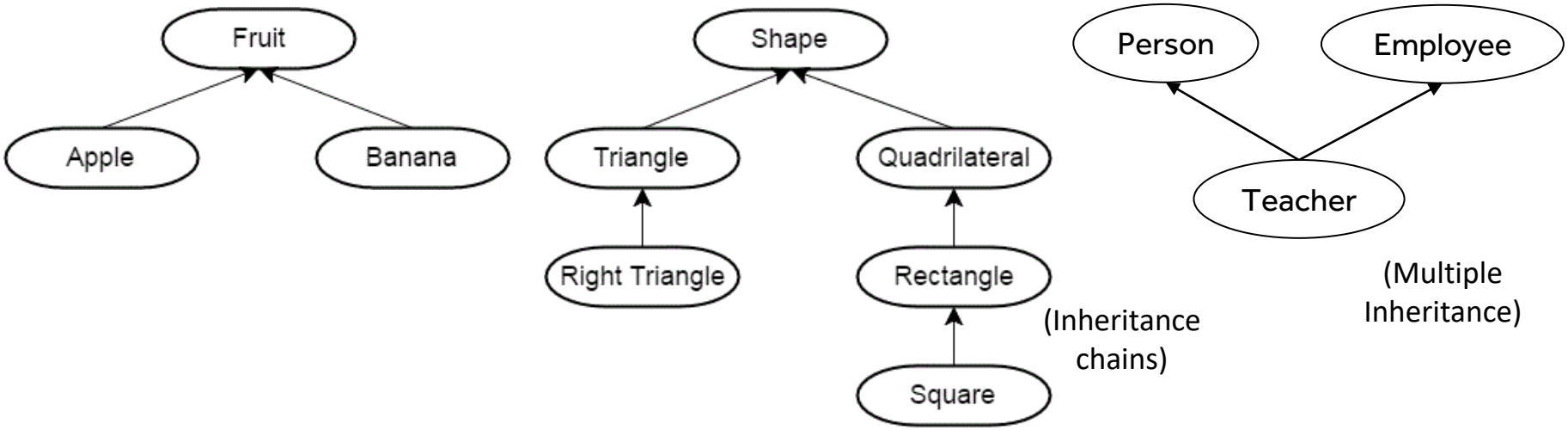
Ch9: Inheritance & Virtual Functions

Inheritance

Inheritance

Inheritance in C++ takes place between classes. In an inheritance (is-a) relationship, the class being inherited from is called the **parent class**, **base class**, or **superclass**, and the class(es) doing the inheriting is called the **child class**, **derived class**, or **subclass**.



A derived class inherits both member functions and data members (or member variables) from the parent (subject to some access restrictions), so we do not have to redefine them. These variables and functions become members of the derived class, in addition to their own members that are specific to that derived class.

Constructors & Initialization of Derived Classes

```
#include <iostream>
class Base {
public:
   Base(int id=0) : m_id{ id } {}
   int getId() const { return m_id; }

private:
   int m_id {};
};
class Derived: public Base {
public:
   // Call Base(int) constructor in the Derived constructor
   Derived(double cost=0.0, int id=0) : Base{ id } , m_cost{ cost } {}
   double getCost() const { return m_cost; }

private:
   double m_cost;
};
int main() {
   Derived derived{ 1.3, 5 }; // use Derived(double, int) constructor
   std::cout << "Id: " << derived.getId() << '\n';
   std::cout << "Cost: " << derived.getCost() << '\n';
}
```

The derived classes can not access private members of the base class directly. Derived classes will need to use public member functions of the base to access private members of the base class.

Order of Construction for Inheritance Chains

```
#include <iostream>
class A {
public:
   A(int a) {
      std::cout << "A: " << a << '\n';
   }
};
class B: public A {
public:
   B(int a, double b) : A{ a }  {
      std::cout << "B: " << b << '\n';
   }
};
class C: public B {
public:
   C(int a, double b, char c) : B{ a, b } {
      std::cout << "C: " << c << '\n';
   }
};
int main() {
   C c{ 5, 4.3, 'R' };
}
```

C++ constructs derived classes in phases, starting with the most-base class (at the top of the inheritance tree) and finishing with the most-child class (at the bottom of the inheritance tree). As each class is constructed, the appropriate constructor from that class is called to initialize that part of the class.

Inheritance & Access Specifiers

```
#include <iostream>
class Base {
public:
   Base(int id = 0, double value = 0.0) : m_id{ id }, m_value{ value } { }
   int getId() const { return m_id; }
protected:
   double m_value {};
private:
   int m_id {};
};
class Derived: public Base {
public:
   // Base constructor is called here
   Derived(double cost = 0.0, int id = 0, double value = 0.0) : Base{ id, value } , m_cost{ cost } {}
   double getCost() const { return m_cost; }
   double getValue() const { return m_cost + m_value; }
private:
   double m_cost{};
};
int main() {
   Derived derived{ 1.3, 5, 10.0};
   std::cout << "Id: " << derived.getId() << '\n';
   std::cout << "Cost: " << derived.getCost() << '\n';
   std::cout << "Value: " << derived.getValue() << '\n';
}
```

The **protected access specifier** allows the class that the member belongs to, its friends, and its derived classes to access the protected members. However, protected members are not accessible from outside the class.

Note: Favor private members over protected members.

Different Kinds of Inheritance: public, protected, private

```
// Inherit from Base publicly
class Pub: public Base {
};

// Inherit from Base protectedly
class Pro: protected Base {
};

// Inherit from Base privately
class Pri: private Base {
};

class Def: Base // Defaults to private inheritance
{};
```

With public inheritance, inherited **public** members stay **public**, inherited **protected** members stay **protected**, and private members stay inaccessible.

With protected inheritance, the **public** and **protected** members become **protected**, and private members stay inaccessible.

With private inheritance, the **public** and **protected** members become **private**, and private members stay inaccessible.

Note: A class (and its friends) can always access its own non-inherited members. The access specifiers only affect whether outsiders and derived classes can access those members.
Note: When derived classes inherit members, those members may change access specifiers in the derived class. It only affects whether outsiders and classes derived from the derived class can access those inherited members.
Best Practice: Use public inheritance unless you have a specific reason to do otherwise.

Multiple Inheritance

```
#include <string>
#include <string_view>

class Person {
private:
   std::string m_name{};
   int m_age{};
public:
   Person(std::string_view name, int age)
      : m_name{ name }, m_age{ age } {}
   const std::string& getName() const { return m_name; }
   int getAge() const { return m_age; }
};

class Employee {
private:
   std::string m_employer{};
   double m_wage{};
public:
   Employee(std::string_view employer, double wage)
      : m_employer{ employer }, m_wage{ wage } {}
   const std::string& getEmployer() const { return m_employer; }
   double getWage() const { return m_wage; }
};
```

```
// Teacher publicly inherits Person and Employee
class Teacher : public Person, public Employee {
private:
   int m_teachesGrade{};
public:
   Teacher(std::string_view name, int age, std::string_view employer,
           double wage, int teachesGrade)
      : Person{ name, age }, Employee{ employer, wage },
        m_teachesGrade{ teachesGrade } {}
};

int main() {
   Teacher t{ "Mary", 45, "Boo", 14.3, 8 };
}
```

Multiple Inheritance: Mixins

```cpp
#include <string>

struct Point2D {
    int x{};
    int y{};
};

// mixin Box class
class Box {
public:
    void setTopLeft(Point2D point) { m_topLeft = point; }
    void setBottomRight(Point2D point) { m_bottomRight = point; }
private:
    Point2D m_topLeft{};
    Point2D m_bottomRight{};
};

// mixin Label class
class Label {
public:
    void setText(const std::string_view str) { m_text = str; }
    void setFontSize(int fontSize) { m_fontSize = fontSize; }
private:
    std::string m_text{};
    int m_fontSize{};
};
```

```cpp
// mixin Tooltip class
class Tooltip {
public:
    void setText(const std::string_view str) { m_text = str; }
private:
    std::string m_text{};
};

// Button using three mixins
class Button : public Box, public Label, public Tooltip {};

int main() {
    // using :: scope resolution prefixes are not required but recommended
    Button button{};
    button.Box::setTopLeft({ 1, 1 });
    button.Box::setBottomRight({ 10, 10 });
    button.Label::setText("Submit");
    button.Label::setFontSize(6);
    button.Tooltip::setText("Submit the form to the server");
}
```

Calling Inherited Functions

```
class Base {
public:
   Base(int id = 0, double value = 0.0) : m_id{ id }, m_value{ value } { }
   int getId() const { return m_id; }
   void identify() const { std::cout << "Original Functionality (Base).\n"; }
   void print(int x)    { std::cout << "Base::print(int)\n"; }
   void print(double x) { std::cout << "Base::print(double)\n"; }
protected:
   double m_value {};
private:
   int m_id {};
};
```

```
class Derived: public Base {
public:
   Derived(double cost = 0.0, int id = 0, double value = 0.0) : Base{ id, value } , m_cost{ cost } {}
   double getCost() const { return m_cost; }
   double getValue() const { return m_cost + m_value; }
   // Adding to existing functionality
   void identify() const {
      std::cout << "More Functionality! (Derived)\n";
      Base::identify();
   }
   using Base::print; // Adding all Base::print() functions eligible for overload resolution
   void print(double x) { std::cout << "Derived::print(double)"; }
private:
   double m_cost{};
};
```

When a member function (like print) is called on a derived class object, the compiler first looks to see if any function with that name exists in the derived class. If so, the function overload resolution process is used to determine whether there is a match in the derived class. If not, the compiler walks up the inheritance chain, checking each parent class in turn in the same way to find a match.

Now, by using "**using** Base::print;" we are telling the compiler that all Base functions named print should be visible in Derived, which will cause them to be eligible all together for overload resolution to find the best match.

Calling Inherited Functions (cont.)

```
int main() {
    Base base {};
    base.identify();

    Derived derived{ 1.3, 5, 10.0};
    std::cout << "Id: " << derived.getId() << '\n';
    std::cout << "Cost: " << derived.getCost() << '\n';
    std::cout << "Value: " << derived.getValue() << '\n';

    derived.identify();

    derived.print(5); // calls Base::print(int), which is the best matching function visible in Derived

    // or you can always specify which one to call:
    derived.Base::print(5.0);
    derived.Derived::print(5.0);
}
```

Deleting Functions in Derived Class

```
#include <iostream>
class Base {
private:
   int m_value {};
public:
   Base(int value) : m_value { value } {}
   int getValue() const { return m_value; }
};

class Derived : public Base {
public:
   Derived(int value) : Base { value } {}
   int getValue() const = delete; // mark this function as inaccessible
};

int main() {
   Derived derived { 7 };
   // The following won't work because getValue() has been deleted!
   //  std::cout << derived.getValue();

   // We can still call the Base::getValue() function directly
   std::cout << derived.Base::getValue();
}
```

You can mark member functions as deleted in the derived class, which ensures they cannot be called at all through a derived object.

Preventing Inheritance of a Class

When final is used with a class, it prevents other classes from inheriting from it.

```
class Base final {
   // Class members
};

// This will cause a compilation error
class Derived : public Base {
   // Class members
};
```

Note: If you do not intend your class to be inherited from, mark your class as final. This will prevent other classes from inheriting from it in the first place, without imposing any other use restrictions on the class itself.

Virtual Function and Polymorphism

Virtual Function and Polymorphism

A **Virtual Function** is a special type of member function (declared in the base class using the virtual keyword) that you expect to be redefined (overridden) in derived classes and, when called, resolves to the most-derived version of the function for the actual type of the object being referenced or pointed to (rather than the type of the reference or pointer itself).

- The function in the derived class must have the same signature (name, parameter types, and whether it is const) and also return type as the base version of the function. Such functions are called overrides.

Polymorphism refers to the ability of an entity to have multiple forms (the term "polymorphism" literally means "many forms"):

- **Compile-time Polymorphism** (static binding) resolves by the compiler, and it includes function overload resolution and template resolution.
- **Runtime Polymorphism** (or dynamic binding) resolves at runtime, and it includes virtual function resolution.

❖ When you use a **virtual function**, you enable **runtime polymorphism**, which allows you to call derived class methods through a base class pointer or reference.

Virtual Function and Polymorphism

- Use the virtual keyword on virtual functions in a base class.
- Use the override specifier (but not the virtual keyword) on override functions in derived classes.

```
#include <iostream>

class Base {
public:
   virtual void show() {
      std::cout << "Base class show function called." << std::endl;
   }
   void display() {
      std::cout << "Base class display function called." << std::endl;
   }
};

class Derived : public Base {
public:
   void show() override {
      std::cout << "Derived class show function called." << std::endl;
   }
   void display() {
      std::cout << "Derived class display function called." << std::endl;
   }
};
```

```
int main() {

   // ==== Using References (Recommended) ====
   Derived derivedObj1;
   Base& baseRef {derivedObj1};

   // Virtual function, binded at runtime
   baseRef.show();

   // Non-virtual function, binded at compile time
   baseRef.display();


   // ==== Using Pointers ====
   Derived derivedObj2;
   Base* basePtr {&derivedObj2};

   // Virtual function, binded at runtime
   basePtr->show();

   // Non-virtual function, binded at compile time
   basePtr->display();
}
```

Preventing Overriding of a Virtual Function

When final is used with a virtual member function, it prevents derived classes from overriding that function.

```
class Base {
public:
    virtual void someFunction() final {
        // Function implementation
    }
};

class Derived : public Base {
public:
    // This will cause a compilation error
    void someFunction() override {
        // Function implementation
    }
};
```

Virtual Destructors

Using a **Virtual Destructor** in a Base class is crucial when you are working with polymorphism and dynamic memory allocation (new/delete) to ensuring proper cleanup and avoid resource leaks and undefined behavior. It ensures that the Derived class destructor is called properly when an object is deleted through a base class pointer or reference.

```
class Base {
public:
    virtual void show() {
        std::cout << "Base class show function called." << std::endl;
    }

    void display() {
        std::cout << "Base class display function called." << std::endl;
    }

    // Virtual destructor
    virtual ~Base() {
        std::cout << "Base class destructor called." << std::endl;
    }
    // or use Default virtual destructor
    // virtual ~Base() = default;
};
```

```
class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived class show function called." << std::endl;
    }

    void display() {
        std::cout << "Derived class display function called." << std::endl;
    }

    // Destructor
    ~Derived() {
        std::cout << "Derived class destructor called." << std::endl;
    }
    // or use Default destructor
    // ~Derived() = default;
};
```

Pure Virtual Function, Abstract Classes, and Interface Classes

Pure Virtual Function & Abstract Classes

C++ allows you to create a special kind of virtual function called a **Pure Virtual Function** that has no body in the base class and is made by adding "= 0" to the end of the virtual function prototype.

```
class Base {
public:
   virtual void pureVirtualFunction() = 0; // Pure virtual function
};
```

- A pure virtual function simply acts as a placeholder that is meant to be redefined (overridden) by derived classes.
- A class containing at least one pure virtual function is called an **Abstract Class** and can not be instantiated.
- A class that inherits pure virtual functions must concretely define them or it will also be considered abstract (and can not be instantiated).

Pure Virtual Function & Abstract Classes

```
#include <iostream>
class Base { // This is an abstract base class
public:
    Base(int number) : m_number{ number } {}
    int getNumber() const { return m_number; }
    virtual void pureVirtualFunction() = 0; // Pure virtual function
    virtual ~Base() = default;
protected:
    int m_number {};
};
class Derived : public Base {
public:
    Derived(int number):Base{number}{}
    void pureVirtualFunction() override {
        std::cout << "Implementation of pure virtual function in Derived class" << std::endl;
    }
};
int main() {
    // Base b; // Error: cannot declare variable 'b' to be of abstract type 'Base'
    Derived d{2};
    d.pureVirtualFunction();
    std::cout << d.getNumber() << std::endl;
    // or
    Derived d2{2};
    Base& baseRef {d2};
    d2.pureVirtualFunction();
}
```

Interface Classes

An **Interface Class** is an abstract class that has no data members, and all of its member functions are pure virtual.

- It is designed to be used as a Base class but is not intended to be instantiated on its own.
- Interfaces are useful when you want to define the functionality that derived classes must implement but leave the details of how the derived class implements that functionality entirely up to the derived class. By using interface classes, you can ensure that different classes adhere to a common interface, making your code more modular and easier to maintain.
- Interface classes are often named beginning with a capital **I**.

Interface Classes

```
#include <iostream>
class Interface {
public:
    // Pure virtual function
    virtual void someFunction() = 0;
    virtual ~Interface() {}
};
// Derived class that implements the interface
class Implementation : public Interface {
public:
    // Implementing the pure virtual function
    void someFunction() override {
        std::cout << "Implementation of someFunction" << std::endl;
    }
};
int main() {
    // Interface obj; // This will give a compilation error as Interface cannot be instantiated
    Implementation obj;
    obj.someFunction();
}
```

Interface Classes

```
class IShape {
public:
   virtual ~IShape() {}  // Virtual destructor
   virtual void draw() = 0;  // Pure virtual function
   virtual double area() = 0;  // Pure virtual function
};
```

```
class Square : public IShape {
private:
   double side;
public:
   Square(double s) : side(s) {}

   void draw() override {
      // Implementation of draw for Square
      std::cout << "Drawing Square" << std::endl;
   }

   double area() override {
      // Implementation of area for Square
      return side * side;
   }
};
```

```
class Circle : public IShape {
private:
   double radius;
public:
   Circle(double r) : radius(r) {}

   void draw() override {
      // Implementation of draw for Circle
      std::cout << "Drawing Circle" << std::endl;
   }

   double area() override {
      // Implementation of area for Circle
      return 3.14159 * radius * radius;
   }
};
```