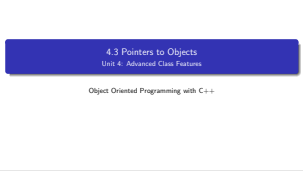
4.3 Pointers to Objects

Unit 4: Advanced Class Features

Object Oriented Programming with C++



Lecture 31: Pointers to Objects

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

Lecture 31: Pointers to Objects
- Unit 4: Advanced Class Features
- Topic 4.3: Pointers to Objects, this Pointer, Pointers to Derived Classes

Welcome everyone to Lecture 31. Today we are diving into a crucial topic in advanced C++ programming: Pointers to Objects. We will explore how memory management extends to user-defined types, the hidden 'this' pointer, and how pointers interact with inheritance hierarchies.

Introduction to Object Pointers

- Just like pointers to fundamental data types (int, float), we can have pointers to objects of a class.
- An object pointer holds the memory address of an object.
- Declaration syntax: ClassName *ptr;
- Assignment: ptr = &objectName;

Introduction to Object Pointers
- Just like pointers to fundamental data types (int, float), we can have pointers to objects of a class.
- An object pointer holds the memory address of an object.
- Declaration syntax: ClassName *ptr;
- Assignment: ptr = &objectName;

We already know how pointers work with primitive data types. In C++, classes are custom data types. Therefore, we can create pointers that point to instances of these classes, known as objects. This allows for dynamic memory allocation and efficient manipulation of complex data structures without copying large amounts of data.

Accessing Members via Pointers

- The dot operator (.) is used to access members via an object name.
- The arrow operator (-¿) is used to access members via a pointer to an object.
- ptr-¿memberName is exactly equivalent to (*ptr).memberName
- This applies to both data members and member functions.

Accessing Members via Pointers
- The dot operator (.) is used to access members via an object name.
- The arrow operator (-¿) is used to access members via a pointer to an object.
- ptr-¿memberName is exactly equivalent to (*ptr).memberName
- This applies to both data members and member functions.

When you have an object, you use the dot operator. But when you have a pointer to an object, you must dereference it first. C++ provides a shorthand for this: the arrow operator. Writing ptr-arrow-member is much cleaner and more readable than parenthesis-asterisk-ptr-parenthesis-dot-member. This is the standard way to interact with dynamically allocated objects.

Code Example: Basic Object Pointer

- class Box {
- public:
- int volume() { return length *width* height; }
- int length, width, height;
- };
- Box b1;
- b1.length = 5; b1.width = 5; b1.height = 5;
- Box *ptr = &b1;
- cout ¡¡ ptr-¿volume(); // Outputs 125

Code Example: Basic Object Pointer
- class Box {
- public:
- int volume() { return length *width* height; }
- int length, width, height;
- };
- Box b1;
- b1.length = 5; b1.width = 5; b1.height = 5;
- Box *ptr = &b1;
- cout ¡¡ ptr-¿volume(); // Outputs 125

Here is a simple example. We have a Box class. We create an object b1 and set its dimensions. Then, we declare a pointer 'ptr' of type 'Box' and assign it the address of b1. Notice how we use ptr-¿volume() to call the method. It executes the volume function on the b1 object, resulting in 125.

Dynamic Memory Allocation for Objects

- Pointers are heavily used with the 'new' and 'delete' operators.
- ClassName *ptr = new ClassName(); allocates memory dynamically.
- delete ptr; frees the memory.
- Crucial for creating objects whose lifetimes extend beyond the scope they were created in.

Dynamic Memory Allocation for Objects
- Pointers are heavily used with the 'new' and 'delete' operators.
- ClassName *ptr = new ClassName(); allocates memory dynamically.
- delete ptr; frees the memory.
- Crucial for creating objects whose lifetimes extend beyond the scope they were created in.

A major use case for object pointers is dynamic allocation. By using 'new', we allocate the object on the heap rather than the stack. This is essential when we don't know how many objects we need at compile time, or when objects need to persist after a function returns. Always remember to use 'delete' to prevent memory leaks.

The 'this' Pointer: Concept

- Every object in C++ has access to its own address through an important pointer called 'this'.
- The 'this' pointer is an implicit parameter to all non-static member function calls.
- Inside a member function, 'this' may be used to refer to the invoking object.
- It is a constant pointer: ClassName* const this;

The 'this' Pointer: Concept
- Every object in C++ has access to its own address through an important pointer called 'this'.
- The 'this' pointer is an implicit parameter to all non-static member function calls.
- Inside a member function, 'this' may be used to refer to the invoking object.
- It is a constant pointer: ClassName* const this;

Now let's talk about the 'this' pointer. It's a special, hidden pointer available inside every non-static member function. When you call a method on an object, the compiler secretly passes the address of that object to the method as the 'this' pointer. It's the mechanism that allows a function to know which object's data to operate on.

Use Cases of 'this' Pointer: Name Hiding

- When local variables (like function parameters) have the same name as class member variables, the local variable 'hides' the member.
- The 'this' pointer can be used to resolve this ambiguity.
- this-¿variableName explicitly refers to the class member.

Use Cases of 'this' Pointer: Name Hiding
- When local variables (like function parameters) have the same name as class member variables, the local variable 'hides' the member.
- The 'this' pointer can be used to resolve this ambiguity.
- this-¿variableName explicitly refers to the class member.

One of the most common explicit uses of the 'this' pointer is resolving name conflicts. If your constructor or setter method has a parameter named exactly the same as a class attribute, the compiler will default to the local parameter. To tell the compiler you mean the class attribute, you prefix it with 'this-¿'.

Code Example: Resolving Shadowing

- class Employee {
- private:
- int id;
- public:
- Employee(int id) {
- this-¿id = id; // Resolves ambiguity
- }
- };

Code Example: Resolving Shadowing
- class Employee {
- private:
- int id;
- public:
- Employee(int id) {
- this-¿id = id; // Resolves ambiguity
- }
- };

In this snippet, the Employee constructor takes a parameter 'id'. The class also has a member 'id'. Writing id = id would just assign the parameter to itself, leaving the member uninitialized. By writing this-¿id = id, we clearly state: assign the parameter 'id' to the object's member 'id'.

Use Cases of 'this' Pointer: Returning the Object

- A member function can return a reference to the invoking object using '*this'.
- Useful for implementing method chaining.
- Function signature must return a reference: ClassName& functionName()

Use Cases of 'this' Pointer: Returning the Object
- A member function can return a reference to the invoking object using '*this'.
- Useful for implementing method chaining.
- Function signature must return a reference: ClassName& functionName()

Another powerful use of 'this' is returning the current object from a method. Since 'this' is a pointer, *'this' is the actual object itself. If a method returns a reference to the class (ClassName&), returning* 'this' allows us to call multiple methods on the same object in a single statement.

Code Example: Method Chaining

- class Config {
- public:
- Config& setWidth(int w) { width = w; return *this; }
- Config& setHeight(int h) { height = h; return *this; }
- int width, height;
- };
- Config c;
- c.setWidth(100).setHeight(200);

Code Example: Method Chaining
class Config {
public:
Config& setWidth(int w) { width = w; return *this; }
Config& setHeight(int h) { height = h; return *this; }
int width, height;
};
Config c;
c.setWidth(100).setHeight(200);

Here we see method chaining in action. The setWidth and setHeight methods both return *this as a reference. This means c.setWidth(100) modifies the object and then returns it, allowing .setHeight(200) to be called immediately on the result. This pattern is very common in modern C++ design.

Pointers to Derived Classes

- A base class pointer can point to an object of any of its derived classes.
- This is a cornerstone of runtime polymorphism in C++.
- BaseClass *basePtr = new DerivedClass();
- However, a derived class pointer cannot point to a base class object.

Pointers to Derived Classes
- A base class pointer can point to an object of any of its derived classes.
- This is a cornerstone of runtime polymorphism in C++.
- BaseClass *basePtr = new DerivedClass();
- However, a derived class pointer cannot point to a base class object.

Moving on to inheritance, we encounter a crucial rule: a pointer of a base class type can point to an object of a derived class. Because a derived class 'is-a' base class, it contains all the components of the base class, making this operation safe. The reverse, however, is not true, as a base object lacks the derived features.

Access Restrictions with Base Pointers

- Even if a base pointer points to a derived object, it can only access members defined in the base class.
- It cannot access newly added members specific to the derived class.
- The compiler only looks at the type of the pointer, not the type of the object it points to.

Access Restrictions with Base Pointers
- Even if a base pointer points to a derived object, it can only access members defined in the base class.
- It cannot access newly added members specific to the derived class.
- The compiler only looks at the type of the pointer, not the type of the object it points to.

It's important to understand the limitation here. If you use a base class pointer to point to a derived object, the compiler still treats it as a base pointer. It will only allow you to call functions or access variables that are defined in the base class. To access derived-specific members, you need virtual functions or casting.

Code Example: Base Class Pointer

- class Base { public: void show() { cout ¡¡ "Base"; } };
- class Derived : public Base { public: void showDerived() {} };
- Derived d;
- Base *bptr = &d; // Valid
- bptr-¿show(); // Valid: Calls Base::show
- // bptr-¿showDerived(); // ERROR: Base has no showDerived

Code Example: Base Class Pointer
- class Base { public: void show() { cout ¡¡ "Base"; } };
- class Derived : public Base { public: void showDerived() {} };
- Derived d;
- Base *bptr = &d; // Valid
- bptr-¿show(); // Valid: Calls Base::show
- // bptr-¿showDerived(); // ERROR: Base has no showDerived

Look at this example. The pointer bptr is of type Base*, but it holds the address of a Derived object 'd'. We can successfully call show() because it exists in Base. But if we try to call showDerived(), the compiler throws an error, because the bptr type (Base) doesn't know about showDerived().

The Need for Virtual Functions (Preview)

- How do we make the base pointer act smartly and call the derived version of a function?
- We use 'virtual' functions in the base class.
- When a function is virtual, the compiler looks at the actual object type at runtime, not the pointer type.
- This is known as Dynamic Binding or Late Binding.

The Need for Virtual Functions (Preview)
- How do we make the base pointer act smartly and call the derived version of a function?
- We use 'virtual' functions in the base class.
- When a function is virtual, the compiler looks at the actual object type at runtime, not the pointer type.
- This is known as Dynamic Binding or Late Binding.

This access restriction leads us directly to the concept of polymorphism. If a base class function is declared as 'virtual', calling it via a base pointer to a derived object will execute the derived class's overridden version. This is dynamic binding, which we will explore fully in the upcoming lectures.

Lecture Summary

- Object pointers store the address of instances and use '-¿' for access.
- The 'this' pointer is an implicit reference to the calling object, useful for shadowing and chaining.
- Base class pointers can point to derived class objects, enabling polymorphic behavior.
- Pointer type determines member accessibility at compile time.

Lecture Summary
- Object pointers store the address of instances and use '-¿' for access.
- The 'this' pointer is an implicit reference to the calling object, useful for shadowing and chaining.
- Base class pointers can point to derived class objects, enabling polymorphic behavior.
- Pointer type determines member accessibility at compile time.

To summarize, today we learned how pointers interact with objects. We demystified the 'this' pointer and saw its practical applications. We also introduced the concept of pointing base class pointers to derived objects, setting the stage for runtime polymorphism. Mastering these pointer behaviors is critical for advanced C++ development.