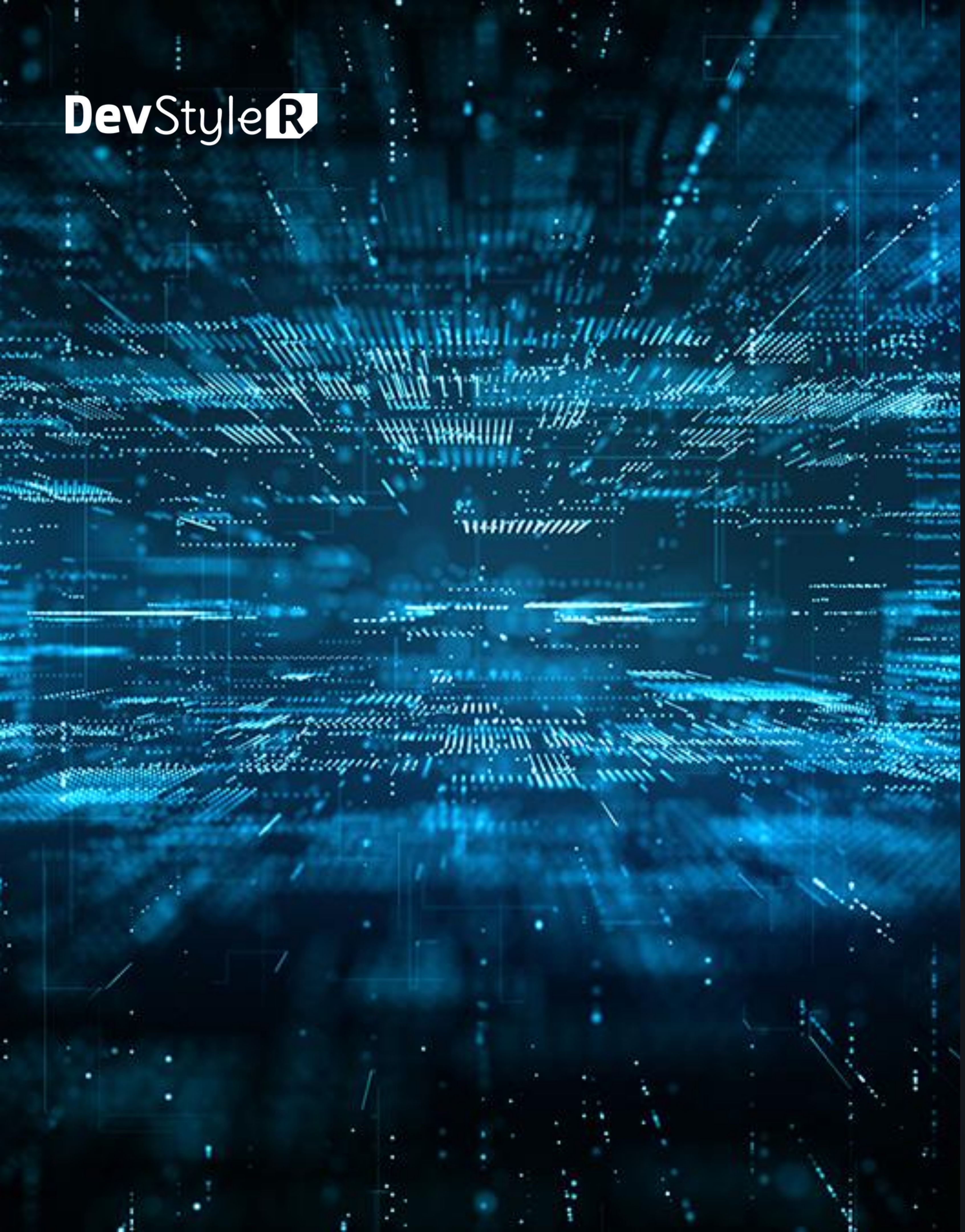
DevStyleR ACADEMY

Programming with C++ Core

# OBJECT-ORIENTED PROGRAMMING: POLYMORPHISM

2024



## Summary

- Polymorphism Examples
- Relationships Among Objects in an Inheritance Hierarchy
  - Invoking Base-Class Functions from Derived-Class Objects
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Derived-Class Member-Function Calls via Base-Class Pointers
  - Virtual Functions
- Type Fields and switch Statements
- Abstract Classes and Pure virtual Functions
- Virtual Destructor

## POLYMORPHISM EXAMPLES

- Polymorphism occurs when a program invokes a virtual function through a base-class pointer or reference
    - C++ dynamically chooses the correct function for the class from which the object was instantiated
- Example: SpaceObjects
    - Video game manipulates objects of types that inherit from SpaceObject, which contains member function draw
    - Function draw implemented differently for the different classes
    - Screen-manager program maintains a container of SpaceObject pointers
    - Call draw on each object using SpaceObject pointers
        - Proper draw function is called based on object’s type
    - A new class derived from SpaceObject can be added without affecting the screen manager

## RELATIONSHIPS AMONG OBJECTS IN AN INHERITANCE HIERARCHY

- Demonstration
  - Invoking base-class functions from derived-class objects
  - Aiming derived-class pointers at base-class objects
  - Derived-class member-function calls via base-class pointers
  - Demonstrating polymorphism using virtual functions
    - Base-class pointers aimed at derived-class objects
- Key concept
  - An object of a derived class can be treated as an object of its base class

## INVOKING  BASE-CLASS FUNCTIONS FROM DERIVED-CLASS OBJECTS

- Aim base-class pointer at base-class object
    - Invoke base-class functionality
- Aim derived-class pointer at derived-class object
    - Invoke derived-class functionality

DevStyleR ACADEMY

## INVOKING  BASE-CLASS FUNCTIONS FROM DERIVED-CLASS OBJECTS

- Aim base-class pointer at derived-class object
  - Because derived-class object is an object of base class
  - Invoke base-class functionality
    - Invoked functionality depends on type of the handle used to invoke the function, not type of the object to which the handle points
  - virtual functions
    - Make it possible to invoke the object type's functionality, rather than invoke the handle type's functionality
    - Crucial to implementing polymorphic behavior

DevStyleR ACADEMY

Programming with C++ Core

# LIVE DEMO

## CommissionEmployee

2024

## AIMING DERIVED-CLASS POINTERS AT BASE-CLASS OBJECTS

- Aim a derived-class pointer at a base-class object
  - C++ compiler generates error
    - CommissionEmployee (base-class object) is not a BasePlusCommissionEmployee (derived-class object)
  - If this were to be allowed, programmer could then attempt to access derived-class members which do not exist
    - Could modify memory being used for other data

DevStyle ACADEMY

# LIVE DEMO

## DerivedBaseClassObject

## DERIVED-CLASS MEMBER-FUNCTION CALLS VIA BASE-CLASS POINTERS

- Aiming base-class pointer at derived-class object
    - Calling functions that exist in base class causes base-class functionality to be invoked
    - Calling functions that do not exist in base class (may exist in derived class) will result in error
        - Derived-class members cannot be accessed from base-class pointers
        - However, this can be accomplished using downcasting

DevStyleR ACADEMY

Programming with C++ Core

# LIVE DEMO

## DerivedMemeberFunction

2024

DevStyleR ACADEMY

## VIRTUAL FUNCTIONS

- Which class’s function to invoke
  - Normally
    - Handle determines which class’s functionality to invoke
  - With virtual functions
    - Type of the object being pointed to, not type of the handle, determines which version of a virtual function to invoke
    - Allows program to dynamically (at runtime rather than compile time) determine which function to use
      - Called dynamic binding or late binding

DevStyleR ACADEMY

## VIRTUAL FUNCTIONS

- virtual functions
  - Declared by preceding the function’s prototype with the keyword virtual in base class
  - Derived classes override function as appropriate
  - Once declared virtual, a function remains virtual all the way down the hierarchy
  - Static binding
    - When calling a virtual function using specific object with dot operator, function invocation resolved at compile time
  - Dynamic binding
    - Dynamic binding occurs only off pointer and reference handles

DevStyleR ACADEMY

Programming with C++ Core

# LIVE DEMO

## Polymorphism

2024

## SUMMARY

### Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

- Aiming a base-class pointer at a base-class object
  - Is straightforward
- Aiming a derived-class pointer at a derived-class object
  - Is straightforward
- Aiming a base-class pointer at a derived-class object
  - Is safe, but can be used to invoke only member functions that base-class declares (unless downcasting is used)
  - Can achieve polymorphism with virtual functions
- Aiming a derived-class pointer at a base-class object
  - Generates a compilation error

## ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- Abstract classes
  - Classes from which the programmer never intends to instantiate any objects
    - Incomplete — derived classes must define the “missing pieces”
    - Too generic to define real objects
  - Normally used as base classes, called abstract base classes
    - Provides an appropriate base class from which other classes can inherit
    - Classes used to instantiate objects are called concrete classes
      - Must provide implementation for every member function they define

# ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- Pure virtual function
  - A class is made abstract by declaring one or more of its virtual functions to be “pure”
    - Placing “= 0” in its declaration
      - virtual void draw() const = 0;
      - “= 0” is known as a pure specifier
  - Do not provide implementations
    - Every concrete derived class must override all base-class pure virtual functions with concrete implementations
      - If not overridden, derived-class will also be abstract
  - Used when it does not make sense for base class to have an implementation of a function, but the programmer wants all concrete derived classes to implement the function

DevStyleR ACADEMY

## ABSTRACT CLASSES AND PURE VIRTUAL FUNCTIONS

- We can use the abstract base class to declare pointers and references
  - Can refer to objects of any concrete class derived from the abstract class
  - Programs typically use such pointers and references to manipulate derived-class objects polymorphically
- Polymorphism is particularly effective for implementing layered software systems

DevStyleR ACADEMY

## VIRTUAL DESTRUCTORS

- Nonvirtual destructors
    - Destructors that are not declared with keyword virtual
    - If a derived-class object is destroyed explicitly by applying the delete operator to a base-class pointer to the object, the behavior is undefined
- virtual destructors
    - Declared with keyword virtual
        - All derived-class destructors are virtual
    - If a derived-class object is destroyed explicitly by applying the delete operator to a base-class pointer to the object, the appropriate derived-class destructor is called
        - Appropriate base-class destructor(s) will execute afterwards

# EXERCISES

Shape

TwoDimensionalShape
- getArea

ThreeDimensionalShape
- getArea
- getVolume

Implement the Shape hierarchy designed in Lecture "Inheritance"

- Each TwoDimensionalShape should contain function getArea to calculate the area of the two-dimensional shape.
- Each ThreeDimensionalShape should have member functions getArea and getVolume to calculate the surface area and volume of the three-dimensional shape, respectively.
- Create a program that uses a vector of Shape pointers to objects of each concrete class in the hierarchy.
- The program should print the object to which each vector element points.
- Also, in the loop that processes all the shapes in the vector determine whether each shape is a TwoDimensionalShape or a ThreeDimensionalShape.
- If a shape is a TwoDimensionalShape, display its area. If a shape is a ThreeDimensionalShape, display its area and volume

## EXERCISES

Shape

TwoDimensionalShape

Square

Rectangle

Circle

Develop a basic graphics package

- Use the Shape hierarchy implemented in Exercise 1.
- Limit yourself to two-dimensional shapes such as squares, rectangles, triangles and circles.
- Interact with the user.
- Let the user specify the position, size, shape and fill characters to be used in drawing each shape.
- The user can specify more than one of the same shape.
- As you create each shape, place a Shape * pointer to each new Shape object into an array. Each Shape class should now have its own draw member function.
- Write a polymorphic screen manager that walks through the array, sending draw messages to each object in the array to form a screen image.
- Redraw the screen image each time the user specifies an additional shape.

## EXERCISES

Package

TwoDayPackage

OvernightPackage

Use the Package inheritance hierarchy created in Lecture "Inheritance" to create a program that displays the address information and calculates the shipping costs for several Packages.

- The program should contain a vector of Package pointers to objects of classes TwoDayPackage and OvernightPackage.
- Loop through the vector to process the Packages polymorphically.
- For each Package, invoke get functions to obtain the address information of the sender and the recipient, then print the two addresses as they would appear on mailing labels.
- Also, call each Package's calculateCost member function and print the result.
- Keep track of the total shipping cost for all Packages in the vector, and display this total when the loop terminates.

## EXERCISES

Account

SavingsAccount

CheckingAccount

Develop a polymorphic banking program using the Account hierarchy created in Lecture “Inheritance”

- Create a vector of Account pointers to SavingsAccount and CheckingAccount objects.
- For each Account in the vector, allow the user to specify an amount of money to withdraw from the Account using member function debit and an amount of money to deposit into the Account using member function credit.
- As you process each Account, determine its type.
- If an Account is a SavingsAccount, calculate the amount of interest owed to the Account using member function calculateInterest, then add the interest to the account balance using member function credit.
- After processing an Account, print the updated account balance obtained by invoking base class member function getBalance.

DevStyleR ACADEMY

# THANK YOU

ACADEMY@DEVSTYLER.IO