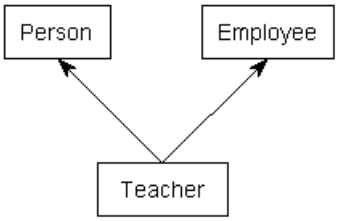
17.9 — Multiple inheritance

ALEX AUGUST 2, 2021

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>

class Person
{
private:
    std::string m_name;
    int m_age {};

public:
    Person(std::string name, int age)
        : m_name { name }, m_age { age }
    {
    }

    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 employer, double wage)
        : m_employer { employer }, m_wage { wage }
    {
    }

    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 name, int age, std::string employer, double wage, int
teachesGrade)
        : Person { name, age }, Employee { employer, wage }, m_teachesGrade { teachesGrade
}
    {
    }
};
```

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:

```
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, 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.

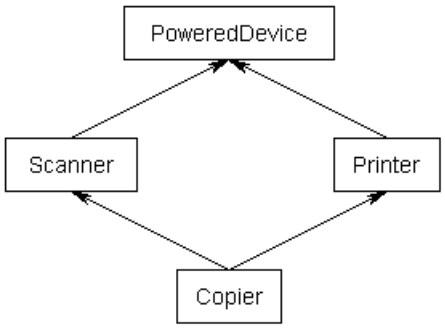
For example, consider the following set of classes:

```
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 18.8 -- Virtual base classes).

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
17.x Chapter 17 comprehensive quiz

Back to table of contents

Previous lesson
17.8 Hiding inherited functionality

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest