MTH 4300/4299: Programming and Computer Science II

Lecture 16: Inheritance and Polymorphism

Jaime Abbariao

▒▒▒ The Problem: Code Duplication

Imagine we need two classes — `Student` and `Employee`:

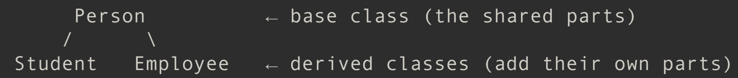
```
class Student {
private:
  std::string name;
  int age;
  double gpa;
public:
  Student(std::string n, int a, double g)
    : name(n), age(a), gpa(g) {}
  std::string get_name() const { return name; }
  int get_age() const { return age; }
  double get_gpa() const { return gpa; }
};

class Employee {
private:
  std::string name;
  int age;
  std::string company;
public:
  Employee(std::string n, int a, std::string c)
    : name(n), age(a), company(c) {}
  std::string get_name() const { return name; }
  int get_age() const { return age; }
  std::string get_company() const { return company; }
};
```

`name`, `age`, `get_name()`, `get_age()` are duplicated. What if we could write the shared parts once?

What is Inheritance?

Inheritance lets a new class (the derived class) reuse the members of an existing class (the base class).



The derived class inherits all the data and functions of the base class, then adds its own.

Terminology (these all mean the same thing):

Relationship	Names
The class being inherited from	base class, parent class, superclass
The class that inherits	derived class, child class, subclass

The key idea: a `Student` is a `Person`. An `Employee` is a `Person`.

Basic Syntax

First, define the base class:

```cpp
class Person {
private:
  std::string name;
  int age;
public:
  Person(std::string n, int a) : name(n), age(a) {}
  std::string get_name() const { return name; }
  int get_age() const { return age; }
};
```

Then define the derived class using `: public BaseClass`:

```cpp
class Student : public Person {
private:
  double gpa;
public:
  Student(std::string n, int a, double g)
    : Person(n, a), gpa(g) {}
  double get_gpa() const { return gpa; }
};
```

`Student` now has everything `Person` has (`name`, `age`, `get_name()`, `get_age()`), plus its own `gpa` and `get_gpa()`.

Using the Derived Class

```
int main() {
  Student s("Alice", 20, 3.8);

  // Inherited from Person:
  std::cout << s.get_name() << std::endl;  // Alice
  std::cout << s.get_age() << std::endl;   // 20

  // Defined in Student:
  std::cout << s.get_gpa() << std::endl;   // 3.8

  return 0;
}
```

From the outside, `Student` looks like it has all three getters. The caller doesn't need to know which ones are inherited and which are defined in `Student`.

Access Specifiers and Inheritance

A derived class inherits all members from the base class, but it cannot directly access `private` members. This is where `protected` comes in.

Access Specifier	Outside code	Derived class	Class itself
`public`	Yes	Yes	Yes
`protected`	No	Yes	Yes
`private`	No	No	Yes

- `public` — anyone can access
- `protected` — hidden from the outside world, but visible to derived classes
- `private` — only the class itself can access (not even derived classes)

Rule of thumb: use `protected` for members that derived classes need to access directly. Use `private` with public getters/setters if you want tighter control.

```
class Person {
private:
  std::string name;    // only Person can access directly
protected:
  int age;             // Person and derived classes can access
public:
  Person(std::string n, int a) : name(n), age(a) {}
  std::string get_name() const { return name; }
};

class Student : public Person {
public:
  Student(std::string n, int a, double g)
    : Person(n, a), gpa(g) {}

  void print_info() const {
    // std::cout << name;   // ERROR — name is private in Person
    std::cout << get_name();  // OK — get_name() is public
    std::cout << ", age " << age;  // OK — age is protected
    std::cout << ", GPA " << gpa << std::endl;
  }
private:
  double gpa;
};
```

Constructors and Inheritance

A derived class must initialize the base class part. You do this by calling the base constructor in the member initializer list:

```
Student(std::string n, int a, double g)
  : Person(n, a), gpa(g) {}
//  ^^^^^^^^^^^   ^^^^^^
//  base part     derived part
```

What if you forget to call the base constructor?

```
Student(std::string n, int a, double g)
  : gpa(g) {}  // no call to Person(n, a)!
```

The compiler will try to call `Person()` — the default constructor (no arguments). If `Person` doesn't have one, you get a compiler error:

```
error: no matching function for call to 'Person::Person()'
```

Takeaway: always explicitly call the base constructor when the base class requires arguments.

Construction and Destruction Order

When you create a derived object, the base class is constructed first, then the derived class. Destruction happens in reverse order.

```
class Person {
public:
  Person(std::string n, int a) : name(n), age(a) {
    std::cout << "Person constructor" << std::endl;
  }
  ~Person() {
    std::cout << "Person destructor" << std::endl;
  }
private:
  std::string name;
  int age;
};

class Student : public Person {
public:
  Student(std::string n, int a, double g)
    : Person(n, a), gpa(g) {
    std::cout << "Student constructor" << std::endl;
  }
  ~Student() {
    std::cout << "Student destructor" << std::endl;
  }
private:
  double gpa;
};
```

▩ Construction and Destruction Order — Output

```cpp
int main() {
  Student s("Alice", 20, 3.8);
  std::cout << "---" << std::endl;
  return 0;
}
```

Output:

```
Person constructor
Student constructor
---
Student destructor
Person destructor
```

- Construction: base first, then derived (build the foundation, then add the floors)
- Destruction: derived first, then base (tear down the floors, then the foundation)

This always happens automatically — you don't control the order.

Adding and Overriding Member Functions

A derived class can:

1. Add new functions that the base class doesn't have
2. Override base class functions by defining a function with the same name and signature

```
class Person {
public:
  Person(std::string n, int a) : name(n), age(a) {}
  void introduce() const {
    std::cout << "Hi, I'm " << name
              << ", age " << age << std::endl;
  }
protected:
  std::string name;
  int age;
};

class Student : public Person {
public:
  Student(std::string n, int a, double g)
    : Person(n, a), gpa(g) {}
  // Override — same name and signature as Person::introduce()
  void introduce() const {
    std::cout << "Hi, I'm " << name
              << ", age " << age
              << ", GPA " << gpa << std::endl;
  }
private:
  double gpa;
};
```

Overriding in Action

```
int main() {
  Person p("Bob", 35);
  p.introduce();
  // Output: Hi, I'm Bob, age 35

  Student s("Alice", 20, 3.8);
  s.introduce();
  // Output: Hi, I'm Alice, age 20, GPA 3.8

  return 0;
}
```

When you call `introduce()` on a `Student`, the derived version runs — it overrides the base version.

The `Person` version is still there — it just gets hidden by the `Student` version.

Calling the Base Class Version

Sometimes the derived version wants to reuse the base implementation and add to it. Use `BaseClass::function_name()`:

```
class Student : public Person {
public:
  Student(std::string n, int a, double g)
    : Person(n, a), gpa(g) {}

  void introduce() const {
    Person::introduce();  // call the base version first
    std::cout << "My GPA is " << gpa << std::endl;
  }
private:
  double gpa;
};
```

```
Student s("Alice", 20, 3.8);
s.introduce();
// Output:
// Hi, I'm Alice, age 20
// My GPA is 3.8
```

This avoids duplicating the base class logic. If `Person::introduce()` changes, the `Student` version automatically picks up the change.

```cpp
class Shape {
public:
  Shape(std::string n) : name(n) {}
  std::string get_name() const { return name; }
  void describe() const {
    std::cout << "I am a " << name << std::endl;
  }
private:
  std::string name;
};

class Circle : public Shape {
public:
  Circle(double r) : Shape("Circle"), radius(r) {}
  double area() const { return 3.14159 * radius * radius; }
  void describe() const {
    Shape::describe();  // reuse base version
    std::cout << "Radius: " << radius
              << ", Area: " << area() << std::endl;
  }
private:
  double radius;
};
```

Notice how `Circle::describe()` calls `Shape::describe()` to print "I am a Circle", then adds its own details.

```cpp
class Rectangle : public Shape {
public:
  Rectangle(double w, double h)
    : Shape("Rectangle"), width(w), height(h) {}
  double area() const { return width * height; }
  void describe() const {
    Shape::describe();  // reuse: "I am a Rectangle"
    std::cout << "Width: " << width
              << ", Height: " << height
              << ", Area: " << area() << std::endl;
  }
private:
  double width;
  double height;
};

int main() {
  Circle c(5.0);
  c.describe();
  // I am a Circle
  // Radius: 5, Area: 78.5398

  Rectangle r(3.0, 4.0);
  r.describe();
  // I am a Rectangle
  // Width: 3, Height: 4, Area: 12

  return 0;
}
```

▒▒▒ Why `public` Inheritance?

You may have noticed we always write `class Derived : public Base`. There are three kinds of inheritance:

Inheritance type	Base `public` members become	Base `protected` members become
`public`	`public` in derived	`protected` in derived
`protected`	`protected` in derived	`protected` in derived
`private`	`private` in derived	`private` in derived

In practice, you almost always use `public` inheritance. It preserves the "is-a" relationship:

- A `Student` is a `Person` — anywhere you expect a `Person`, a `Student` should work.
- `public` inheritance keeps the base class interface intact.

`protected` and `private` inheritance are rare and used for implementation reuse rather than "is-a" relationships. You won't need them in this course.

Common Pitfalls

1. Forgetting to call the base constructor

If the base class has no default constructor, you must call it explicitly — otherwise you get a compiler error.

2. Accidentally hiding base functions

If a derived class defines a function with the same name but a different signature, it hides (not overrides) the base version. Be careful to match the signature exactly.

3. Object slicing

```
Student s("Alice", 20, 3.8);
Person p = s;  // copies only the Person part — gpa is lost!
```

When you copy a derived object into a base object, the derived parts get "sliced off." We'll see how to handle this properly with polymorphism in a future lecture.

4. Resource management

Each class is responsible for managing its own resources. If `Person` allocates memory, `Person` needs its own destructor/copy/move operations. `Student` only manages what it adds. The Rule of Five applies to each class independently.

▓▓ Summary — Inheritance

- Inheritance lets a derived class reuse the members of a base class
- Syntax: `class Derived : public Base { ... };`
- `protected` members are accessible to derived classes but not outside code
- Derived constructors must call the base constructor in the initializer list
- Construction order: base first, then derived. Destruction: reverse.
- Derived classes can override base functions by defining a function with the same name
- Use `Base::function()` to call the base version from the derived class
- `public` inheritance models the "is-a" relationship
- Watch out for slicing — polymorphism solves this

▒▒▒ The Slicing Problem

Recall that copying a derived object into a base variable loses the derived parts:

```
Student s("Alice", 20, 3.8);
Person p = s;        // sliced — gpa is gone
p.introduce();       // calls Person::introduce(), not Student's
```

But what if we use a pointer or reference to the base class?

```
Person* ptr = &s;
ptr->introduce();   // which introduce() runs?
```

Without any special mechanism, the base version runs — even though `ptr` points to a `Student`. The compiler decides which function to call based on the declared type (`Person*`), not the actual object (`Student`).

This is the problem polymorphism solves.

▓▓ What is Polymorphism?

Polymorphism means "many forms" — the same function call can behave differently depending on the actual type of the object.

In C++, polymorphism is achieved with two ingredients:

1. Mark the base class function as `virtual`
2. Access the object through a pointer or reference to the base class

When both conditions are met, the program decides at runtime which version of the function to call — this is called dynamic dispatch.

Virtual Functions

Add the virtual keyword to the base class function:

```cpp
class Person {
public:
  Person(std::string n, int a) : name(n), age(a) {}

  virtual void introduce() const {
    std::cout << "Hi, I'm " << name
              << ", age " << age << std::endl;
  }

  virtual ~Person() = default;  // always make the destructor virtual!

protected:
  std::string name;
  int age;
};
```

The derived class can use override to make the override explicit:

```cpp
class Student : public Person {
public:
  Student(std::string n, int a, double g)
    : Person(n, a), gpa(g) {}

  void introduce() const override {
    std::cout << "Hi, I'm " << name
              << ", age " << age
              << ", GPA " << gpa << std::endl;
  }
private:
  double gpa;
};
```

Polymorphism in Action

```cpp
int main() {
  Student s("Alice", 20, 3.8);
  Person p("Bob", 35);

  Person* ptr;

  ptr = &p;
  ptr->introduce();
  // Output: Hi, I'm Bob, age 35

  ptr = &s;
  ptr->introduce();
  // Output: Hi, I'm Alice, age 20, GPA 3.8

  return 0;
}
```

Now the actual object type determines which function runs — not the pointer type. This is the power of polymorphism.

Polymorphism with References

References work the same way as pointers:

```cpp
void greet(const Person& p) {
  p.introduce();  // calls the correct version
}

int main() {
  Person p("Bob", 35);
  Student s("Alice", 20, 3.8);

  greet(p);  // Hi, I'm Bob, age 35
  greet(s);  // Hi, I'm Alice, age 20, GPA 3.8

  return 0;
}
```

The function `greet` takes a `const Person&` — but it correctly calls `Student::introduce()` when passed a `Student`. This is why polymorphism is so useful: one function can work with any derived type.

▒▒▒ The `override` Keyword

Always use `override` when overriding a virtual function. It tells the compiler to check that you're actually overriding something:

```
class Student : public Person {
public:
  // Correct — matches the base class signature
  void introduce() const override { ... }

  // COMPILER ERROR — typo in the name, no base function to override
  void intorduce() const override { ... }
};
```

Without `override`, a typo silently creates a new function instead of overriding. With `override`, the compiler catches the mistake.

Rule: always write `override` on derived class virtual functions.

▓▓▓ Virtual Destructors

If you delete a derived object through a base pointer, you must have a virtual destructor:

```
Person* ptr = new Student("Alice", 20, 3.8);
delete ptr;  // without virtual ~Person(), Student's destructor won't run!
```

Rule: if a class has any virtual functions, give it a virtual destructor:

```
class Person {
public:
  virtual ~Person() = default;
  // ...
};
```

This ensures the correct destructor chain runs when deleting through a base pointer.

Pure Virtual Functions and Abstract Classes

Sometimes the base class can't provide a meaningful implementation — it only defines the interface:

```
class Shape {
public:
  Shape(std::string n) : name(n) {}
  virtual ~Shape() = default;

  std::string get_name() const { return name; }

  virtual double area() const = 0;  // pure virtual — no implementation

  virtual void describe() const {
    std::cout << "I am a " << name
              << " with area " << area() << std::endl;
  }

private:
  std::string name;
};
```

The `= 0` makes `area()` a pure virtual function. This makes `Shape` an abstract class — you cannot create a `Shape` object directly:

```
Shape s("thing");  // ERROR — Shape is abstract
```

▓▓ Implementing Pure Virtual Functions

Derived classes must implement all pure virtual functions to be concrete:

```
class Circle : public Shape {
public:
  Circle(double r) : Shape("Circle"), radius(r) {}
  double area() const override {
    return 3.14159 * radius * radius;
  }
private:
  double radius;
};

class Rectangle : public Shape {
public:
  Rectangle(double w, double h)
    : Shape("Rectangle"), width(w), height(h) {}
  double area() const override {
    return width * height;
  }
private:
  double width;
  double height;
};
```

```cpp
int main() {
  Circle c(5.0);
  Rectangle r(3.0, 4.0);

  // Use base class pointers
  Shape* shapes[] = { &c, &r };

  for (Shape* s : shapes) {
    s->describe();
  }
  // Output:
  // I am a Circle with area 78.5398
  // I am a Rectangle with area 12

  return 0;
}
```

Notice how `describe()` calls `area()` — which is pure virtual — and the correct derived version runs. This is polymorphism at work: `Shape` defines the interface, each derived class provides the implementation.

▒▒▒ Inheritance vs. Polymorphism — Summary

Concept	What it does
Inheritance	Derived class reuses base class members
Overriding (without `virtual`)	Derived hides the base function; which one runs depends on the declared type
Polymorphism (`virtual`)	Which function runs depends on the actual object type at runtime
Pure virtual (`= 0`)	Forces derived classes to implement a function; makes the base class abstract

Key rules:

- Use `virtual` on base class functions you want to override polymorphically
- Always use `override` in derived classes
- Always make destructors `virtual` if the class has any virtual functions
- Use pure virtual functions when the base class can't provide a default implementation

▓▓ Lab: Building a Vehicle Hierarchy

Create a file called `vehicle.cpp` with the following:

Base class `Vehicle`:

- Private members: `std::string make`, `int year`
- Constructor that takes `make` and `year`
- Public `get_make()` and `get_year()` methods
- Public `describe()` method that prints: `"[year] [make]"`

Derived class `Car` (inherits from `Vehicle`):

- Private member: `int num_doors`
- Constructor that takes `make`, `year`, and `num_doors`
- Override `describe()` to print: `"[year] [make] — Car with [num_doors] doors"`
  - Call `Vehicle::describe()` to print the first part

Derived class `Truck` (inherits from `Vehicle`):

- Private member: `double payload_capacity`
- Constructor that takes `make`, `year`, and `payload_capacity`
- Override `describe()` to print: `"[year] [make] — Truck with [payload_capacity] ton capacity"`
  - Call `Vehicle::describe()` to print the first part

▒▒▒ Lab: Expected Output

Write a `main()` function that creates and describes each vehicle:

```
int main() {
  Car c("Toyota", 2024, 4);
  c.describe();

  Truck t("Ford", 2023, 2.5);
  t.describe();

  return 0;
}
```

Expected output:

```
2024 Toyota — Car with 4 doors
2023 Ford — Truck with 2.5 ton capacity
```

Hints:

- Remember to call the `Vehicle` constructor from `Car` and `Truck` constructors
- Use `Vehicle::describe()` inside the derived `describe()` methods to avoid repeating code
- Think about whether `describe()` in the base class should print a newline or not (hint: it shouldn't, if the derived classes want to add to the same line)