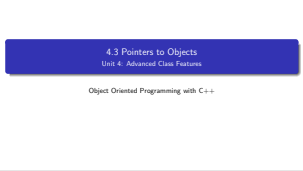
4.3 Pointers to Objects

Unit 4: Advanced Class Features

Object Oriented Programming with C++



Lecture 35: Pointers to Objects

- Unit 4: Advanced Class Features
- Topic 4.3: Pointers to Objects, the 'this' Pointer, and Pointers to Derived Classes

Lecture 35: Pointers to Objects
- Unit 4: Advanced Class Features
- Topic 4.3: Pointers to Objects, the 'this' Pointer, and Pointers to Derived Classes

Welcome back, class, to Lecture 35. Today we are diving into a crucial topic in C++: Pointers to Objects. As we move deeper into object-oriented design, understanding how memory addresses interact with our custom classes is absolutely essential. We will cover basic object pointers, the hidden 'this' pointer, and how pointers interact within inheritance hierarchies.

Learning Objectives

- Declare and instantiate pointers to objects dynamically using new and delete.
- Master the arrow operator (-¿) for member access.
- Grasp the concept of the implicit 'this' pointer and its common use cases.
- Understand the mechanics of using base class pointers to refer to derived class objects.

Learning Objectives
- Declare and instantiate pointers to objects dynamically using new and delete.
- Master the arrow operator (-¿) for member access.
- Grasp the concept of the implicit 'this' pointer and its common use cases.
- Understand the mechanics of using base class pointers to refer to derived class objects.

By the end of this 50-minute session, you should feel comfortable managing objects dynamically via pointers. You will know exactly what the 'this' pointer is and why it exists. Finally, we will set the stage for polymorphism by looking at how base and derived classes interact through pointers, which is the cornerstone of advanced OOP in C++.

Introduction to Object Pointers

- Just like built-in types (int, float), we can create pointers to user-defined classes.
- Object pointers store the memory address where the object's data members reside.
- Crucial for Dynamic Memory Allocation (allocating objects on the heap).
- Syntax: ClassName *ptrName;

Introduction to Object Pointers
- Just like built-in types (int, float), we can create pointers to user-defined classes.
- Object pointers store the memory address where the object's data members reside.
- Crucial for Dynamic Memory Allocation (allocating objects on the heap).
- Syntax: ClassName *ptrName;

We have extensively used pointers with basic data types like integers and characters. The concept is identical for objects. An object pointer simply holds the address where a specific instance of a class is stored in memory. This becomes primarily useful when we want objects to exist beyond the scope of a single function, requiring us to manage their memory manually on the heap.

Dynamic Memory Allocation

- We use 'new' to allocate memory on the heap, and 'delete' to free it.
- Example: Student *sPtr = new Student("Alice", 20); // ... use the object ... delete sPtr; // Crucial for preventing memory leaks

Dynamic Memory Allocation
- We use 'new' to allocate memory on the heap, and 'delete' to free it.
- Example: Student *sPtr = new Student("Alice", 20); // ... use the object ... delete sPtr; // Crucial for preventing memory leaks

Notice how we use the 'new' keyword to dynamically create a Student object. The 'new' operator allocates the memory, calls the constructor, and returns the memory address. 'sPtr' now holds this address. Because we allocated it dynamically, it is the programmer's strict responsibility to free this memory using 'delete sPtr' when it is no longer needed, otherwise our program will suffer from memory leaks.

Member Access: The Arrow Operator (-¿)

- To access members directly via an object instance, we use the dot (.) operator.
- To access members via a pointer to an object, we use the arrow (-¿) operator.
- The expression ptr-¿member is syntactic sugar for (*ptr).member

Member Access: The Arrow Operator (-¿)
- To access members directly via an object instance, we use the dot (.) operator.
- To access members via a pointer to an object, we use the arrow (-¿) operator.
- The expression ptr-¿member is syntactic sugar for (*ptr).member

If you have a standard object instance, you use the dot operator to access its methods. But if you have a pointer to that instance, you must dereference it first. You could write (*sPtr).display(), making sure to use parentheses because the dot operator has higher precedence than the dereference star. However, C++ provides the arrow operator (-¿) which makes the code much cleaner and easier to read: sPtr-¿display().

Arrays of Pointers to Objects

- We can create arrays where each element is a pointer to an object, rather than the object itself.
- Highly useful for managing large collections of objects efficiently.
- Syntax: ClassName *arr[10];

Arrays of Pointers to Objects
- We can create arrays where each element is a pointer to an object, rather than the object itself.
- Highly useful for managing large collections of objects efficiently.
- Syntax: ClassName *arr[10];

Instead of an array of contiguous objects, we can have an array of pointers. This is highly flexible. For example, if we have an array of Employee pointers, we can dynamically allocate only the Employee objects we actually need. This saves memory if the array isn't full, and makes sorting operations much faster since we are only swapping memory addresses, not the heavy objects themselves.

The 'this' Pointer: Concept

- Every object in C++ has access to its own address through a hidden pointer called 'this'.
- The 'this' pointer is an implicit parameter passed to all non-static member functions.
- The compiler automatically passes the address of the calling object to the function behind the scenes.

The 'this' Pointer: Concept
- Every object in C++ has access to its own address through a hidden pointer called 'this'.
- The 'this' pointer is an implicit parameter passed to all non-static member functions.
- The compiler automatically passes the address of the calling object to the function behind the scenes.

Now we move to a fundamental underlying concept: the 'this' pointer. When you call an object's method, like obj.updateData(), how does the method internally know which object's data to operate on? The compiler secretly passes a pointer to the calling object as a hidden first argument to the function. This pointer is formally named 'this'.

Use Case 1: Resolving Shadowing

- Used when local variables or parameters have the exact same name as class attributes.
- class Point { int x, y; public: void setCoordinates(int x, int y) { this-¿x = x; // 'this-¿x' is the attribute, 'x' is the local parameter this-¿y = y; } };

Use Case 1: Resolving Shadowing
- Used when local variables or parameters have the exact same name as class attributes.
- class Point { int x, y; public: void setCoordinates(int x, int y) { this-¿x = x; // 'this-¿x' is the attribute, 'x' is the local parameter this-¿y = y; } };

The most common practical use of 'this' that you will write yourself is resolving name conflicts. In the setCoordinates method, the parameters 'x' and 'y' shadow the class members 'x' and 'y'. If we just wrote x = x, we'd be assigning the parameter to itself. By explicitly writing 'this-¿x', we explicitly tell the compiler we want the member variable of the current object.

Use Case 2: Returning the Current Object

- A function can return a reference to the calling object to allow method chaining.
- Point& setX(int x) { this-¿x = x; return *this; // dereference 'this' to return the object itself }
- Usage: p1.setX(10).setY(20);

Use Case 2: Returning the Current Object
- A function can return a reference to the calling object to allow method chaining.
- Point& setX(int x) { this-¿x = x; return *this; // dereference 'this' to return the object itself }
- Usage: p1.setX(10).setY(20);

Another very powerful use of 'this' is returning the object itself from a method. By returning '*this' by reference, we can chain function calls sequentially on a single line. This is commonly seen in the builder design pattern or heavily used in C++ stream operations, which is why we can write 'cout ¡¡ a ¡¡ b'.

Restrictions on 'this'

- The 'this' pointer is a const pointer. You cannot change what 'this' points to (e.g., this = &otherObject; is illegal).
- Static member functions do NOT have a 'this' pointer.
- Friend functions do NOT have a 'this' pointer.

Restrictions on 'this'
- The 'this' pointer is a const pointer. You cannot change what 'this' points to (e.g., this = &otherObject; is illegal).
- Static member functions do NOT have a 'this' pointer.
- Friend functions do NOT have a 'this' pointer.

It's important to know the limitations. You cannot assign a new address to 'this'—it is bound to the calling object for the duration of the method call. Also, static functions belong to the class blueprint as a whole, not a specific instantiated object, so there is no 'this' pointer available inside them. The same applies to friend functions, as they are not true class members despite having access privileges.

Pointers to Derived Classes

- C++ allows a Base class pointer to point to a Derived class object.
- This is called 'Upcasting'. It is safe and happens implicitly.
- Syntax: Base* bPtr = new Derived();

Pointers to Derived Classes
- C++ allows a Base class pointer to point to a Derived class object.
- This is called 'Upcasting'. It is safe and happens implicitly.
- Syntax: Base* bPtr = new Derived();

We now transition to how pointers work alongside inheritance. A very powerful rule in C++ is that a pointer defined as a base class type can safely hold the address of a derived class object. This makes sense logically in OOP: if a 'Car' inherits from 'Vehicle', a Car 'is-a' Vehicle. Therefore, a Vehicle pointer is perfectly capable of pointing to a Car object.

Member Access via Base Pointers

- Even if a Base pointer points to a Derived object, it can ONLY access members defined in the Base class.
- The compiler determines access based on the pointer's declared type, not the object's actual runtime type.
- This behavior is known as Early Binding (or Static Binding).

Member Access via Base Pointers
- Even if a Base pointer points to a Derived object, it can ONLY access members defined in the Base class.
- The compiler determines access based on the pointer's declared type, not the object's actual runtime type.
- This behavior is known as Early Binding (or Static Binding).

Here is the critical catch to upcasting. If 'Vehicle* vPtr = new Car()', and the Car class has a specific method called 'honkHorn()', vPtr CANNOT call 'honkHorn()'. The compiler only looks at the declared type of the pointer (Vehicle), and the Vehicle class doesn't know anything about 'honkHorn()'. The pointer acts as a restricted lens, only letting you see the Base parts of the Derived object.

Base Pointer Limitations: Example

- class Base { public: void show() { cout ¡¡ "Base"; } };
- class Derived : public Base { public: void show() { cout ¡¡ "Derived"; } };
- Base* ptr = new Derived(); ptr-¿show(); // Outputs "Base"!

Base Pointer Limitations: Example
- class Base { public: void show() { cout ¡¡ "Base"; } };
- class Derived : public Base { public: void show() { cout ¡¡ "Derived"; } };
- Base* ptr = new Derived(); ptr-¿show(); // Outputs "Base"!

Look at this code carefully. The Derived class redefines the 'show' method. Even though 'ptr' genuinely points to a 'Derived' object in memory, calling 'ptr-¿show()' executes the 'Base' class version. Because the pointer is statically typed as Base, the compiler binds the function call at compile-time to Base::show(). This compile-time decision is Early Binding.

Downcasting (Brief Overview)

- Converting a Base class pointer back down to a Derived class pointer.
- Unlike upcasting, downcasting must be done explicitly.
- Inherently dangerous if the base pointer doesn't actually point to the requested derived type in memory.
- Requires dynamic_cast for safety (to be covered in advanced topics).

Downcasting (Brief Overview)
- Converting a Base class pointer back down to a Derived class pointer.
- Unlike upcasting, downcasting must be done explicitly.
- Inherently dangerous if the base pointer doesn't actually point to the requested derived type in memory.
- Requires dynamic_cast for safety (to be covered in advanced topics).

If you absolutely must access Derived-specific methods while holding a Base pointer, you have to cast the pointer back down. This is called downcasting. It requires an explicit cast in code, and it's risky. If you try to downcast a Vehicle pointer to a Car pointer, but the Vehicle pointer was actually pointing to a Motorcycle object, your program will likely crash. We will learn about 'dynamic_cast' later, which safely checks these types at runtime.

Why Use Base Pointers? Preview of Polymorphism

- Enables generic programming: an array of Base pointers can hold heterogeneous objects of various Derived types.
- Function parameters taking a Base *can accept any Derived* object as an argument.
- Combined with 'virtual' functions, this entirely enables Runtime Polymorphism (Late Binding).

Why Use Base Pointers? Preview of Polymorphism
- Enables generic programming: an array of Base pointers can hold heterogeneous objects of various Derived types.
- Function parameters taking a Base can accept any Derived object as an argument.
- Combined with 'virtual' functions, this entirely enables Runtime Polymorphism (Late Binding).

You might be asking: why bother pointing to a Derived object with a Base pointer if we lose access to the Derived methods? The true power comes when we have an array of Base pointers. This array can hold cars, trucks, and motorcycles all at once! In the next lecture, we will introduce 'virtual' functions. Virtual functions fix the early binding limitation, allowing our Base pointers to execute the correct Derived methods dynamically at runtime. This is true Polymorphism.

Summary & Next Steps

- Object pointers manage memory dynamically and use '-¿' for access.
- The 'this' pointer resolves scoping issues and allows method chaining.
- Base pointers safely hold Derived addresses, but restrict access to Base members under early binding.

Summary & Next Steps
- Object pointers manage memory dynamically and use '-¿' for access.
- The 'this' pointer resolves scoping issues and allows method chaining.
- Base pointers safely hold Derived addresses, but restrict access to Base members under early binding.

To summarize today's lesson: pointers are just as powerful with custom objects as they are with primitives. The 'this' pointer is an elegant, hidden mechanism for instance-awareness inside methods. And pointer upcasting is the absolute bedrock of C++ inheritance hierarchies. Please review these examples closely, as they are mandatory prerequisites for our next topic on virtual functions. Are there any final questions before we wrap up?