Ch 5: Runtime Polymorphism

CSCI 330

Chapter Overview:

Runtime Polymorphism: dynamic dispatch through inheritance and virtual functions

Key concepts (in context of polymorphism):

- Class hierarchies
- virtual functions
- **override** and **final** keywords
- Abstract base classes
- Destructors and slicing

What is Polymorphism

- Polymorphism: code you write once and reuse with different types.
- Approaches: Compile-time vs Runtime polymorphism
  - Compile-time: templates, auto, overloads
  - Runtime: function dispatch depends on object type at runtime
- Use case
  - Writing flexible APIs for diverse object types

Inheritance and Virtual Functions

- Base class defines interface using **virtual** functions
- Derived class overrides functionality

```
struct Animal {
 virtual void speak() const {
   std::puts("Animal noise");
 }
};

struct Dog : Animal {
 void speak() const override {
   std::puts("Woof!");
 }
};
```

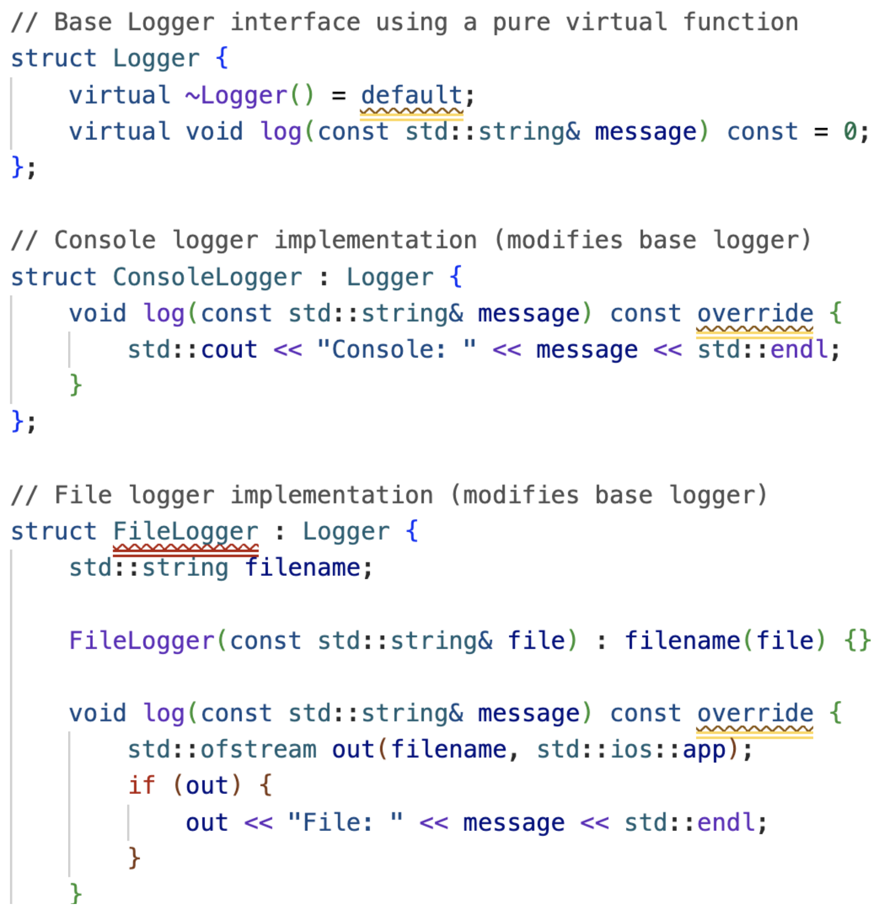
- Dynamic dispatch with Animal* a = new Dog(); a->speak();

logger.cpp example (download from Canvas)

Here’s a better example than 5.1, 5.2 in the book
Note:

- the base class cannot be implemented as is,
- To implement, the function accepts a **reference to the base class** Logger.
- This is **runtime polymorphism**: behavior is **determined dynamically** based on the actual type of object, not the static type.

Download full file to explore.



```cpp
// Base Logger interface using a pure virtual function
struct Logger {
    virtual ~Logger() = default;
    virtual void log(const std::string& message) const = 0;
};

// Console logger implementation (modifies base logger)
struct ConsoleLogger : Logger {
    void log(const std::string& message) const override {
        std::cout << "Console: " << message << std::endl;
    }
};

// File logger implementation (modifies base logger)
struct FileLogger : Logger {
    std::string filename;

    FileLogger(const std::string& file) : filename(file) {}

    void log(const std::string& message) const override {
        std::ofstream out(filename, std::ios::app);
        if (out) {
            out << "File: " << message << std::endl;
        }
    }
```

The override and final keywords

- override: tells compiler you intend to override a base method
- final: prevents further overrides
- Helps catch mistakes like mismatched signatures
- Example:

```
struct Base {
 virtual void foo();
};
struct Derived : Base {
 void foo() override final;
};
```

Virtual Destructors

- Always make base class destructors **virtual** if deleting via base pointer
- Prevents undefined behavior and resource leaks
- Example:

```
struct Base {
 virtual ~Base() = default;
};
```

Abstract Base Classes (Pure Virtual Functions)

- Force derived classes to implement specific behavior
- Syntax: virtual void f() = 0;
- Cannot instantiate abstract class
- Example:

```
struct Shape {
  virtual void draw() const = 0;
};
```

Object Slicing

- Occurs when assigning a derived object to a base object by value
- Only base part is copied; derived-specific data is “sliced off”
- Avoid slicing: use references or pointers for polymorphism
- Example:

```
Dog d;
Animal a = d; // slicing!
```

Dynamic Dispatch Tradeoffs

Pros:

- Extensible APIs
- Decouples interface from implementation

Cons:

- Indirection via vtable
- cannot inline calls
- Slightly slower than static dispatch

Alternatives to Inheritance

- Prefer composition or templates when possible
- User runtime polymorphism when:
  - You need dynamic behavior
  - You don’t know all derived types at compile time