Virtual Functions, Introduction to Polymorphism

Programming Fundamentals

Dr Kevin Chalmers

School of Computing
Edinburgh Napier University
Edinburgh

k.chalmers@napier.ac.uk

Edinburgh Napier UNIVERSITY

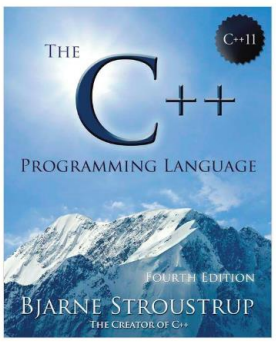
Notes

Outline

Notes

References

- Stroustrup (2011), *The C++ Programming Language, 4th Edition*
  - The inventor of C++
- Also look at the official C++ documentation coming with your development environment



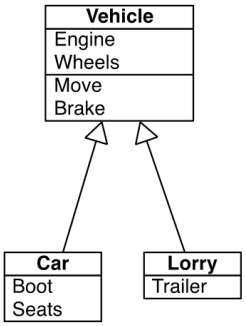
Notes

Outline

Notes

Reminder Inheritance

- In the last unit, we covered the basics of object-orientation.
- One of these notions was inheritance.
- Inheritance is a code reuse mechanism allowing the creation of *derived (child)* classes from a *base (parent)* class.



Notes

Overriding Behaviours

- In some situations we need to have different implementations of the same methods in different inherited classes.
  - The base class provides a *interface* (or contract) for the type of behaviour
  - The derived class implements its own version of that behaviour
- In the previous example, even if a `Car` and a `Lorry` are vehicles, the braking system can be different.
- The `brake()` method needs to be overridden so each child class offers its own implementation.

Notes

Casting

- A first method to override behaviours is to use the casting operator on a child object to force a call on a parent, e.g. in our example we have `Car` and `Vehicle`:

Casting from `Car` to `Vehicle`

```
Car *c = new Car();
Vehicle *v = (Vehicle *)c;
c->brake();
```

- If doing so, we are calling the implementation of `brake` defined in the `Vehicle` class. This can cause some problems.

Notes

Issues with Casting

- By casting, we treat a `Car` object as a `Vehicle` object.
- This is useful if we are working with a generic collection of `Vehicle` (e.g. `vector<Vehicle*>`) that can contain `Cars` or `Lorrys`.
- However, we still want our objects to behave as intended: `Cars` as cars, `Lorrys` as lorries.
- Is there a different method we can use to ensure the right method is called?

Notes

Virtual Functions

- C++ introduces the `virtual` keyword to flag any function that can be overridden when a call is made to a child object.
- For our vehicle example (constructor and other methods omitted):

Declaring a `virtual` Function

```
class Vehicle
{
public:
    virtual void brake()
    {
        // Some code
    }
};
```

Notes

Virtual Functions

- By adding the `virtual` keyword, we tell the compiler to create a pointer to a *virtual method table* (aka. vtable).
- A virtual method table is a table containing pointers to the right version of the methods that should be called.
- This allows us to call the correct function of a child object even if it is treated as a parent object through casting. This allows a more generic form of programming to happen.

Notes

Example - Vehicles

- We have a `Vehicle` class, which implements two methods: `move()` and `brake()`

Extended `Vehicle` Example

```
class Vehicle
{
    // Constructor omitted
public:
    virtual void move() { /* Some code */ }
    virtual void brake() { /* Some code */ }
};
```

Notes

Example - Vehicles

- There are two subclasses of `Vehicle` - `Car` and `Lorry` - each bringing their own implementation of those methods

Subclassing `Vehicle`

```
class Car : public Vehicle
{
    // Constructors omitted
public:
    void move() { /* car code */ }
    void brake() { /* car code */ }
};

class Lorry : public Vehicle
{
    // Same as car but with Lorry specific code
};
```

Notes

Example - Vehicles

- We can store all `Car` and `Lorry` objects in a collection of type `Vehicle` and call their `brake()` method. The vtable will do its job to call the right method, e.g.:

Calling `virtual` Behaviour

```
vector<Vehicle*> v;
// Add some Cars and Lorries and add them to v
// Stop all vehicles
for (int i = 0; i < v.size(); i++)
    v[i]->brake();
// Will call the correct brake() method of Car or
   Lorry
```

Notes

Dynamic Dispatch

- The virtual method table is a C++ tool supporting polymorphism.
- In our example, all objects in the `vector` are technically pointers to vehicles (`Vehicle*`). The compiler cannot determine any further than this.
- By defining methods as virtual, the compiler creates a link to the vtable.
- The correct implementation of `brake()` is called at execution time.

Notes

Polymorphism

- Polymorphism is a crucial notion of Object-Oriented Programming (OOP) that you will investigate further in later programming modules.
- It is a very powerful approach to programming that will allow you to treat multiple objects of different types in bulk by treating them at a more general level.
  - This is called *generalisation*.
  - Deriving a general class is called *specialization*
- Just like `malloc`/`free` and `new`/`delete` with memory management, virtual functions allow you to manage method calls beyond compile time.

Notes

Questions?

Notes

Outline

Notes

Virtual Destructors

- If we work with dynamic memory allocation, objects need a destructor.
- Destructors follow the same rules as other methods and need to be declared `virtual` if we use polymorphism, e.g.

Declaring a `virtual` Destructor

```
class Vehicle
{
    // Other methods skipped
    virtual ~Vehicle()
    {
        // Code
    }
};
```

Notes

Virtual Destructors

- Destructors are called in the following order: child, then parent.
- If we do not declare the parent destructor as `virtual`, we risk a situation where the child destructor is never called. In our previous example:

Invoking the `virtual` Destructor

```
Vehicle *l = new Lorry();
delete l;
```

- If the destructor for `Vehicle` is not declared `virtual`, it will be the only one called, causing a possible memory leak.

Notes

Questions?

Notes

Outline

Notes

Abstract Classes

- In our previous examples we have two classes - `Car` and `Lorry`, inherited from a class called `Vehicle`.
- We never actually create any `Vehicle` object. Just like in real life, it is an abstract concept whose instances are in the form of cars, buses, vans etc.
- We can therefore explicitly declare `Vehicle` as an abstract class, i.e. a class that cannot generate objects unless another subclass inherits from it.

Notes

Pure Virtual Functions

- A lot of modern languages (C#, Java) have an explicit way to declare abstract classes. C++ is slightly different.
- To declare an abstract class in C++, we need to declare one or more of its methods as *pure virtual*.
- To do this, we need to declare the method as equal to 0, e.g.

Declaring a Pure Virtual Method

```
virtual void brake() = 0;
```

Notes

Example - Vehicles

- Let's take our previous example and make `Vehicle` an abstract class this time:

Declaring `Vehicle` as Abstract

```
class Vehicle
{
    // Rest omitted
public:
    virtual void move() = 0;
    virtual void brake() = 0;
};
```

- If we try to instantiate a `Vehicle`, the compiler would now send an error message.

Notes

Example - Vehicles

- As a consequence, all the pure virtual methods *must* to be implemented in a child class, or the child class will also be abstract.

Declaring `Car` as Abstract

```
class Car : public Vehicle
{
public:
    void move() { /* Car code */ }
    // brake() deliberately omitted
};
```

- We can no longer create instances of type `Car`

Notes

Questions?

Notes

Outline

Notes

override

- C++11 introduced the `override` keyword, already present in other languages, to clearly mark a function as overriding a `virtual` one, e.g.:

Using the `override` Keyword

```
class Car : public Vehicle
{
public:
    void move() override { /* Car code */ }
    void brake() override { /* Car code */ }
};
```

Notes

override

- The use of `override` is not mandatory but strongly encouraged if your project is in C++11 or newer.
- If you work in legacy C++, you will not be able to use it.
- The presence of `override` ensures a compile error happens if conditions are not met
  - Missing base method
  - Mismatching argument types
  - etc.
- Without the keyword, the compiler may or may not throw a warning, which can make debugging harder in complex projects.

Notes

final

- In some situations, you do not want a method to be overridden.
- C++11 supports the `final` keyword to ban any attempt at implementing an `override` function.
- This is useful if you work on APIs for example, to limit what third-party developers can do.

Declaring a Function as `final`

```
class Car : public Vehicle
{
public:
    void brake() override final { /* Car code */ }
};
```

Notes

Questions?

Notes

Outline

Notes

Summary

- This lecture introduced some notions of polymorphism to our exploration of C++:
  - *Dynamic dispatching*, enabled by the use of `virtual` functions, and allowing the to work with collections of sub-classes in bulk.
  - *Abstract classes*, which cannot be instantiated unless sub-classes are created. They are enabled by the use of pure virtual functions.
  - Modern code-safety features of C++11, such as the `override` and `final` keywords.

Notes

Summary

- Polymorphism is a crucial notion in object-oriented development. Today we have only scratched the surface with sub-classing and dynamic dispatching.
- Many languages offer many methods, beyond the scope of this module. For example, another useful feature of C++ is Templates that bring the notion of *Generic Datatypes*.
  - `vector<type>` is an example of using a template.

Notes

To do...

- In the lab - lots of work with polymorphism. Just to keep you thinking!
  - Feel free to continue using Visual Studio if you want. However, coursework 2 submission will be code and make files again.
- Coursework 2 is now live on Moodle. Any queries contact Kevin ASAP.
- Next time - operator overloading (just what you need for the coursework).

Notes

Notes