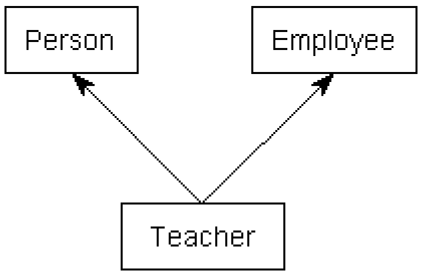
24.9 — Multiple inheritance

ALEX[1] JULY 11, 2024

So far, all of the examples of inheritance we've presented have been single inheritance -- that is, each inherited class has one and only one parent. However, C++ provides the ability to do multiple inheritance. **Multiple inheritance** enables a derived class to inherit members from more than one parent.

Let's say we wanted to write a program to keep track of a bunch of teachers. A teacher is a person. However, a teacher is also an employee (they are their own employer if working for themselves). Multiple inheritance can be used to create a Teacher class that inherits properties from both Person and Employee. To use multiple inheritance, simply specify each base class (just like in single inheritance), separated by a comma.



```cpp
#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 };

    return 0;
}
```

Mixins

A **mixin** (also spelled “mix-in”) is a small class that can be inherited from in order to add properties to a class. The name mixin indicates that the class is intended to be mixed into other classes, not instantiated on its own.

In the following example, the `Box`, `Label`, and `Tooltip` classes are mixins that we inherit from in order to create a new `Button` class.

```cpp
// h/t to reader Waldo for this example
#include <string>

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

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

class Label // mixin Label class
{
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{};
};

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

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

int main()
{
	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");
}
```

You may be wondering why we use explicit `Box::`, `Label::`, and `Tooltip::` scope resolution prefixes when this isn't necessary in most cases.

1. `Label::setText()` and `Tooltip::setText()` have the same prototype. If we called `button.setText()`, the compiler would produce an ambiguous function call compilation error. In such cases, we must use the prefix to disambiguate which version we want.
2. In non-ambiguous cases, using the mixin name provides documentation as to which mixin the function call applies to, which helps make our code easier to understand.
3. Non-ambiguous cases may become ambiguous in the future if we add additional mixins. Using explicit prefixes helps prevent this from occurring.

For advanced readers

Because mixins are designed to add functionality to the derived class, not to provide an interface, mixins typically do not use virtual functions (covered in the next chapter). Instead, if a mixin class needs to be customized to work in a particular way, templates are typically used. For this reason, mixin classes are often templatized.

Perhaps surprisingly, a derived class can inherit from a mixin base class using the derived class as a template type parameter. Such inheritance is called **Curiously Recurring Template Pattern** (CRTP for short), which looks like this:

```cpp
// The Curiously Recurring Template Pattern (CRTP)

template <class T>
class Mixin
{
    // Mixin<T> can use template type parameter T to access members of Derived
    // via (static_cast<T*>(this))
};

class Derived : public Mixin<Derived>
{
};
```

You can find a simple example using CRTP here (https://en.cppreference.com/w/cpp/language/crtp)[2].

Problems with multiple inheritance

While multiple inheritance seems like a simple extension of single inheritance, multiple inheritance introduces a lot of issues that can markedly increase the complexity of programs and make them a maintenance nightmare. Let's take a look at some of these situations.

First, ambiguity can result when multiple base classes contain a function with the same name. For example:

```cpp
#include <iostream>

class USBDevice
{
private:
    long m_id {};

public:
    USBDevice(long id)
        : m_id { id }
    {
    }

    long getID() const { return m_id; }
};

class NetworkDevice
{
private:
    long m_id {};

public:
    NetworkDevice(long id)
        : m_id { id }
    {
    }

    long getID() const { return m_id; }
};

class WirelessAdapter: public USBDevice, public NetworkDevice
{
public:
    WirelessAdapter(long usbId, long networkId)
        : USBDevice { usbId }, NetworkDevice { networkId }
    {
    }
};

int main()
{
    WirelessAdapter c54G { 5442, 181742 };
    std::cout << c54G.getID(); // Which getID() do we call?

    return 0;
}
```

When `c54G.getID()` is compiled, the compiler looks to see if WirelessAdapter contains a function named getID(). It doesn't. The compiler then looks to see if any of the parent classes have a function named getID(). See the problem here? The problem is that c54G actually contains TWO getID() functions: one inherited from USBDevice, and one inherited from NetworkDevice. Consequently, this function call is ambiguous, and you will receive a compiler error if you try to compile it.

However, there is a way to work around this problem: you can explicitly specify which version you meant to call:

```cpp
int main()
{
    WirelessAdapter c54G { 5442, 181742 };
    std::cout << c54G.USBDevice::getID();

    return 0;
}
```

While this workaround is pretty simple, you can see how things can get complex when your class inherits from four or six base classes, which inherit from other classes themselves. The potential for naming conflicts increases exponentially as you inherit more classes, and each of these naming conflicts needs to be resolved explicitly.

Second, and more serious is the diamond problem (https://en.wikipedia.org/wiki/Diamond_problem)[3], which your author likes to call the "diamond of doom". This occurs when a class multiply inherits from two classes which each inherit from a single base class. This leads to a diamond shaped inheritance pattern.

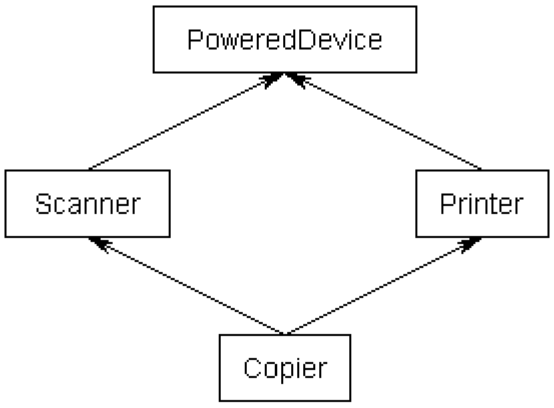
For example, consider the following set of classes:

```cpp
class PoweredDevice
{
};

class Scanner: public PoweredDevice
{
};

class Printer: public PoweredDevice
{
};

class Copier: public Scanner, public Printer
{
};
```



Scanners and printers are both powered devices, so they derived from PoweredDevice. However, a copy machine incorporates the functionality of both Scanners and Printers.

There are many issues that arise in this context, including whether Copier should have one or two copies of PoweredDevice, and how to resolve certain types of ambiguous references. While most of these issues can be addressed through explicit scoping, the maintenance overhead added to your classes in order to deal with the added complexity can cause development time to skyrocket. We'll talk more about ways to resolve the diamond problem in the next chapter (lesson 25.8 -- Virtual base classes (https://www.learncpp.com/cpp-tutorial/virtual-base-classes/)[4]).

Is multiple inheritance more trouble than it's worth?

As it turns out, most of the problems that can be solved using multiple inheritance can be solved using single inheritance as well. Many object-oriented languages (eg. Smalltalk, PHP) do not even support multiple inheritance. Many relatively modern languages such as Java and C# restrict classes to single inheritance of normal classes, but allow multiple inheritance of interface classes (which we will talk about later). The driving idea behind disallowing multiple inheritance in these languages is that it simply makes the language too complex, and ultimately causes more problems than it fixes.

Many authors and experienced programmers believe multiple inheritance in C++ should be avoided at all costs due to the many potential problems it brings. Your author does not agree with this approach, because there are times and situations when multiple inheritance is the best way to proceed. However, multiple inheritance should be used extremely judiciously.

As an interesting aside, you have already been using classes written using multiple inheritance without knowing it: the iostream library objects std::cin and std::cout are both implemented using multiple inheritance!

Best practice

Avoid multiple inheritance unless alternatives lead to more complexity.

Next lesson
24.x Chapter 24 summary and quiz

5

Back to table of contents

6

Previous lesson
24.8 Hiding inherited functionality

7

8

inheritedindividual
August 2, 2024 11:49 am PDT

what about in a case where a shark entity would have babies with a bird entity so the result inherits from them both to become a flying shark? would it be ok in a situation like this? (with the flying shark not being able to be inherited from afterwards)

5 Reply

Faizan
June 25, 2024 8:42 am PDT

I feel as if composition is better than mixins, I can't think of a scenario where you'd absolutely have to use a mixin although I'm sure they exist.

With the example given Button has an is-a relationship with box but you could also say that it has a has-a relationship too. When it comes to Label and tool-tip they both just flat out are has-a relationships.

I dunno, I understand mixins but I'm just struggling to see where I'd ever actually use them.

0 Reply

Viktor
Reply to Faizan 11 December 5, 2024 3:09 am PST

Like Rafal says, and adding information from this chapter, buttom uses 3 mixins. But combobox could uses the same or part mixins. another buttom type (buttom2, buttom4...) can use all of them. recycle code etc

0 Reply

rafal
Reply to Faizan 11 July 3, 2024 12:19 pm PDT

Live example from my life:
When you don't want to use same interface.
I made breakout game as uni project. I (was made to) written my own openGL engine for that.
It had class representing cube. I initially had it as composition, but changed it to to inheritance because:

- I did not want to rewrite methods for setting position of the brick it's size etc. Those were used by class that manages game map. And would boil down to wrappers.
- Did not want to expose implementation details (member variable Cube), also that'd mean I would have to write more code to first access this member and then invoke methods or write wrappers.

I actually first used composition then decided it's better as inheritance as it hides implementation details. I had getter not wrappers. So I changed it to inheritance and wrote other code. Then I tried to go back, but that'd be breaking change for other existing code that utilized inherited methods. And I'd end up with useless wrappers if I wanted to have it as composition.

3 Reply

D D
December 9, 2023 5:10 am PST

Hello.

1. Some fields are not brace initialized
2. Why do you write this way (`.Box`, `.Label`), it's redundant?

```
button.Box::setTopLeft({ 1, 1 });
    button.Box::setBottomRight({ 10, 10 });
    button.Label::setText("Username: ");
    button.Label::setFontSize(6);
```

0 Reply

Alex Author
Reply to D D[12] December 11, 2023 10:06 am PST

1. Fixed
2. Two reasons: First, it adds documentary value (especially in cases where the function name might be ambiguous otherwise). Second, it ensures we don't get an ambiguous function call compile error in cases where two mixins have a function with the same prototype.

While neither of these are the case in this particular example, doing so guards against these possibilities if we later add another mixin.

Updated the example to better illustrate and address this point.

1 Reply

Foobar
September 8, 2023 4:05 am PDT

Hi Alex,
here's a little advice: if you have time, you can express the inheritance with standard UML diagrams. Although the illustration in these lessons themselves are clear enough :P

Last edited 1 year ago by Foobar

3 Reply

Issei
August 7, 2023 3:01 am PDT

I like this SO answer (https://stackoverflow.com/a/407928)[13] on multiple inheritance.

"This choice is inherent to the problem, and in C++, unlike other languages, you can actually do it without dogma forcing your design at language level."

"Third, object orientation is great, but it is not The Only Truth Out ThereTM in C++. Use the right tools, and always remember you have other paradigms in C++ offering different kinds of solutions."

Really highlights to me the freedom C++ affords to the programmer, which can be both a good thing and a bad thing.

1 Reply

Yuuki
January 8, 2023 2:36 am PST

In the first example, could we not have returned a `std::string_view` instead as `m_name` is not a local variable? I believe it was talked about somewhere(or is it going to cause UB if m_name is empty?). Code being referred to:

```
const std::string& getName() const { return m_name; }
```

Thanks!

0 Reply

Alex Author
Reply to Yuuki 14 January 11, 2023 1:31 pm PST

`m_name` has type `std::string`, so it's better to return a `const std::string&` to avoid unnecessary conversions.

1 Reply

Waldo Lemmer
September 21, 2022 3:03 am PDT

C++23 lets you implement the CRTP like so (cppreference):

```
struct Base { void name(this auto& self) { self.impl(); } };
struct D1 : public Base { void impl() { std::puts("D1::impl()"); } };
struct D2 : public Base { void impl() { std::puts("D2::impl()"); } };
```

It's a little hard to wrap my head around it, but it looks like Base::name()'s *this* parameter's type is deduced from the derived object on which it is called, so it calls impl() for that derived object. This means that, instead of Base being a class template, its member function(s) are now function templates

If Base::name() were a normal member function, the code wouldn't compile since Base has no impl() member function. If Base::impl() were created, the call inside Base::name() would resolve to Base::impl(), not D1::impl(), even on an object of type D1

0 Reply

Alex Author

Reply to Waldo Lemmer 15 September 22, 2022 12:46 pm PDT

Yep, this uses the new C++23 explicit object parameter feature, described here: https://en.cppreference.com/w/cpp/language/member_functions#Explicit_object_parameter

From that article: "explicit object parameter deduces to the derived type", so for a D1, Base::name's explicit `this` would have type D1*.

1 Reply

Strain

Reply to Alex 16 April 27, 2024 10:55 am PDT

Wouldn't it be `DI&`, instead? The explicit `this` is defined as `auto&`, and the reference would stay. Or am I somehow tripping?

0 Reply

Alex Author

Reply to Strain 17 April 27, 2024 1:20 pm PDT

Eer yes, I believe so.

0 Reply

Edyk

August 3, 2022 8:26 am PDT

Hi Alex,

in the first example, in the Teacher-constructor, why don't you pass employer as reference?

Last edited 2 years ago by Edyk

0 Reply

Alex Author

Reply to Edyk 18 August 6, 2022 10:35 pm PDT

Typo. I updated the example to use `std::string_view` instead.

0 Reply

Grey Eminence

August 1, 2022 7:12 am PDT

Hi Alex,
how does this cast work? (CRTP)
`static_cast<T*>(this)`

Would it be the other way around (static_cast<this>(T*)), it is even hard to understand because we haven't defined a user-defined type conversion between these types, have we? However, in this case we convert from the derived class to the base class it derives from, so it "knows how the base class looks like".

How does is work in this case (and in the other one (static_cast<this>(T*)))?

Last edited 2 years ago by Grey Eminence

0 Reply

Alex Author

Reply to Grey Eminence [19] August 6, 2022 8:47 pm PDT

We're just casting the `this` pointer from the base type to the Derived type (which is T*) so we can access Derived members. You can't put `this` inside the angled brackets, as `this` is a pointer, not a type.

1 Reply

Waldo

July 16, 2022 8:02 am PDT

I think this lesson needs a few practical examples of classes that use multiple inheritance (e.g. Mixins (https://en.wikipedia.org/wiki/Mixin)[20])

Edit:

```cpp
#include <string>

struct Point2D {
    int x;
    int y;
};

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

class Label {
public:
    void setTopLeft(Point2D point) { m_topLeft = point; }
    void setFontSize(int fontSize) { m_fontSize = fontSize; }
private:
    Point2D m_topLeft{};
    int m_fontSize{};
};

class Button: public Box, public Label {};

int main() {
    Button button{};
    button.Box::setTopLeft({ 1, 1 });
    button.Box::setBottomRight({ 10, 10 });
    button.Label::setTopLeft({ 2, 2 });
    button.Label::setFontSize(6);
}
```

Last edited 2 years ago by Waldo

6 Reply

Alex Author

Reply to Waldo [21] July 19, 2022 11:22 am PDT

Good idea! Added a section about mixins (borrowing your example, slightly modified) and added a note about CRTP for advanced readers. Thanks!

3 Reply

Waldo

Reply to Alex [22] July 19, 2022 2:07 pm PDT

Thanks! Just one thing:

> mixins typically do not use virtual functions or destructors

Virtual functions are introduced in the next chapter

4 Reply

Alex Author

Reply to Waldo [23] July 20, 2022 11:41 am PDT

Moved that paragraph into the advanced box. Thanks!

3 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://en.cppreference.com/w/cpp/language/crtp
3. https://en.wikipedia.org/wiki/Diamond_problem
4. https://www.learncpp.com/cpp-tutorial/virtual-base-classes/
5. https://www.learncpp.com/cpp-tutorial/chapter-24-summary-and-quiz/
6. https://www.learncpp.com/
7. https://www.learncpp.com/cpp-tutorial/hiding-inherited-functionality/
8. https://www.learncpp.com/multiple-inheritance/
9. https://www.learncpp.com/cpp-tutorial/adding-new-functionality-to-a-derived-class/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-598793
12. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-590705
13. https://stackoverflow.com/a/407928
14. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-575980
15. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-573411
16. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-573462
17. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-596311
18. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-571512
19. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-571399
20. https://en.wikipedia.org/wiki/Mixin
21. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-570685
22. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-570779
23. https://www.learncpp.com/cpp-tutorial/multiple-inheritance/#comment-570814
24. https://g.ezoic.net/privacy/learncpp.com