Inheritance and Polymorphism

C++ for Developers

Content

- Better Constructor
- Inheritance
- Abstract Classes
- Polymorphism

Better Constructors

C++ for Developers

Constructor (Assigning Values)

```cpp
Animal::Animal(std::string n, std::string s, int a)
{
    name = n;
    sound = s;
    age = a;
}
```

1. Default construction of the object.

Animal
name = “”;
sound = “”;
age = uninit;

→

2. Assigning values.

Animal
name = n;
sound = s;
age = a;

lafftale
for developers

What is a default constructor?

A constructor without parameters that initializes an object to its default state.

0 , null , “” and so on.

This is not automatically provided in C++ and need to be defined.

Note: This does not apply to basic data types, they inhibit “garbage” if not assigned when declared.

Examples

Strings
When no argument is passed or value is assigned, a string is by default "" - empty.

Vector
When no argument or value is passed, a vector is by default empty.

Why is this important?

```
class Animal {
private:
    std::string name;
    std::string sound;
    int age;

public:
    Animal(std::string n, std::string s, int a);
```

1. Call Constructor Animal(n,s,a);

```
Animal animal = Animal("Ringo", "Ahssjsjsjs", 231);
```

2. Construct attributes - Default Constructors

```
// Animal
std::string name;
std::string sound;
int age;
```

3. Assign Values

```
Animal::Animal(std::string n, std::string s, int a) {
    name = n;
    sound = s;
    age = a;
}
```

```
class Owner {
private:
    std::string name;
    std::string phone;
    Animal animal;

public:
    Owner(std::string n, std::string p, Animal a);
```

1. Call Constructor Owner(n,s,a);

```
Owner owner = Owner("Birk", "0748312374", animal);
```

2. Construct attributes - Default Constructors

```
// Owner
std::string name;
std::string phone;
Animal animal;
```

```
no default constructor exists for class "Animal" C/C++(291)
Animal animal
```

lafftale for developers

Member-initializer lists

```
Owner::Owner(std::string n, std::string p, Animal a)
: name(n), phone(p), animal(a) {}
```

Syntax:

Class::Constructor(var n)
: fieldName(n) {
    // Constructor body
}

REMEMBER THIS

Calls all constructors with parameters instead!

```
// Owner
std::string name = "Birk";
std::string phone = "0748312374";
Animal animal = Animal("Ringo", "Ahssjsjsjs", 231);
```

**Not optimal code yet - but we will improve it once we learn about memory!*

Needed when:

const
Need to be initialized at declaration, can’t be changed later.

reference
Need to be bound at declaration.

No default constructor
Needed to create objects without parameters

Inherits
When the object is a derived class

From now on... Mem-init!

```
Owner::Owner(std::string n, std::string p, Animal a)
: name(n), phone(p), animal(a) {
    if (name.empty() ||
        phone.empty() ||
        animal.getName().empty() ||
        animal.getSound().empty() ||
        animal.getAge() <= 0) {
            throw std::invalid_argument("Unable to instantiate Owner");
    }
}
```

Inheritance

C++ for Developers

lafftale
for developers

Inheritance

- What is inheritance?
- Syntax
- Access Modifiers

Inheritance

A feature in OOP to enable derived classes to inherit the features of another class.

- Base / Parent class
- Derived / Child class

Child inherits features (fields and methods) from the parent class.

Syntax

1. Create a base class

```
class Animal {
private:
    std::string name;
    std::string sound;
    int age;

public:
    Animal(std::string n, std::string s, int a);

    void makeSound();
    void run();

    void setName(std::string n);
    void setSound(std::string s);
    void setAge(int a);

    const std::string& getName();
    const std::string& getSound();
    int getAge();
};
```

2. Create a derived class

```
class Cat : public Animal {
private:
    double jumpHeight;

public:
    Cat(std::string n, std::string s, int a, double jh);

    void jump();
};
```

3. Implement the classes

```
Animal animal = Animal("Ringo", "Ahssjsjsjs", 231);
animal.run();
animal.makeSound();

Cat cat = Cat("Snuffles", "Mjaow", 8, 2.31);
cat.run();
cat.makeSound();
cat.jump();
```

Access modifiers

- **private** - only available inside the class.
- **protected** - only available inside the base and derived classes.
- **public** - available both inside and outside the inheritance hierarchy

Access modifiers

animal.h

```cpp
class Animal {
private:
    std::string name;
    std::string sound;
    int age;

public:
    Animal(std::string n, std::string s, int a);

    void makeSound();
    void run();

    void setName(std::string n);
    void setSound(std::string s);
    void setAge(int a);

    const std::string& getName();
    const std::string& getSound();
    int getAge();
};
```

cat.h

```cpp
class Cat : public Animal {
private:
    double jumpHeight;

public:
    Cat(std::string n, std::string s, int a, double jh);

    void jump();
};
```

```cpp
void Cat::jump() {
    std::cout
        << name << " jumped " << jumpHeight << "cm into the air, "
        << "which is impressive since it's" << age << " years old!\n";
}
```

```cpp
void Cat::jump() {
    std::cout
        << getName() << " jumped " << jumpHeight << "cm into the air, "
        << "which is impressive since it's" << getAge() << " years old!\n";
}
```

Access modifiers

Why do we use private inside classes?

- Encapsulation
- Data Validation
- Reusing code
- Safe inheritance

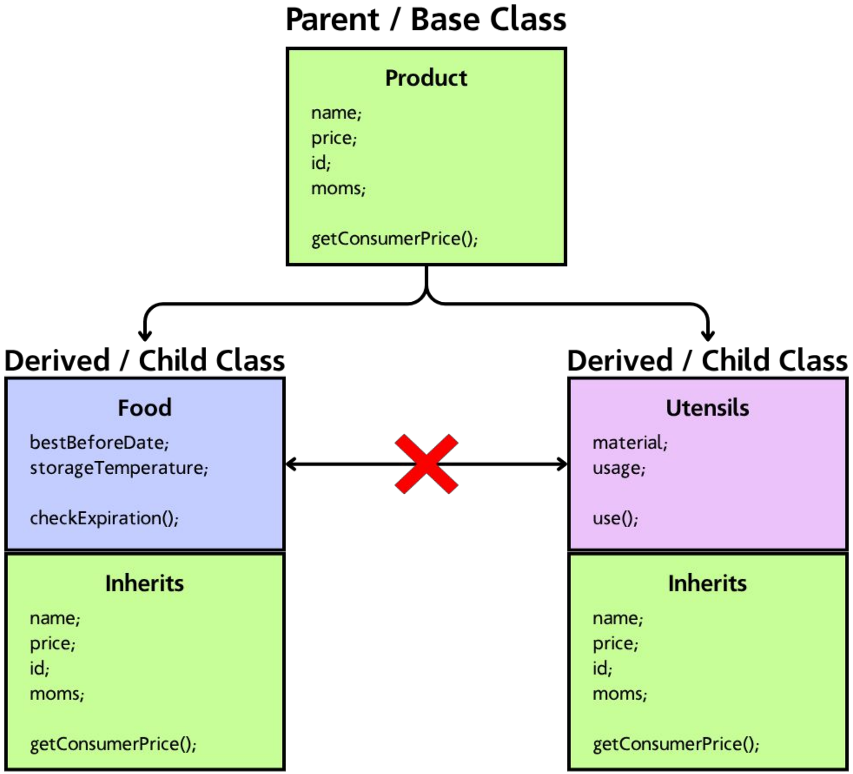
Let's look at some code!

https://github.com/lafftale1999/cpp_for_developers/tree/main/week_3/1_oop_inheritance/1_2_store

Follow along while I code an inheritance hierarchy!

lafftale for developers

Hierarchy



Parent / Base Class
Product
name;
price;
id;
moms;
getConsumerPrice();
Derived / Child Class
Food
bestBeforeDate;
storageTemperature;
checkExpiration();
Inherits
name;
price;
id;
moms;
getConsumerPrice();
Derived / Child Class
Utensils
material;
usage;
use();
Inherits
name;
price;
id;
moms;
getConsumerPrice();

Abstract Classes

C++ for Developers

Abstract Classes

By declaring abstract classes and inheriting them, we can enforce methods declared in the class. They can not be instantiated.

This is similar to interfaces in Java and C#.

Differences in C++

What separates it from interface?
Abstract classes can still define member functions, variables, constructors and destructors.

Defining abstract classes

```
class Shape {
public:
    virtual ~Shape() = default;
    virtual void printArea() const = 0;
    virtual void printCircumference() const = 0;
};
```

New keywords

- virtual
- virtual destructors
- virtual functions
- trailing const
- override

Virtual keyword

Tells the compiler that this function may be replaced in a derived class, and when the program call it through a base pointer/reference, we want the derived version to run.

When we use the virtual keyword like this:

```
virtual void printCircumference() const = 0;
```

It's called a *pure virtual function*. Meaning it has no base implementation and <u>need</u> to overridden in the derived classes.

Override

When you declare a virtual function in a base class, a derived class can provide its own implementation. Marking it with override tells the compiler to check that it really replaces a base virtual function.

Then, when you call the function through a base reference or pointer, the derived version runs (dynamic dispatch).

Virtual Destructor

Informs the compiler that when the destructor is called through base reference or a pointer - it should use the derived classes destructor.

```
virtual ~Shape() = default;
```

We will dig deeper on this subject further ahead!

Trailing const

These act as a promise to the compiler that this function do not, (can't even), modify the object holding the member function.

This is mandatory when passing objects as consts in arguments.

```
void Circle::printArea() const {
    double area = pow((diameter / 2), 2) * M_PI;
    std::cout << "Circle area: " << diameter << std::endl;
}
```

```
void printCircleInformation(const Circle& circle) {
    circle.printArea();
    circle.printCircumference();
}
```

These member functions must be consts!

```
void printArea() const;
void printCircumference() const;

void printArea();
void printCircumference();
```

They can be overloaded!

lafftale
for developers

To summarize

These keywords inform the compiler that:

- **virtual** - This function may be changed in the derived classes.
- **override** - This function is meant to replace a virtual from a base class.
- **trailing const** - The function does not modify member attributes.

Use-cases of interfaces

Separates the *what* from the *how.*

Creates an easy to understand interface where the programmers can know *what* to use, without needing to know *how*.

- Driver standardization
- Sensor abstraction
- Portability between hardware

Polymorphism

C++ for Developers

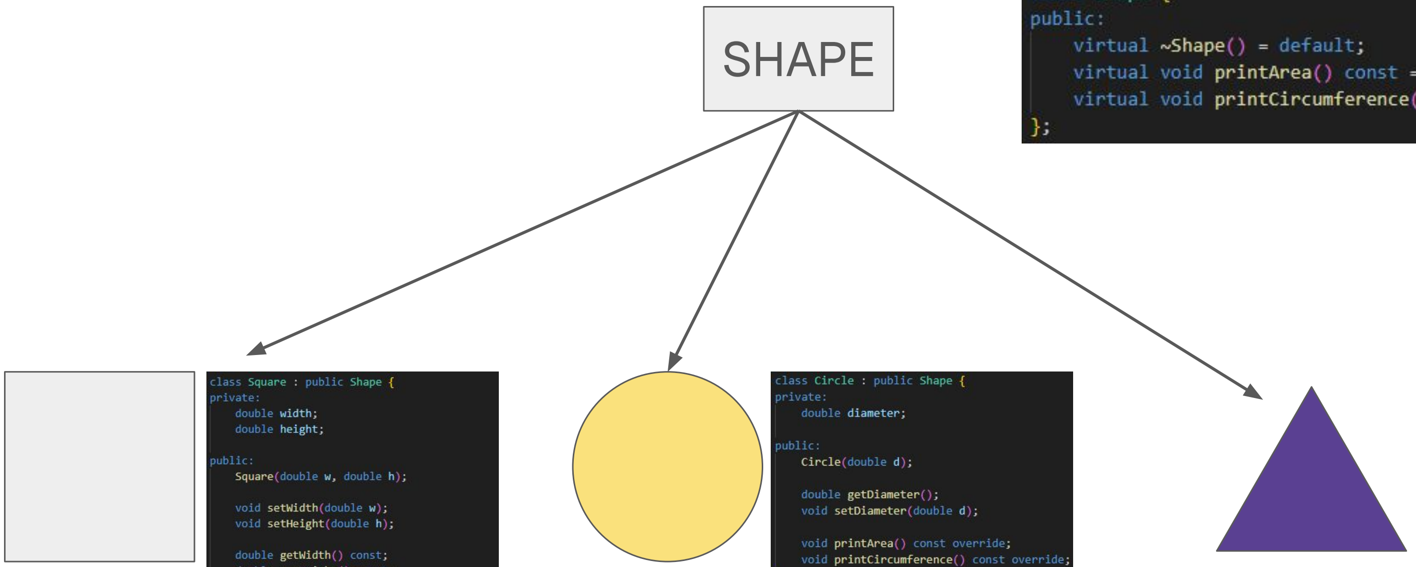
Recap

- Use member-initializer lists in constructor
- Inheritance enables derived classes to inherit features from the base class
- Encapsulate each class to ensure data validation
- Virtual informs the compiler that the use can be overridden from a derived class.

Polymorphism

Mechanism in OOP that enables derived classes to be used as instances of shared base classes.

What does it mean?




```cpp
class Shape {
public:
    virtual ~Shape() = default;
    virtual void printArea() const = 0;
    virtual void printCircumference() const = 0;
};
```

```cpp
class Square : public Shape {
private:
    double width;
    double height;

public:
    Square(double w, double h);

    void setWidth(double w);
    void setHeight(double h);

    double getWidth() const;
    double getHeight() const;

    void printArea() const override;
    void printCircumference() const override;
};
```

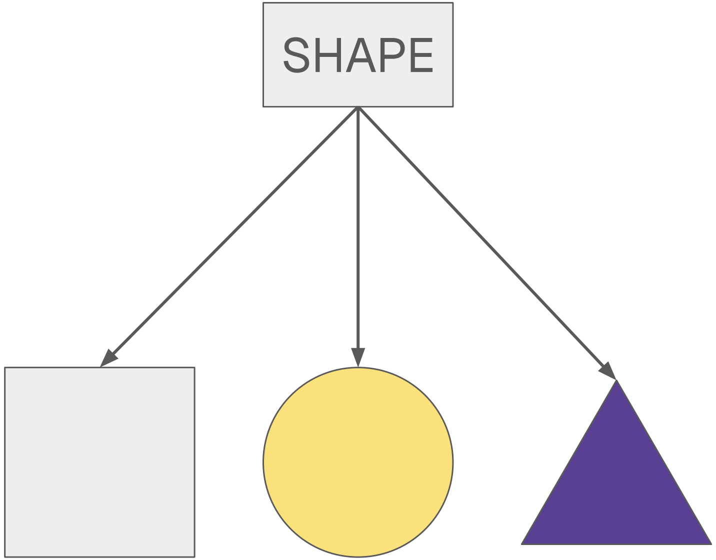
```cpp
class Circle : public Shape {
private:
    double diameter;

public:
    Circle(double d);

    double getDiameter();
    void setDiameter(double d);

    void printArea() const override;
    void printCircumference() const override;
};
```

```cpp
class Triangle : public Shape {
private:
    double height;
    double aLen;
    double base;
    double cLen;

public:
    Triangle(double h, double b);
    Triangle(double h, double a, double b, double c);

    double getHeight() const;
    double getBase() const;
    void setHeight(double h);
    void setWidth(double w);

    void printArea() const override;
    void printCircumference() const override;
};
```

Implementing

```cpp
int main(void) {
    Circle circle = Circle(10);
    Triangle triangle = Triangle(10,10);
    Square square = Square(10, 10);

    std::vector<Shape*> shapes = {&circle, &triangle, &square};

    for (const auto& s : shapes) {
        s->printArea();
        s->printCircumference();
    }
}
```

```
Circle area: 78.5398
Circle circumference: 31.4159
Triangle area: 50
Triangle circumference: Undefineable - need more information
Square area: 100
Square circumference: 40
```



What is accessible through polymorphism?

```
Circle circle = Circle(10);
Triangle triangle = Triangle(10,10);
Square square = Square(10, 10);

std::vector<Shape*> shapes = {&circle, &triangle, &square};

for (const auto& s : shapes) {
    s->printArea();
    s->printCircumference();
    s->
}
          printArea            virtual void Shape::printArea() const
          printCircumference
          ~Shape
```

We can’t reach the member functions declared in the derived classes - only the ones publicly declared in Shape.

How does it work?

Virtual tables

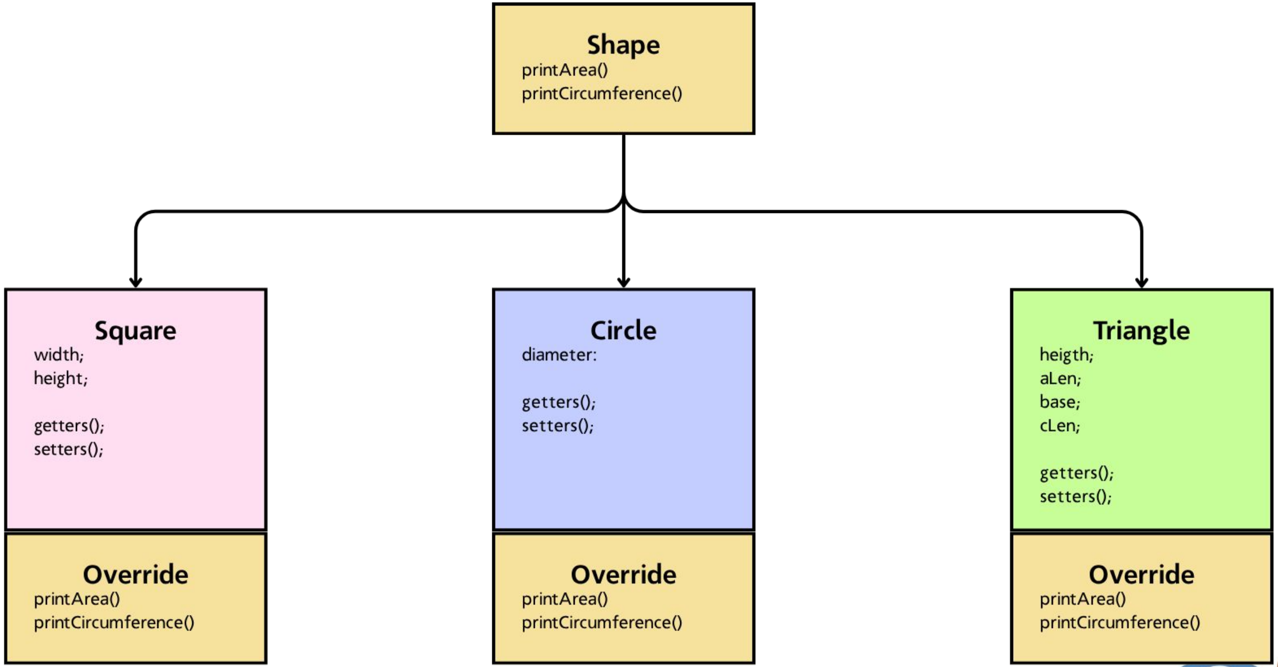
A table containing pointers to the implementation of the virtual function.

Virtual pointers

Pointers based on *‘this’*, pointing to the derived objects virtual table instead of the base class.

Only created if a class have a virtual member function

Hierarchy


Shape
printArea()
printCircumference()
Square
width;
height;
getters();
setters();
Override
printArea()
printCircumference()
Circle
diameter:
getters();
setters();
Override
printArea()
printCircumference()
Triangle
heigth;
aLen;
base;
cLen;
getters();
setters();
Override
printArea()
printCircumference()

Instantiation

```cpp
class Shape {
public:
    virtual ~Shape() = default;
    virtual void printArea() const = 0;
    virtual void printCircumference() const = 0;
};
```

1. Instantiate an object of Circle

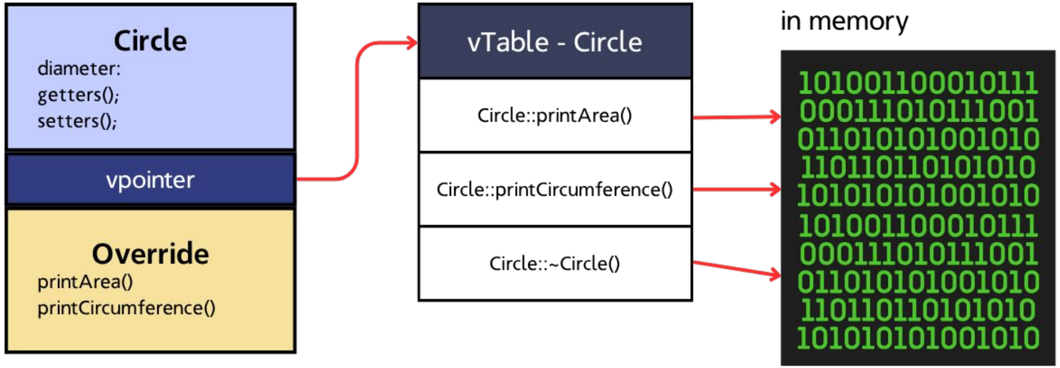
```cpp
class Circle : public Shape {
private:
    double diameter;

public:
    Circle(double d);

    double getDiameter() const;
    void setDiameter(double d);

    void printArea() const override;
    void printCircumference() const override;
};
```

```cpp
Shape* s = new Circle(10);
```

2. Adds a vpointer to the object

3. Points to its vtable

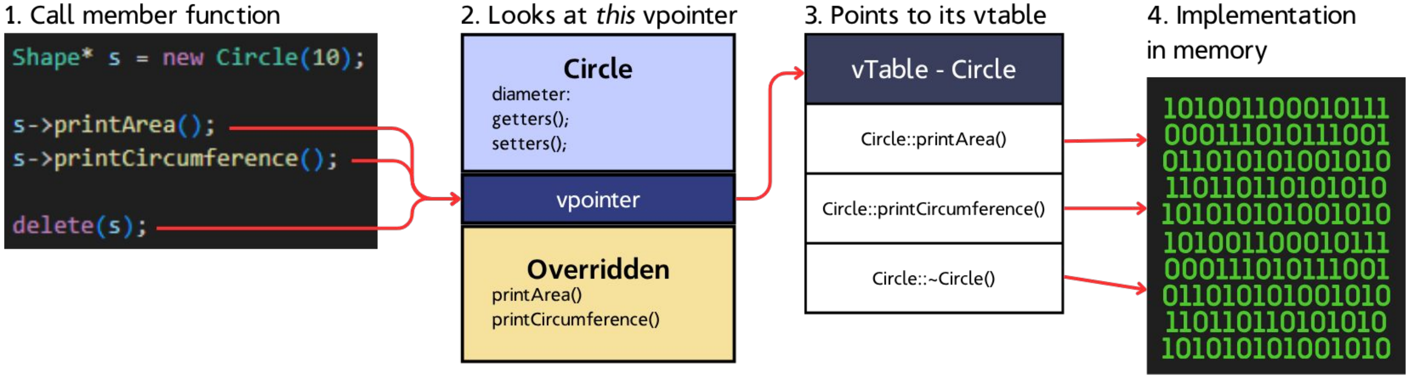
4. Implementation in memory



** This is an abstraction*

Calling through base class

```cpp
class Shape {
public:
    virtual ~Shape() = default;
    virtual void printArea() const = 0;
    virtual void printCircumference() const = 0;
};
```



* *This is an abstraction*

Why is this good?

- Uniform use of different objects that share inheritance
- Reusing code where it makes sense
- Ensures good practice when inheriting classes

Real world example: Payment

```cpp
class PaymentProcessor {
public:
    virtual ~PaymentProcessor() {}
    virtual bool process(double amount) = 0;
};

class CreditCardProcessor : public PaymentProcessor {
public:
    bool process(double amount) override {
        // talk to credit card gateway
        return true;
    }
};

class PayPalProcessor : public PaymentProcessor {
public:
    bool process(double amount) override {
        // use PayPal API
        return true;
    }
};

class CryptoProcessor : public PaymentProcessor {
public:
    bool process(double amount) override {
        // crypto transfer
        return true;
    }
};
```

```cpp
void checkout(PaymentProcessor& processor, double amount) {
    if (processor.process(amount)) {
        std::cout << "Payment successful\n";
    } else {
        std::cout << "Payment failed\n";
    }
}
```

```cpp
int main(void) {

    PaymentProcessor* p = new CreditCardProcessor();

    p->process(2500);

    delete(p);

    return 0;
}
```

lafftale for developers

Real world example: GUI

```cpp
class Widget {
public:
    virtual ~Widget() {}
    virtual void draw() = 0;
    virtual void handleEvent(int eventId) = 0;
};

class Button : public Widget {
public:
    void draw() override {
        // draw button rectangle, text, etc.
    }
    void handleEvent(int eventId) override {
        // check for clicks
    }
};

class TextBox : public Widget {
public:
    void draw() override {
        // draw text box and caret
    }
    void handleEvent(int eventId) override {
        // handle key input
    }
};
```

```cpp
int main(void) {
    std::vector<Widget*> widgets = { new Button(), new TextBox() };

    for (auto* w : widgets) {
        w->draw();
    }

    return 0;
}
```

Best practices

- Often better to write several, smaller base classes.
- Focus on cohesiveness - methods and fields are directly related to the class.
- Inherit several smaller base classes, rather than one big one.

lafftale
for developers