a single

Constructor ::
- It's a special member function that is automatically called when an object is crea
- it initializes the data members of the object.
- They have the same name as class, but don't have return type.

Destructor ::
- It's a special member function that is automatically called when an object is dest
---  (goes out of scope or explicitly deleted)
- It's name is same as the class, but it is preceded by a tilde (~)
- They are used for cleanup, like releasing memory or resources.
*/

class Car {
    public :
            string brand; // data member
            string model;
            int year;

            // Constructor
            Car (string b, string m, int y){
                brand = b;
                model = m;
                year = y;
                cout << "Car object created for " << brand << " " << model << " " <<
            }

            // Destructor
            ~Car(){
                cout << "Car object destroyed for " << brand << " " << model << " "
            }

            void displayInfo (){ // method call
                cout << "Brand : " << brand << "\nModel: " << model << "\nyear : " <
            }
};

int main ()
{
    cout << "hello world !!!" << endl;
    // Car car1;
```

```
    // car1.brand = "Toyota";
    // car1.model = "Corolla";
    // car1.year = 2000;

    Car car1("Toyota", "Corolla", 2000);
    car1.displayInfo();


    return 0;
}
```

▼ Member initialization better than constructor (Modified Constructor)

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

/*
Class ::
- A class is just a blueprint for creating objects.
- It defines a datatype by bundling data and methods that work on data into a single

Constructor ::
- It's a special member function that is automatically called when an object is crea
- it initializes the data members of the object.
- They have the same name as class, but don't have return type.

Destructor ::
- It's a special member function that is automatically called when an object is dest
---  (goes out of scope or explicitly deleted)
- It's name is same as the class, but it is preceded by a tilde (~)
- They are used for cleanup, like releasing memory or resources.
*/

class Car {
    public :
            string brand; // data member
            string model;
            int year;

/*
Member initialization list ::
- It's a more efficient way to initialize class members, especially for constant mem
- It's defined after the contructor's signature and before the constructor body.
*/

            // Constructor but with better initialization
            Car (string b, string m, int y) : brand(b), model(m), year(y) {
                cout << "Car object created for " << brand << " " << model << " " <<
```

```
                }
                // Car (string b, string m, int y){
                //     brand = b;
                //     model = m;
                //     year = y;
                //     cout << "Car object created for " << brand << " " << model << " "
                // }

                // Destructor
                ~Car(){
                    cout << "Car object destroyed for " << brand << " " << model << " "
                }

                void displayInfo (){ // method call
                    cout << "Brand : " << brand << "\nModel: " << model << "\nyear : " <
                }
    };

    int main ()
    {
        cout << "hello world !!!" << endl;

        Car car1("Toyota", "Corolla", 2000);
        car1.displayInfo();


        return 0;
    }
```

▼ const_member functions

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

class Car {
    public :
            string brand; // data member
            string model;
            int year;

            // Constructor but with better initialization
            Car (string b, string m, int y) : brand(b), model(m), year(y) {
                cout << "Car object created for " << brand << " " << model << " " <<
            }

            // Destructor
            ~Car(){
                cout << "Car object destroyed for " << brand << " " << model << " "
```

```
            }

/*
Const member functions ::
- A const member function guarantees that it will not modify member variables of the
- if you try modifying any new member variable inside a const member function, the c
*/
            void displayInfo () const { // method call
                cout << "Brand : " << brand << "\nModel: " << model << "\nyear : " <
            }
            // void changeYear (int newYear) const {
            //     year = newYear; // throws an error
            // }
};

int main ()
{
    cout << "hello world !!!" << endl;

    Car car1("Toyota", "Corolla", 2000);
    car1.displayInfo();


    return 0;
}
```

▼ static member function

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

class Car {
/*
Static Members ::
- Static members (variables or functions) belong to the class rather than to any obj
- static variables retain their values between function calls and are shared by all
- static functions can only access static variables.
*/
    public :
            static int CarCount; // static member variable
            string brand;
            string model;
            int year;

            // Constructor
            Car (string b, string m, int y) : brand(b), model(m), year(y) {
                cout << "Car object created for " << brand << " " << model << " " <<
                CarCount++;
```

```
                }

                static void displayCarCount () {
                    cout << "Total numbers of cars : " << CarCount << endl;
                }

                // Destructor
                ~Car(){
                    cout << "Car object destroyed for " << brand << " " << model << " "
                }
                void displayInfo () const { // method call
                    cout << "Brand : " << brand << "\nModel: " << model << "\nyear : " <
                }
    };

    // initialize static member
    int Car::CarCount = 0;

    int main ()
    {
    /*
    - CarCount is staic member variable, it will be shared by all objects of the Car Cla
    - displayCarCount fucntion is static and can be called without an object (Car::displ
    */
        Car::displayCarCount();
        Car car1("Toyota", "Corolla", 2000);
        car1.displayInfo();
        Car::displayCarCount();
        Car car2("BMW", "X5", 2022);
        car2.displayInfo();
        Car::displayCarCount();

        return 0;
    }
```

▼ Operator Overloading

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

class Complex {
    private :
        float real;
        float imag;

    public :
        // Constructor
        Complex (float r=0, float i=0) : real(r), imag(i) {}
```

```
        // Operator overloading for "+"
        Complex operator+(const Complex &obj){
            Complex temp;
            temp.real = real + obj.real; // or this->real can be used.
            temp.imag = imag  + obj.imag;
            return temp;
        }
        void display() const {
            cout << real << " + " << imag << "i" << endl;
        }
};

int main ()
{
    Complex c1(3.4, 5.5);
    Complex c2(2.3, 4.4);
/*
- Operator "+" is overloaded to handle expressions like c1 + c2 work naturally with
*/
    Complex c3 = c1 + c2; // using overloaded "+" operator
    c3.display();
}
```

▼ Friend functions

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

/*
Friend functions ::
- A function that is not a member of a class but still has access to it's private an
- Useful when you want an external function to access the class private's data.
*/
class Box {
    private:
        double width;
    public:
        Box (double w) : width(w) {}

        // Friend function
        friend void printWidth (Box &box) {
            cout << box.width << endl;
        }
};

int main ()
```

```
{
    Box b(10.5);
    printWidth(b); //Accessing private member

    return 0;
}
```

▼ Copy Constructor

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

/*
Copy Constructor ::
- It's a special constructor used to create a new object as a copy of an existing ob
- You can define yout own copy constructor, or the compiler provides a default one,
- Shallow copy, in that objects share same memory location. (it just duplicated poin
- Custom copy constructor will do a deep copy. (duplicates the actual data)

General copy constructor ::
-- ClassName (const ClassName &obj); // reference top object that is being copied an
*/

class Line {
    private:
        int *length;

        public:
        Line(int l) {
            length = new int;
            *length = l;
        }

        // copy constructor
        Line (const Line &obj){
            length = new int;
            *length = *(obj.length);
        }

        void display() {
            cout << "Length : " << *length << endl;
        }

        // Destructor
        ~Line() {
            delete length;
        }
```

```
};

int main ()
{
    Line line1(10);
    Line line2 = line1; // copy constructor is called

    line1.display();
    line2.display();

}
```

▼ Move Constructor

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

/*
Move constructor ::
- It's used to efficiently tranfer resources from one object to another, especially
- It prevents unneccessary deep copying.

General ::
- ClassName(ClassName &&obj);
*/

class Example {
    private:
        int *data;
    public:
        // Constructor
        Example(int value){
            data = new int(value);
            cout << "Constructor constructor, Resource allocated to " << data << end
        }

        // Copy Constructor
        Example(const Example &obj){
            data = new int(*(obj.data));
            cout << "Copy constructor, Resource copied to " << data << endl;
        }

        // Move constructor
        Example (Example &&obj) noexcept {
            data = obj.data; // transfer ownership of data.
            obj.data = nullptr; // nullify the source pointer.
            cout << "Move constructor, Resource moved to " << data << endl;
        }
```

```
        //Destructor
        ~Example() {
            if (data){
                cout << "Destructor : Resource freed at " << data << endl;
                delete data;
            }else{
                cout << "No resource to free" << endl;
            }
        }

        // Display
        void display() const {
            if (data){
                cout << "Value : " << *data << endl;
            }else {
                cout << "No data available" << endl;
            }
        }
};

int main()
{
    Example ex1(10);
    ex1.display();
    Example ex2= move(ex1);
    ex1.display();
    ex2.display();
}
```

▼ Dynamic Memory Allocation

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

// new and delete operators are used to allocate and deallocate memory at runtime.

class DynamicArray {
    private:
        int *array;
        int size;

    public:
        DynamicArray(int s) : size(s) {
            array = new int[size];
            cout << "Array of size " << size << " created." << endl;
        }
```

```
        void setValue(int index, int value){
            if (index >= 0 && index < size){
                array[index] = value;
            }
        }

        int getValue(int index) const {
            if (index >=0 && index < size){
                return array[index];
            }
            cout << "Out of bounds" << endl;
            return -1; // if out of bounds
        }

        ~DynamicArray(){
            delete[] array;
            cout << "Array destroyed." << endl;
        }

};

int main()
{
    DynamicArray arr(5);

    for (int i=0; i<5; i++)
    {
        arr.setValue(i, i+1);
    }

    for (int i=0; i<5; i++)
    {
        cout << "Element" << i << " " << arr.getValue(i) << endl;
    }

    return 0;
}
```

▼ Copy Assigned Operator

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

/*
Copy Constructor
- The copy assignment operator pergorms a deep copy of resources from one object to
- It's used when assigning a value to an already existing object.
- It assigns the contents of one object to another existing object of the same class
```

```
- Unlike the copy assignment operator, which initializes a new object, the copy assi
--- already existing object.
- ClassName& operator=(const ClassName &obj)
*/
class Example {
    private:
        int *data;

    public:
        Example (int value){
            data = new int(value);
            cout << "Constructor : Resource allocated at " << data << endl;
        }

        // Copy constructor
        Example (const Example &obj){
            data = new int(*obj.data);
            cout << "Copy Constructor : Resource copied to " << data << endl;
        }

        // Copy Assignment Operator
        Example& operator=(const Example &obj){
            if (this != &obj) { // Self-assigned check
                delete data; // free existing resource
                data = new int(*obj.data); // Deep copy
                cout << "Copy Assignment Operator : Resource assigned to " << data <
            }
            return *this;
        }

        // Destreuctor
        ~Example() {
            delete data;
            cout << "Destructor : Resource freed at " << data << endl;
        }

        void display () const {
            cout << "value : " << *data << endl;
        }
};

int main()
{
    Example obj1(10); // Constructor
    Example obj2(20); // Constructor

    Example obj3 = obj1; // Copy Constructor (create a new object as a copy of an ex

    obj2 = obj1; // copy assignment operator Constructor (assign value from an exist
```

```
    obj1.display();
    obj2.display();
    obj3.display();

    return 0;
}
```

▼ Move Assigned Operator

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

/*
Move assginment Operator
- It is used to transfer ownership of resources from one existing to another, rather
- ClassName& operator-(ClassName &&obj) noexcept;
*/

class Example {
    private:
        int *data;
    public:
        // Constructor
        Example(int value){
            data = new int(value);
            cout << "Constructor : Resource allocated at " << data << endl;
        }

        // Move Constructor
        Example (Example &&obj) noexcept {
            data = obj.data;
            obj.data = nullptr;
            cout << "Move Constructor : Resource moved to " << data << endl;
        }

        // Move Assigned Operator
        Example& operator=(Example &&obj) noexcept {
            if (this != &obj) {
                delete data;
                data = obj.data;
                obj.data = nullptr;
                cout << "Move assignment Operator : Resource allocated to " << data
            }
            return *this;
        }

        ~Example() {
```

```
            delete data;
            cout << "Destructor : Resource freed at : " << data << endl;
        }

        void display() const {
            if (data){
                cout << "Value : " << *data << endl;
            }else {
                cout << "No data available" << endl;
            }
        }

};

int main()
{
    Example ex1(10);
    ex1.display();
    Example ex2(20);
    ex2.display();

    Example ex3 = move(ex1); // move constructor
    ex1.display();
    ex3.display();
    ex2 = move(ex3); // move assigned constructor
    ex2.display();

    return 0;
}
```

▼ Rule of five

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

/*
- Defining any of these functions usually implies that your class manages resources,
--- able to handle copying and moving explicitly to avoid resource leaks and undefin
*/

class Example {
    private:
        int *data;

    public:
        // Constructor
        Example(int value) : data(new int(value)) {}
```

```
    // Destructor
    ~Example() {
        delete data;
        cout << "Destructor : Resource freed at " << data << endl;
    }

    // Copy constructor
    Example (const Example &obj){
        data = new int(*obj.data);
        cout << "Copy Constructor : Resource copied to " << data << endl;
    }

    // Copy Assignment Operator
    Example& operator=(const Example &obj){
        if (this != &obj) { // Self-assigned check
            delete data; // free existing resource
            data = new int(*obj.data); // Deep copy
            cout << "Copy Assignment Operator : Resource assigned to " << data <
        }
        return *this;
    }

    // Move Constructor
    Example (Example &&obj) noexcept {
        data = obj.data;
        obj.data = nullptr;
        cout << "Move Constructor : Resource moved to " << data << endl;
    }

    // Move Assigned Operator
    Example& operator=(Example &&obj) noexcept {
        if (this != &obj) {
            delete data;
            data = obj.data;
            obj.data = nullptr;
            cout << "Move assignment Operator : Resource allocated to " << data
        }
        return *this;
    }

    void display() const {
        if (data){
            cout << "Value : " << *data << endl;
        }else {
            cout << "No data available" << endl;
        }
    }

};
```

```
int main()
{
    Example obj1(10); // Constructor
    Example obj2(20); // Constructor

    Example obj3 = obj1; // Copy Constructor (create a new object as a copy of an ex

    obj2 = obj1; // copy assignment operator Constructor (assign value from an exist

    obj1.display();
    obj2.display();
    obj3.display();

    Example ex4(10);
    ex4.display();
    Example ex5(20);
    ex5.display();

    Example ex6 = move(ex4); // move constructor
    ex4.display();
    ex6.display();
    ex5 = move(ex6); // move assigned constructor
    ex5.display();
}
```

▼ Smart Pointers

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

/*
Smart Pointers ::
- They are template classes provided by the STL to manage dynamic memory and other r
- They help prevent common issues like memory leaks, dangling pointers, and double d
--- that resources are properly cleaned up when no longer needed.
Types of Smart Pointers ::
- std::unique_ptr
--- std::unique_ptr<Type> ptr(new Type());
--- It manages a single object with unique ownership.
--- It can't be copied, only moved.
--- It automatically deletes, when it goes out of scope.

- std::shared_ptr
--- std::shared_ptr<Type> ptr1 = std::make_shared<Type>();
    std::shared_ptr<Type> ptr2 = ptr1
--- it manages a shared object with reference counting.
```

```
--- Multiple std::shared_ptr instances can own the same object.
--- The object is deleted when the last std::shared_ptr owning it is destroyed.

- std::weak_ptr
--- std::weak_ptr<Type> weakPtr = ptr1;
--- It provides a non-owning "weak" reference to an object managed by shared_ptr.
--- It helps prevent circular references which can lead to memory leaks.
--- It doesn't affect the reference count of managed object.
*/

class Example {
    public:
        Example(){
            cout << "Example constructor" << endl;
        }

        ~Example() {
            cout << "Example destructor" <<  endl;
        }
};

int main()
{
    // unique pointer
    unique_ptr<Example> ptr1 = make_unique<Example> (); // creating a unique pointer
    //unique_ptr<Example> ptr2 = ptr1; // unique pointer can't be copied.
    unique_ptr<Example> ptr = std::move(ptr1); // moving onwership, ptr1 is now null

    // shared pointer
    shared_ptr<Example> ptr3 = make_shared<Example>(); // creating a shared ptr
    shared_ptr<Example> ptr4 = ptr3; // sharing ownership
    cout << "Use count : " << ptr3.use_count() << endl;

    // weak pointer
    shared_ptr<Example> sharedPtr = make_shared<Example> ();
    weak_ptr<Example> weakPtr = sharedPtr;

    if (auto lockedPtr = weakPtr.lock()){ // Converting weak_ptr to shared_ptr
        cout << "Object is still alive." <<  endl;
    } else {
        cout << "Object has been deleted." << endl;
    }

    sharedPtr.reset(); // Reset shared ptr, object is deleted.

    if (auto lockedPtr = weakPtr.lock()) {
        cout << "Object is still alive." << endl;
    }else {
        cout << "object has been deleted." << endl;
```

```
    }

    return 0;
}
```

▼ Inheritance and Polymorphism

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

/*
Inheritance and Polymorphism ::
- It allows you to create hierachial relationship between classes and enable objects

Inheritance ::
- It allows a class (derived class) to inherit attributes and methods from another c
- It promotes code reuse and establishes a natural hierarchy among classes.

Types ::
- Single inheritance : A derived class inherit from a single base class.
- Multiple inheritance : A derived class inherit from more than one base class.
- Multi-level inheritance : A class is derived from a class which is also derived fr
- Hierarchial inheritance : Multiple classes inherit from a single base class.
*/

// Base Class
class Animal {
    public:
        void eat() {
            cout << "Eating..." << endl;
        }
};

// Derived Class
class Dog : public Animal {
    public:
        void bark() {
            cout << "Barking..." << endl;
        }
};

int main ()
{
    Dog myDog;
    myDog.bark();
    myDog.eat();
```

```
    return 0;
}
```

▼ Single Inheritance

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

/*
In single inheritance, one class inherits from a single base class.
*/

// Base Class
class Animal {
    public:
        void eat() {
            cout << "Animal is eating..." << endl;
        }
};

// Derived Class
class Dog : public Animal {
    public:
        void bark() {
            cout << "Dog is barking..." << endl;
        }
};

int main ()
{
    Dog myDog;
    myDog.bark();
    myDog.eat();

    return 0;
}
```

▼ Multiple Inheritance

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

/*
In multiple inheritance, a class inherits from more than one base class.
*/

// Base Class 1
```

```
class Animal {
    public:
        void eat() {
            cout << "Animal is eating..." << endl;
        }
};

// Base Class 2
class Pet {
    public:
        void play() {
            cout << "Pet is playing." << endl;
        }
};

// Derived Class
class Dog : public Animal, public Pet {
    public:
        void bark() {
            cout << "Dog is barking..." << endl;
        }
};

int main ()
{
    Dog myDog;
    myDog.bark();
    myDog.eat();
    myDog.play();

    return 0;
}
```

▼ Multi-level Inheritance

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

/*
In multi-level inheritance, a derived class is further derived by another class, cre
*/

// Base Class 1
class Animal {
    public:
        void eat() {
            cout << "Animal is eating..." << endl;
        }
};
```

```
// Intermediate Derived Class
class Mammal : public Animal {
    public:
        void walk() {
            cout << "Mammal is walking." << endl;
        }
};

// Further Derived Class
class Dog : public Mammal {
    public:
        void bark() {
            cout << "Dog is barking..." << endl;
        }
};

int main ()
{
    Dog myDog;
    myDog.bark();
    myDog.eat();
    myDog.walk();

    return 0;
}
```

▼ Hierarchial Inheritance

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

/*
In hierarchial inheritance, multiple classess inherit from the same base class.
*/

// Base Class
class Animal {
    public:
        void eat() {
            cout << "Animal is eating..." << endl;
        }
};

// Derived Class 1
class Cat : public Animal {
    public:
        void meow() {
```

```
            cout << "Cat is meowing." << endl;
        }
};

// Derived Class 2
class Dog : public Animal {
    public:
        void bark() {
            cout << "Dog is barking..." << endl;
        }
};

int main ()
{
    Dog myDog;
    Cat myCat;

    myDog.bark();
    myDog.eat();

    myCat.eat();
    myCat.meow();

    return 0;
}
```

▼ Template Programming

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

/*
Templace Programming
- It allows you to write generic and reusable code that works with any data type.
- Templates are a powerful feature in C++ that enable functions and classes to opera

-- Types ::
--- Functional, Class, Template specialization and Variadic templates.

*** Functional Templates ***
- It creates a single function that can work with different data types.
- template <typename T>
T functionName(T param) {
    // Function implementation
}
--- T is a placeholder that will be replaced by actual type when function is called.
*/

template <typename T>
```

```
T add(T a, T b)
{
    return a + b;
}

/*
*** Class Templates ***
- It allows you to create a generic class that can work with any data type.
- template <typename T>
    class ClassName {
        private:
            T member;
        public:
            ClassName(T arg) : member(arg) {}
        T getMember() {
            return member;
        }
    };
*/

template <typename T>
class Box {
    private:
        T value;
    public:
        Box(T v) : value(v) {}
        T getValue() {
            return value;
        }
};

/*
Template Specialization ::
- It allows you to create a specific implementation of a template for a particular d
- template <>
    class ClassName<SpecifierType> {
        // Specialized implementaion
    }
*/

template <typename T>
class Calculator {
    public:
        T add(T a, T b){
            return a + b;
        }
};

template <>
```

```
class Calculator<string> {
    public:
        string add (string a, string b){
            return a + " " + b;
        }
};

/*
variadic Templates ::
- It allows you to create functions or classes that accept a variable number of temp
- template <typename... Args>
void functionName(Args... args) {
    // code
}
*/

template <typename... Args>
void print(Args... args) {
    (cout << ... << args) << endl; // Fold expression C++17
}

int main()
{
    cout << add(3, 4) << endl;
    cout << add(3.4, 5.6) << endl;

    Box<int> intBox(123);
    Box<double> doubleBox(123.456);
    cout << "intBox : " << intBox.getValue() << endl;
    cout << "doubleBox : " << doubleBox.getValue() << endl;

    Calculator<int> intCalc;
    Calculator<string> stringCalc;
    cout << intCalc.add(3, 4) << endl; // Uses the generic template.
    cout << stringCalc.add("Hello", "World ~") << endl; // Uses the special template

    // Variadic
    print(1," ", 2," ", 3.5, " ", "Hello", " ", 'c');

    return 0;
}
```

▼ Lambda Expressions

#include #include #include using namespace std;

```
/*
Lambda Expressions ::
```

```
- It allows you to define anonymous functions (functions without a name) directly in
- It makes the code more consise and readable, especially for short functions that a
- Syntax ::
[capture] (params) -> return type {
    // functions body
}
--- capture defines variables from surrounding scope are captures and how (by value
--- parameters specifies regular function parameters.
-- return type is optional.
*/

int main ()
{
    // without capture
    auto add = [] (int a, int b) -> int{
        return a + b;
    };
    cout << "Sum : " << add(5, 3) << endl;

    // with capture
    int x = 10;
    int y = 20;
    auto addition = [=] () {
        return x + y; // x and y are captured by value. Change inside here won't aff
    };
    auto modify = [&] () {
        x = x + y; // x and y are captured by reference. Change inside here will aff
    };
    cout << "Sum (by value) : " << addition() << endl;

    modify();
    cout << "Modified x : " << x << endl;

    // Lambdas with STL
    vector<int> nums = {5, 2, 8, 1, 3};
    sort(nums.begin(), nums.end(), [](int a, int b){
        return a > b;
    });

    cout << "Sorted nums : " <<  endl;
    for (int n : nums){
        cout << n << " ";
    }
    cout << endl;

    // Generic lambdas C++14 and later, we can use auto to make lambda work with any
    auto print = [](auto x){
        cout << x << endl;
    };
```

```
    print(42);
    print(13.5);
    print("fafsd");
    print("132asfsa");

    return 0;
}
```

▼ Polymorphism

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

/*
Polymorphism ::
- It allows objects of different classes to be treated as objects of a common base c
- It enables funtions to process objects differently based on their actual derived t
--- accessed through a pointer or reference of the base class.

Two Types ::
- Compile-Time (Static) Polymorphism
--- Achieved using function overloading, operator overloading and templates.
--- The function to be called is resovled at compile time.

- Run-Time (Dynamic) Polymorphism
--- Acheived using inheritance and virtual functions.
--- The function to be called is determined at runtime based on the actual object ty
*/

int main()
{
    // Nothing here.
}
```

Function overloading

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

/*
- It allows multiple functions with the same name but different parameter lists to b
- The corret function to be called is determined by the arguments passed.
*/

class Print {
    public:
        void show (int i) {
```

```
            cout << "Integer : " << i << endl;
        }
        void show (double i) {
            cout << "Double : " << i << endl;
        }
        void show (string i) {
            cout << "String : " << i << endl;
        }
};

int main()
{
/*
- Function is overloaded 3 times with different arguments.
- The compiler will determine which function to call based on argument type at the c
*/
    Print p;
    p.show(10);
    p.show(10.7);
    p.show("fas4234");

    return 0;
}
```

▼ Virtual Function

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

/*
Run-Time Polymorphism ::
- Using inheritance and virtual functions.

Virtual Functions ::
- It is a function in a base class that is over-ridden in a derived class.
- They allow the correct function to be called for an object, regardless the type of
*/

// Base Class
class Animal {
    public:
        virtual void sound() {
            cout << "Animal makes a sound." << endl;
        }
};

/*
Declaring,
```

```
    virtual void sound() = 0; // it will become a pure virtual function which has no imp
    - A class containing a pure virtual function is considered abstract and cannot be in
    */

    // Derived Class
    class Dog : public Animal {
        public:
            void sound() override {
                cout << "Dog barks." << endl;
            }
    };

    // Another Derived Class
    class Cat : public Animal {
        public:
            void sound() override {
                cout << "Car meows." << endl;
            }
    };

    int main()
    {
        Animal *animalPtr;
        Dog dog;
        Cat cat;

        animalPtr = &dog;
        animalPtr->sound();

        animalPtr = &cat;
        animalPtr->sound();
    }
```