Friend Functions / Friend Classes

Purpose:
Allow **external functions or classes** to access **private/protected members** of a class.

```
class Box {
private:
   int width;
public:
   Box(int w) : width(w) {}

   friend void showWidth(Box b); // friend function
};

void showWidth(Box b) {
   cout << "Width is: " << b.width << endl; // allowed
}
```

```
class Engine;

class Car {
   friend class Engine; // Engine can access Car's private members
private:
   int speed = 200;
};

class Engine {
public:
   void boost(Car c) {
      cout << "Boosting car to speed: " << c.speed + 100 << endl;
   }
};
```

Object Slicing

What It Is:
When a **derived object is assigned to a base class object**, the extra parts (members of derived) get **sliced off**.

```
class Animal {
public:
   void speak() { cout << "Animal speaks\n"; }
};

class Dog : public Animal {
public:
   void bark() { cout << "Dog barks\n"; }
};

int main() {
   Dog d;
   Animal a = d;  // Object slicing: Dog part is sliced off
   // a.bark(); Not accessible
   a.speak();    // Still works
}
```

Tip: Use pointers or references to prevent slicing.

Binding

Term	Meaning	Example Type
Early Binding	Function call resolved at compile-time	Normal functions, overloads
Late Binding	Function call resolved at runtime	Virtual functions

Non-Virtual Functions

- Regular member functions.
- **Early binding**: resolved at **compile-time**.
- Do **not support polymorphism** when called through base class pointers.

```
class Animal {
public:
   void eat() { cout << "Animal eats\n"; }
};
```

Virtual Functions

- Declared with the virtual keyword in the base class.
- Enables **runtime polymorphism** (late binding).
- Can be overridden in derived classes.

```
class Animal {
public:
   virtual void speak() { cout << "Animal speaks\n"; }
};
```

Pure Virtual Functions

- Declared with = 0.
- Forces derived classes to implement the function.
- Makes the class **abstract**.

```
class Shape {
public:
   virtual void draw() = 0; // Pure virtual
};
```

Polymorphism

The ability for different classes to be treated through a common interface.
Types:

Type	Mechanism	Binding Time
Compile-Time	Function/Operator Overloading	Early Binding
Runtime	Virtual Functions, Base Pointers	Late Binding

Compile-Time (Static):

```
int add(int a, int b);
float add(float a, float b);  // Overloaded
```

Runtime (Dynamic):

```
Animal* a = new Dog();
a->speak();  // Virtual function → late binding
```

Abstraction

Hiding complex implementation details and showing only essential features

.

Achieved by:

- Abstract classes
- Pure virtual functions
- Access specifiers (private, public, protected)

```
class Shape {
public:
   virtual void draw() = 0; // Pure virtual → abstraction
};
```

Abstract Class

- Contains at least one pure virtual function.
- Cannot be instantiated.
- Used to define **interfaces** or **base blueprints**.

Interface in C++

- C++ doesn't have a separate interface keyword.
- An interface is typically an **abstract class with only pure virtual functions**.

```
class IPrintable {
public:
   virtual void print() = 0;
   virtual void display() = 0;
};
```

Concept	Keyword / Feature	Purpose
Abstraction	Abstract class / = 0	Hides details, enforces interface
Polymorphism	Overloading / Virtual	Multiple forms, flexible behavior
Virtual Function	virtual	Enables runtime overriding
Pure Virtual Function	= 0	Forces override in derived classes
Interface	All-pure virtual class	Common API without implementation
Early Binding	Normal/overloaded function	Compile-time function resolution
Late Binding	Virtual function	Runtime decision of function call

```
// Interface using an abstract class (pure virtual functions only)
class IPrintable {
public:
   virtual void print() = 0; // Pure virtual function
};

// Abstract class demonstrating abstraction and interface-like design
class Shape : public IPrintable {
public:
   void area() { // Non-virtual function → Early binding
      cout << "Calculating area (generic shape)\n";
   }

   virtual void draw() { // Virtual function → Can be overridden → Late binding
      cout << "Drawing shape\n";
   }

   virtual void print() = 0; // Pure virtual → forces implementation → abstraction
};
```

```
// Derived class: Circle
class Circle : public Shape {
public:
    // Function overloading → Compile-time polymorphism
    void setRadius(int r) {
        radius = r;
    }

    void setRadius(float r) {
        radius = static_cast<int>(r);
    }

    void draw() override { // Overrides virtual function
        cout << "Drawing Circle\n";
    }

    void print() override {
        cout << "Printing Circle\n";
    }

private:
    int radius;
};
```

```
// Derived class: Square
class Square : public Shape {
public:
    void draw() override {
        cout << "Drawing Square\n";
    }

    void print() override {
        cout << "Printing Square\n";
    }
};
```

```
int main() {
    Circle c;
    Square s;

    // Compile-time polymorphism: function overloading
    c.setRadius(5);
    c.setRadius(5.5f);

    // Early binding: direct call to non-virtual function
    c.area();

    // Runtime polymorphism via base class pointer
    Shape* shapePtr;

    shapePtr = &c;
    shapePtr->draw();  // Late binding → calls Circle's draw
    shapePtr->print(); // Late binding → calls Circle's print

    shapePtr = &s;
    shapePtr->draw();  // Late binding → calls Square's draw
    shapePtr->print(); // Late binding → calls Square's print

    return 0;
}
```

```
Calculating area (generic shape)
Drawing Circle
Printing Circle
Drawing Square
Printing Square
```

Virtual Destructors

Without a **virtual destructor**, deleting a derived class through a base class pointer causes **undefined behavior** (base destructor called only!).

Wrong

```
class Base {
public:
   ~Base() { cout << "Base destroyed\n"; }
};

class Derived : public Base {
public:
   ~Derived() { cout << "Derived destroyed\n"; }
};

Base* b = new Derived();
delete b;  // Only Base's destructor called — memory leak
risk
```

Correct

```
class Base {
public:
   virtual ~Base() { cout << "Base destroyed\n"; }
};

class Derived : public Base {
public:
   ~Derived() { cout << "Derived destroyed\n"; }
};
```

Abstract vs concrete in C++

- If a class has **pure virtual functions**, it's **abstract**.
- If it has **no pure virtual functions**, it's **concrete**.

Aspect	**Abstract in C++**	**Concrete in C++**
Definition	A class with at least **one pure virtual function** (= 0).	A class that has **full implementations** of all its functions.
Instantiation	**Cannot** create an object of an abstract class.	**Can** create an object directly.
Purpose	Defines a **common interface** but leaves the implementation to **derived classes**.	Provides **complete, ready-to-use** behavior.
Syntax	Use virtual return_type function_name() = 0; to declare a pure virtual function.	Regular class with implemented methods — no pure virtual functions.
Example	**Abstract class:** cpp class Animal { public: virtual void makeSound() = 0; // Pure virtual function };	**Concrete class:** cpp class Dog { public: void bark() { std::cout << "Woof!" << std::endl; } };
Key Point	Acts like a **blueprint** for subclasses; forces them to implement missing pieces.	Fully **implemented**; can be used directly without extending it.

Abstract vs concrete in C++

- If a class has **pure virtual functions**, it's **abstract**.
- If it has **no pure virtual functions**, it's **concrete**.

```
// Abstract class
class Animal {
public:
   virtual void makeSound() = 0; // Pure virtual function
};

// Concrete subclass
class Dog : public Animal {
public:
   void makeSound() override {
      cout << "Woof!" << endl;
   }
};

int main() {
   Dog d;    // Concrete class, can be instantiated
   d.makeSound();
   // Animal a; // ERROR! Cannot instantiate abstract class
   return 0;
}
```

Abstract classes in C++ define ***what must be done***, **Concrete classes** in C++ define ***how it's done.***