Object-Oriented Programming

Lecture 10: Inheritance

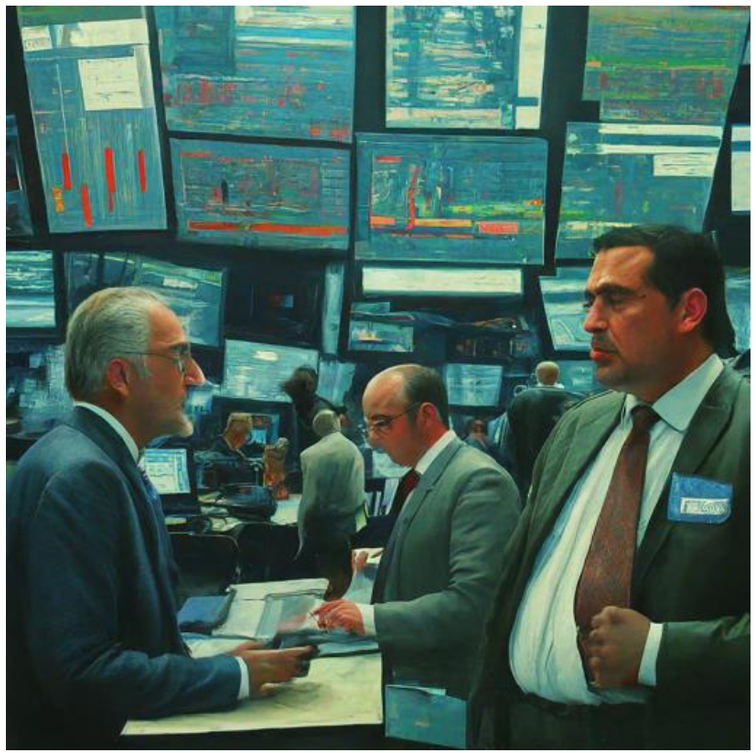
Agenda

- Think object
- Base Classes and Derived Classes
- Relationship between Base and Derived Classes
- Constructors and Destructors in Derived Classes
- public, protected and private Inheritance
- Dynamic memory allocation

Think Object

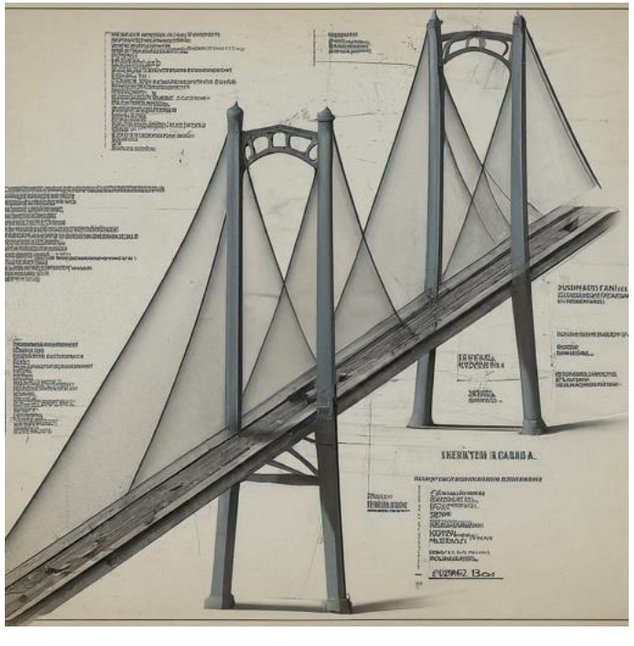
Procedures vs. Objects

- Procedural: Step-by-step instructions
- OOP: Objects with properties and methods



Concept of Object

- Class = Bluprint
- Object = Properties with Methods
- Properties = characteristics
- Methods = actions



Concept of OOP

- **Encapsulation**
  Bundling data (attributes) and methods (functions) that operate on the data into a single unit or class.
  The object's internal state is hidden from the outside, only exposing a controlled interface (public methods) for the outside world to interact with the object.
- **Inheritance**
  Mechanism that allows a new class (called a subclass or derived class) to inherit properties and behaviors (methods) from an existing class (known as a superclass or base class).
  Subclasses can override or extend the functionality of superclass methods.
- **Polymorphism**
  Single function or method can take on multiple forms depending on the context or the object invoking it.
  Two primary types of polymorphism: compile-time (or static), achieved through method overloading, and runtime (or dynamic), achieved through method overriding in inheritance hierarchies.

Benefits of OOP

- Increased Productivity
- Enhanced Software Quality
- Improved Maintainability
- Scalability
- Real-World Modeling

How to Think Objects

- **Think in First Person Perspective**
  Considering objects as entities that have their own identity, state, and behaviors. Unlike procedural programming, where the focus is on functions. Instead of thinking, "How do I sort this list?" you would consider, "How does the list sort itself?"
- **Think Small**
  Focusing on the granularity of components, breaking down the problem into smaller, more manageable objects.
- **Think Interaction**
  In procedural programming, the emphasis is on controlling the flow of the program through a series of procedures or functions. OOP focus on the interaction between objects through well-defined interfaces (methods) to achieve the desired outcomes. This interaction mimics real-world interactions, making designs more intuitive and adaptable.

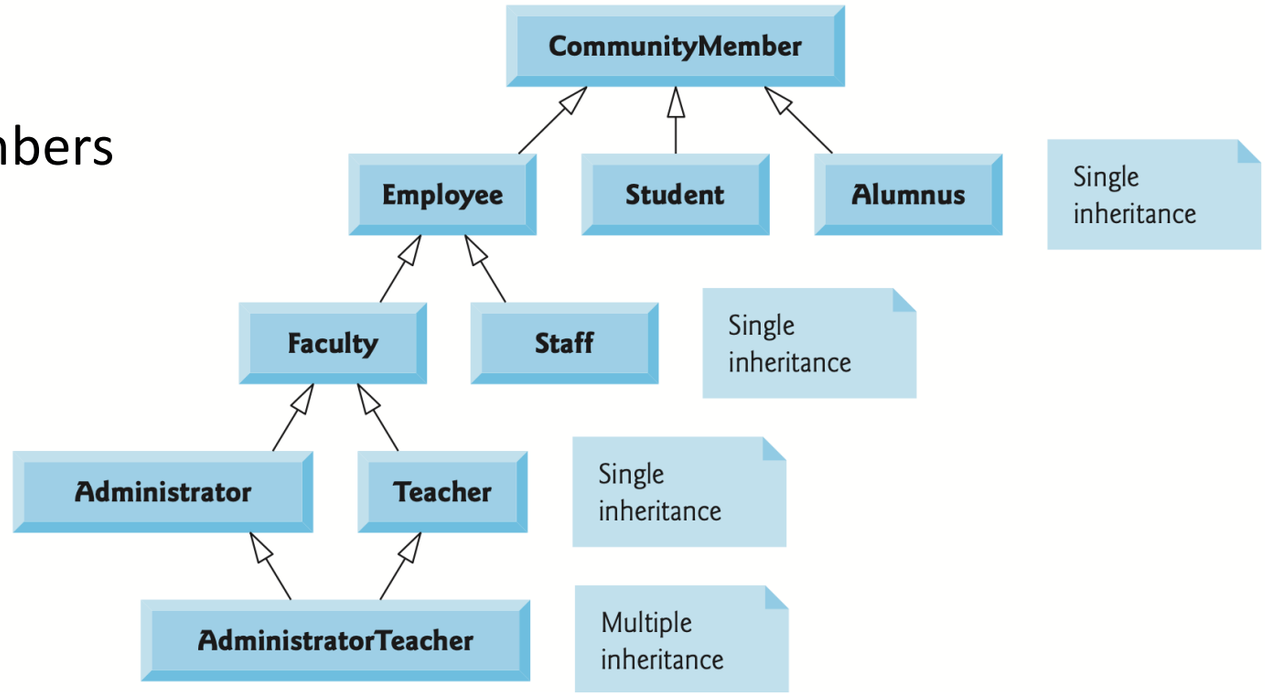
Base Classes and Derived Classes

Base Classes and Derived Classes

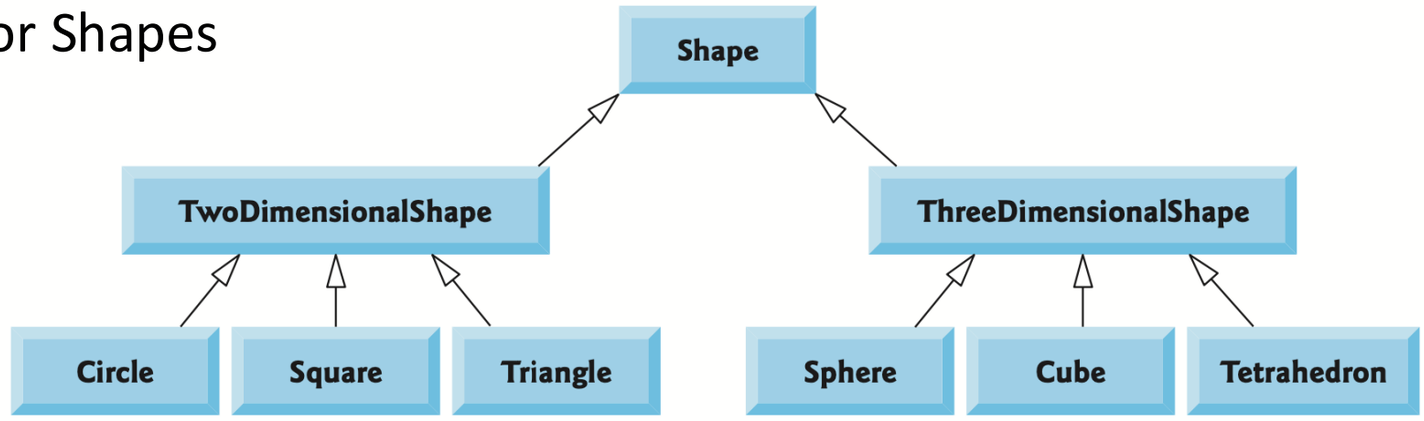
Base class	Derived classes
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
Account	CheckingAccount, SavingsAccount

Class Hierarchy

Inheritance hierarchy for university CommunityMembers



Inheritance hierarchy for Shapes



Relationship between Base and Derived Classes

- Commission employees (who will be represented as objects of a base class) are paid a percentage of their sales
- Base-salaried commission employees (who will be repre- sented as objects of a derived class) receive a base salary plus a percentage of their sales.
- Creating and Using a CommissionEmployee Class
  - “CommissionEmployee.h”
  - “CommissionEmployee.cpp”
  - “oop_lec12_01.cpp”

Relationship between Base and Derived Classes

Creating a BasePlusCommissionEmployee Class Without Using Inheritance

"BasePlusCommissionEmployee.h"
"BasePlusCommissionEmployee.cpp"
"oop_lec12_02.cpp"

Relationship between Base and Derived Classes

Creating a CommissionEmployee–BasePlusCommissionEmployee Inheritance Hierarchy

"BPCommissionEmployee.h"
"BPCommissionEmployee.cpp"
"oop_lec12_03.cpp"

Relationship between Base and Derived Classes

- public members are accessible from anywhere in the program where the object of the class or a derived class is accessible.
- private members are accessible only within the class itself (i.e., from within other member functions of the same class) and by friends of the class.
- protected members are similar to private members but are also accessible in derived classes.
- Inheriting protected data members slightly improves performance, directly access the members without incurring the overhead of calls to set or get member functions.
  “oop_lec12_04.cpp”
  “oop_lec12_05.cpp”
  “oop_lec12_06.cpp”

Constructors and Destructors in Derived Classes

When you create an object of a derived class:

- Base Class Constructor Called First: The constructor for the base class is called first to initialize the base class portion of the object. This ensures that any inherited members are set up correctly before the derived class attempts to use them or add its own initialization.
- Derived Class Constructor Called Next: After the base class constructor completes its execution, control returns to the derived class's constructor, which then performs its own initialization tasks. This includes setting up any members that are specific to the derived class.
- Constructor Initialization Chain: If there are multiple levels of inheritance, this process forms a chain, starting from the topmost base class down to the most derived class. Each constructor in the chain is responsible for initializing its part of the object.
- Destructors Called in Reverse Order: When the object goes out of scope or is explicitly deleted, destructors are called in the reverse order of the constructors. The destructor of the derived class is called first to clean up resources specific to the derived class, followed by the destructors of base classes, up to the topmost base class.
- For example, consider a class A derived from B, and B derived from C. When creating an object of type A, the order of constructor calls is C(), B(), A().

"oop_lec12_07.cpp"

Implicit Explicit Constructor

```
class MyClass {
public:
    MyClass(int x) : value(x) {}

private:
    int value;
};

int main() {
    MyClass obj1 = 10;  // Implicit constructor call
    MyClass obj2(20);   // Explicit constructor call

    return 0;
}
```

```
class MyClass {
public:
    explicit MyClass(int x) : value(x) {}

private:
    int value;
};

int main() {
    // MyClass obj1 = 10;  // Error: implicit conversion is not allowed
    MyClass obj2(20);   // Allowed: Explicit constructor call

    return 0;
}
```

Public Inheritance

- Public Members: If a base class member is public and the class is inherited publicly, the member remains public in the derived class. It can be accessed anywhere the derived object is accessible.
- Protected Members: If a base class member is protected and the class is inherited publicly, the member remains protected in the derived class. It can be accessed within the derived class and in classes further derived from it.
- Private Members: If a base class member is private, it remains inaccessible directly from the derived class, regardless of the inheritance type. It can only be accessed through public or protected member functions of the base class.

Protected Inheritance

- Public Members: If a base class member is public and the class is inherited protectedly, the member becomes protected in the derived class. It can be accessed within the derived class and in classes further derived from it.
- Protected Members: The same as above; protected members remain protected in the derived class.
- Private Members: The same as with public inheritance; private members remain inaccessible directly from the derived class.

Private Inheritance

- Public Members: If a base class member is public and the class is inherited privately, the member becomes private in the derived class. It can only be accessed within the derived class.
- Protected Members: If a base class member is protected and the class is inherited privately, the member also becomes private in the derived class. It can only be accessed within the derived class.
- Private Members: The same as with public and protected inheritance; private members remain inaccessible directly from the derived class.

public, protected and private Inheritance

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

Dynamic Memory Management in C++

- The process of allocating and deallocating memory during program execution at runtime.
- Unlike static memory allocation that happens at compile time.
- Dynamic memory allocation provides flexibility and allows programs to manage memory needs based on specific requirements.

Dynamic Memory Management in C++

Feature	int value = 5; Int* ptr = &value;	int* ptr = new int; *ptr = 5	std::shared_ptr<int> ptr = std::make_shared<int>(5);
Memory Allocation	None	Heap	Heap
Pointer Points To	Existing variable	Newly allocated memory	Newly allocated memory
Value Assignment	Modifies value	Assigns to allocated memory	Automatically initialized during allocation
Ownership	Independent	Manual (requires delete)	Automatic (managed by std::shared_ptr)
Memory Management Risk	None	High (potential leaks)	Low (automatic cleanup)
Usage Scenario	Accessing existing variable	Dynamic memory allocation (independent value)	Dynamic memory allocation with automatic cleanup

main(int argc, char* argv[])

- argc (int) "Argument Count"
  Holds the total number of command-line arguments passed to your program.
  The value of argc is always at least 1 (representing the program name itself).
- argv (char*[]) "Argument Vector"
  An array of character pointers (strings).
  Each element in argv points to a single command-line argument:
  argv[0] is the name of the program.
  argv[1] is the first actual argument passed.
  argv[2] is the second argument, and so on.

"oop_lec12_09.cpp"

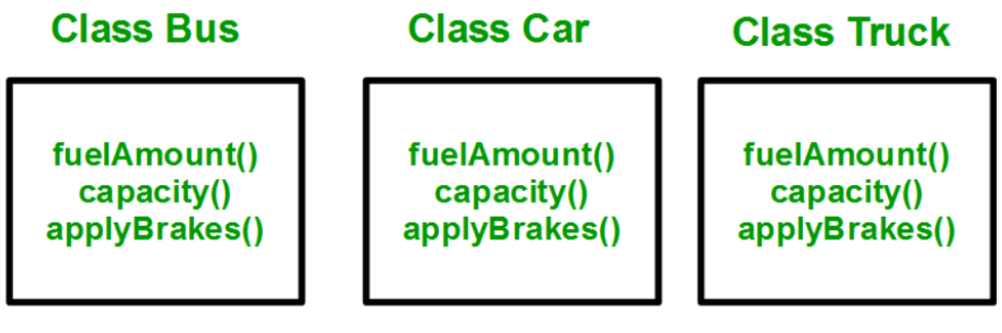
Inheritance

Inheritance : The capability of a class to derive properties and characteristics from another.

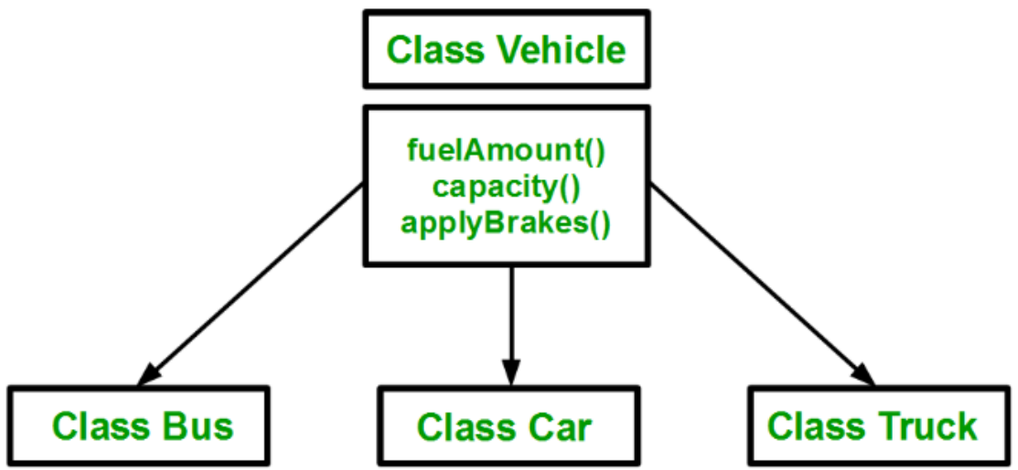
New classes are created from the existing classes. The new class created is called “derived class” or “child class” or “subclass” and the existing class is known as the “base class” or “parent class” or “superclass” . The derived class now is said to be inherited from the base class.

The derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own.

Why and when to use inheritance?




```
class  <derived_class_name> : <access-specifier> <base_class_name>
{
      //body
}
```



Access Specifiers

public: can be accessed from anywhere, both inside and outside the class and its derived classes.

protected: can be accessed directly within the class itself and its derived classes. Outside of these contexts, they are inaccessible.

private: private can only be accessed from within the class itself and are not accessible from derived classes or outside the class.

Access Specifiers

public: can be accessed from anywhere, both inside and outside the class and its derived classes.

protected: can be accessed directly within the class itself and its derived classes. Outside of these contexts, they are inaccessible.

private: private can only be accessed from within the class itself and are not accessible from derived classes or outside the class.

Base Class Access	Inheritance Type	Access in Derived Class
Public	Public	Public
Public	Protected	Protected
Public	Private	Private
Protected	Public	Protected
Protected	Protected	Protected
Protected	Private	Private
Private	Public	Not Accessible
Private	Protected	Not Accessible
Private	Private	Not Accessible

What is Polymorphism?

Polymorphism, from its Greek roots "poly" (many) and "morph" (form), means the ability to take multiple forms.
In C++, it means that the same function call or operator can behave differently depending on the object it's operating on.

Types of Polymorphism in C++

Compile-time Polymorphism (achieved through function overloading and operator overloading)

Function Overloading: Multiple functions with the same name but different parameter lists. The compiler selects the appropriate version at compile time. (lec13_01_function_overload.cpp)

Operator overloading allows you to redefine the way operators work with user-defined types. (lec13_02_operator_overload.cpp)

Runtime Polymorphism (achieved through virtual functions and inheritance)

Virtual Functions: Functions in a base class marked with the virtual keyword and overridden in derived classes. This allows the program to determine the correct function to call at runtime based on the object's type.

Virtual Table

vtable is a mechanism used in C++ (and other object-oriented languages) to support dynamic (runtime) polymorphism through virtual functions.

Each class that has virtual functions (or inherits from a class that has virtual functions) has its own vtable. This table is essentially an array of pointers to the virtual functions defined in the class.

When a virtual function is called on an object, the compiler uses the vtable of the object's class to look up the address of the correct function to call, based on the actual type of the object.

Virtual Table

Creation: The compiler creates a separate vtable for each class that has virtual functions or inherits virtual functions. The vtable contains one entry for each virtual function that can be called on the class's objects, directly or through inheritance.

Object Construction: When an object of a class is created, a pointer to the vtable (often called a vptr) is added to the object's memory layout, usually at the beginning. This pointer points to the vtable of the class the object was instantiated from.

Function Call: When a virtual function is called on an object, the compiler uses the object's vptr to access the correct vtable, and then uses the vtable to find the function to call. This allows the function call to be resolved at runtime based on the actual type of the object, not the type of the pointer/reference to the object.

Virtual Table

```
class Base
{
public:
   virtual void function1() {};
   virtual void function2() {};
};

class D1: public Base
{
public:
   void function1() override {};
};

class D2: public Base
{
public:
   void function2() override {};
};

int main()
{
   D1 d1 {};
   Base* dPtr = &d1;
    dPtr->function1();

   return 0;
}
```

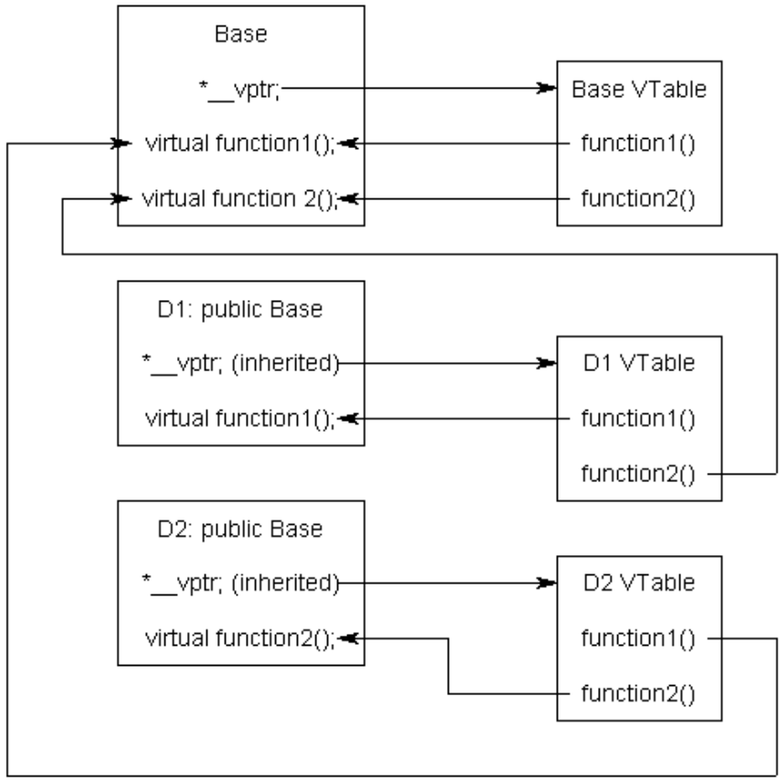
```
class Base
{
public:
   VirtualTable* __vptr;
   virtual void function1() {};
   virtual void function2() {};
};

class D1: public Base
{
public:
   void function1() override {};
};

class D2: public Base
{
public:
   void function2() override {};
};

int main()
{
   Base* dPtr = new d1();
   dPtr>function1();

   return 0;
}
```



Example of Run-time Polymorphism

(lec13_03_rt_polymorphism.cpp)

Conceptual Modeling: Using a base class like Shape with derived classes like Circle and Square reflects the real-world relationship and hierarchy between these entities.

Specialization: Derived classes represent a specialization of the base class. For example, both Circle and Square are specific types of Shape.

Adding New Functionality : Derived classes can extend the base class by adding new functionality. This could include additional methods and properties that are specific to the derived class

(lec13_04_rt_polymorphism_2.cpp)

Aiming Derived-Class Pointers at Base-Class Objects

Base class pointer is used to point to objects of derived classes.
This allows to use objects of different classes (that derive from a common base class) interchangeably, leveraging the concept of polymorphism.
Important considerations and limitations when pointing derived-class pointers at base-class objects, which is generally not recommended or safe.

Aiming Derived-Class Pointers at Base-Class Objects

Base-Class Pointer to Derived-Class Object: This is a common and safe practice in C++. You can assign the address of a derived-class object to a pointer of the base class.

Allows the base class pointer to access the parts of the derived object that are common with the base class.

```
class Base {};
class Derived : public Base {};

Base* basePtr;
Derived derivedObj;
basePtr = &derivedObj; // Safe and common
```

Aiming Derived-Class Pointers at Base-Class Objects

Derived-Class Pointer to Base-Class Object: This is generally unsafe and not recommended.
The derived-class pointer expects the object it points to, to have all the features of the derived class (including member variables and functions).
However, a base-class object lacks these additional features that the derived class might have added.

```
class Base {};
class Derived : public Base {};

Derived* derivedPtr;
Base baseObj;
derivedPtr = &baseObj; // Unsafe, can lead to undefined behavior
```

Derived-Class Member-Function Calls via Base-Class Pointers

Using a pointer of the base class type to call member functions of a derived class. This is possible and safe under the following conditions: Polymorphism: The member functions being called are virtual in the base class and are overridden in the derived class. This allows the base class pointer to dynamically bind to the derived class's implementation of the function at runtime (dynamic polymorphism). Correct Object Type: The base class pointer must actually point to an object of the derived class or a class further derived from it.

Derived-Class Member-Function Calls via Base-Class Pointers

```
auto derivedPtr = dynamic_cast<Derived*>(basePtr);
  if (derivedPtr) {
     derivedPtr->specificFunction(); // Safe, as we've ensured the pointer actually points to a Derived object
  }
```

dynamic_cast is used for safe type conversion, especially in a hierarchy of classes involving polymorphism. It is used to convert a pointer (or reference) of a base class to a pointer (or reference) of a derived class.

This type of casting is checked at runtime to ensure the safety of the operation, making dynamic_cast safer than other casts like static_cast or reinterpret_cast.

Derived-Class Member-Function Calls via Base-Class Pointers

"Derived-Class Member-Function Calls via Base-Class Pointers”
Referring to the overall process - start with a base-class pointer (basePtr) and use it in a context to call a derived-class-specific member function.
This process involves temporarily converting (downcasting) the base-class pointer to a derived-class pointer to access those derived-specific features safely.
The actual call to the derived-class member function (specificFunction()) is indeed made using the derived-class pointer (derivedPtr
The important part is that this derived-class pointer was obtained by safely downcasting a base-class pointer that was originally pointing to a derived-class object.

Virtual Functions and Virtual Destructors

A virtual function is a member function in a base class that expect to redefine in derived classes.
When 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 function.
Polymorphism: This is the primary mechanism in C++ to achieve runtime polymorphism.
Dynamic Binding: When a virtual function is called through a pointer or a reference, the actual function that gets called is resolved at runtime based on the type of the object pointed to, not the type of the pointer or reference.

Virtual Functions and Virtual Destructors

Virtual destructor ensures that when delete a derived class object through a base class pointer, the derived class's destructor is called before the base class's destructor. This is crucial for proper resource cleanup in classes that inherit from base classes with dynamic memory allocation.

Resource Management: Ensures that destructors of derived classes are called, allowing for proper cleanup of resources allocated by derived classes.

Avoid Memory Leaks: Without a virtual destructor in the base class, only the base class's destructor would be called, potentially leading to resource leaks.

Polymorphism, Virtual Functions and Dynamic Binding

Polymorphism
Allows objects of different classes to be treated as objects of a common superclass. It's enabling the same interface to be used for different underlying forms (data types).
Compile-time Polymorphism (Static Binding): Achieved using function overloading and operator overloading. The function to be called is determined at compile time.
Runtime Polymorphism (Dynamic Binding): Achieved using inheritance, virtual functions, and pointers/references. The function to be called is determined at runtime.

Polymorphism, Virtual Functions and Dynamic Binding

Polymorphism
Allows objects of different classes to be treated as objects of a common superclass. It's enabling the same interface to be used for different underlying forms (data types).
Compile-time Polymorphism (Static Binding): Achieved using function overloading and operator overloading. The function to be called is determined at compile time.
Runtime Polymorphism (Dynamic Binding): Achieved using inheritance, virtual functions, and pointers/references. The function to be called is determined at runtime.

Polymorphism, Virtual Functions and Dynamic Binding

Virtual function is a function in a base class that is declared with the virtual keyword. When a base class reference or pointer points to a derived class object and a virtual function is called, C++ determines at runtime which function to invoke.

VTable (Virtual Table): Under the hood, C++ uses a virtual table to implement dynamic binding. A virtual table is an array of pointers to virtual functions, maintained per class.

VPointer (Virtual Pointer): Each object of a class containing virtual functions has a hidden member, a pointer to the virtual table for that class. This pointer, often called a vptr, is set up automatically by the constructor of the class.

Polymorphism, Virtual Functions and Dynamic Binding

Dynamic binding is the process by which a call to an overridden function is resolved at runtime. This is in contrast to static binding, which resolves the function call at compile time.

At Object Creation: When an object of a class containing virtual functions is created, the compiler sets the vptr of the object to point to the virtual table of that class.

At Function Call: When a virtual function call is made through a base class pointer or reference, the compiler generates code to first look at the vptr of the actual object (which points to the virtual table of the actual class of the object). It then uses the vptr to get to the virtual table of the actual object and invokes the function pointed to by the appropriate entry in the table.automatically by the constructor of the class.

Q & A