Polymorphism in C++

Tom Latham

THE UNIVERSITY OF WARWICK

From Static to Dynamic Polymorphism

- "Polymorphism"
  - The ability of one **generic** *type* to behave in many **concrete** ways.
- We have already seen this behaviour with the container classes
  - Enabled through the use of templates
  - The polymorphism is locked at compile time
  - Hence this is referred to as **static polymorphism**
- Today we'll see how to achieve this at run time: **dynamic polymorphism**
  - This is achieved through **type-sharing**
  - In C++, the mechanism for this is **public inheritance**.

Templated functions

- Consider the two `swap` functions on the right - we can see that the only difference between them is the type (of the arguments and of the `tmp` variable)

- We are duplicating code – and you can see that this would proliferate if we wanted to have other functions to swap two `float`'s or two `bool`'s

- This duplication can be eliminated using templates

```
void swap( double& a, double& b )
{
        double tmp {b};
        b = a;
        a = tmp;
}


void swap( int& a, int& b )
{
        int tmp {b};
        b = a;
        a = tmp;
}
```

Static polymorphism

- This function template definition specifies a family of functions that swap the values of two variables
- We have a single bit of code that can act in different ways depending on the type of T – it behaves polymorphically
- However, this difference is locked-in at compile time with the specification of the template parameter, either explicitly or by compiler deduction
  - In this particular case, the compiler can deduce the template parameter from the type of the arguments
  - In some scenarios that isn't possible, and so it has to be specified explicitly:

    ```
    swap<double>(x,y);
    ```

  - This can also be done even when the compiler can deduce the type, e.g. to provide clarity

```
template <typename T>
void swap( T& a, T& b )
{
        T tmp {b};
        b = a;
        a = tmp;
}


int main()
{
        double x {42.3};
        double y {11.2};

        swap(x,y);

        int i {4};
        int j {-6};

        swap(i,j);
}
```

Types

- On the first day we defined four important concepts, including “type”:
  - A **type** defines the set of possible *value*s and a set of operations for an *object*.
  - An **object** is some memory that holds a *value* of some *type*.
  - A **value** is a set of bits interpreted according to a *type*.
  - A **variable** is a named *object*.

- We've also seen that that there are:
  - Built-in types: `int`, `double`, `char`, …
  - User-defined types: `CaesarCipher`, `PlayfairCipher`, …
  - Families of types: `vector<int>`, `vector<double>`, `vector<char>`, …

Sharing Type

- Software design often throws up cases where a set of types all exhibit an "**Is a Kind Of**" relationship to some, more abstract, concept.
    - e.g. `Car`, `Bus`, `Lorry` are all "kinds of" vehicle

- In this example, they all have engines that can be started and stopped, they can be travelling at a given speed, in a certain gear, etc. etc.

- What this means in programming terms is that they all provide the same interface, i.e. the same set of public member functions, e.g.
    - `int changeGear();`
    - `bool startEngine();`

Sharing Type in a Strongly Typed Language

- We might want to store instances of, say, `Car`, `Motorbike`, `Scooter` in a container

```
vector<??> caughtSpeeding;
```

- We might design a class that has a data member that may change between, say, `Car`, `Motorbike`, `Scooter` as time changes:

```
class Employee {
 private:
  ?? personalVehicle;
};
```

- Unfortunately, neither is possible directly as C++ is a “strongly typed” language, i.e. we have to supply, at compile-time, the type “`??`”.
- We need a mechanism to express the type sharing relationship in the code.

Sharing types in mpags-cipher

- By now you have implemented several different ciphers as classes:
  - `CaesarCipher`
  - `VigenereCipher`
  - `PlayfairCipher`

- These are all kinds of cipher and should have the same interface, in particular the `applyCipher` function

- We want to be able to express this relationship in our code to avoid duplication of the code where we used the different ciphers

Public Inheritance

- Type sharing in C++ is achieved through **public inheritance**.

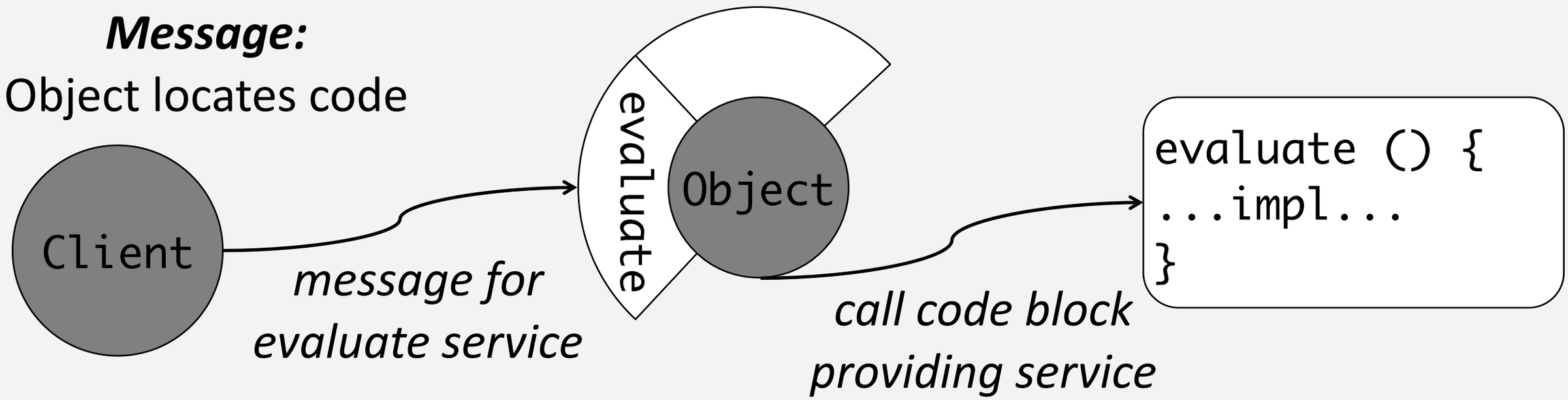
- You create a new `class` that contains only the declarations of the member functions, i.e. the **interface**, that defines the common, abstract type.
    - This class generally contains no data members and has no implementation code!
    - So you generally only need a header (.hpp) file for this base class

- Classes that want to share this type **publicly inherit** from this **base** class.

- What they inherit is the obligation to provide the public member functions specified by the base class.

Dynamic binding

- Binding time is the moment when a function is assigned to service a message
  - Recall that it is the object that determines which code should be called



- To obtain dynamic polymorphism, we need binding to be done at run-time, not at compile time (the default behaviour)
  - i.e. we want “late” (or “dynamic”) binding, rather than “early” (or “static”) binding

Writing Purely Abstract Base Classes

A “pABC” specifies the interface a type must provide and implement, but no data and no implementation for this interface, as shown below.

We see the new C++ keyword `virtual` prepended to each method signature. This tells the compiler to defer the decision on which actual code to call until runtime (**dynamic binding**).

Each method has `=0` appended to inform the compiler that no actual implementation is provided for the method - it is “pure virtual”.

Vehicle.hpp (~/Documents/Teaching/2015-16/Day5-Slides) - VIM

```
1  #ifndef VEHICLE_HPP
2  #define VEHICLE_HPP
3
4  class Vehicle {
5    public:
6      Vehicle() = default;
7      Vehicle(const Vehicle& rhs) = default;
8      Vehicle(Vehicle&& rhs) = default;
9      Vehicle& operator=(const Vehicle& rhs) = default;
10     Vehicle& operator=(Vehicle&& rhs) = default;
11     virtual ~Vehicle() = default;
12
13     virtual int changeDownGear() = 0;
14     virtual int changeUpGear() = 0;
15
16     virtual bool startEngine() = 0;
17     virtual bool stopEngine() = 0;
18
19     virtual double currentSpeed() const = 0;
20 };
21
22 #endif // VEHICLE_HPP
~
:q                                                    11,33         All
```

Notes

It's important to note that the destructor must be declared and must be virtual. This is so that classes inheriting the pABC are destructed correctly.

This has the side effect that all the other "special functions" must also be declared.

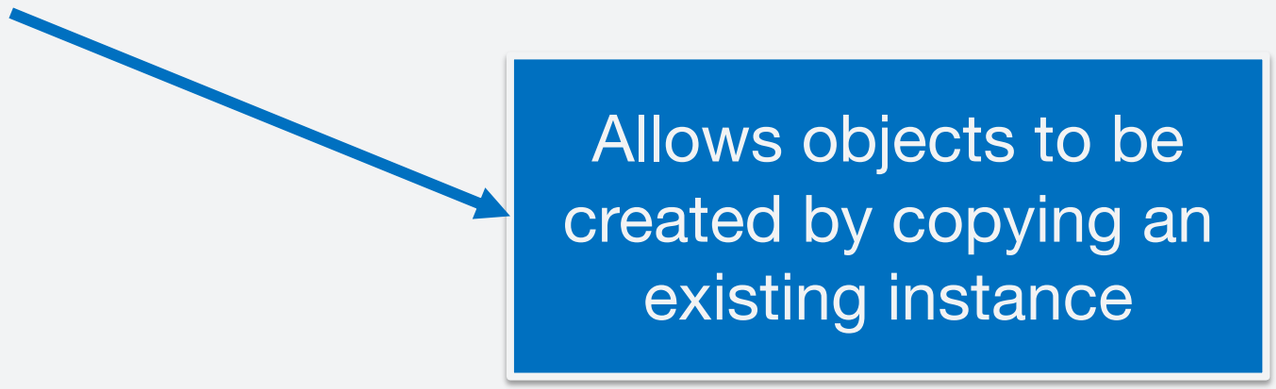
The "= default" syntax indicates that the compiler-supplied versions should be used (we don't have to write the code ourselves) – see next slide for details.

Aside: Compiler-provided functions

- So what are these "special functions" that we saw on the previous slide? And why have we not considered them before?
- They all handle resource management and are called:
  - Copy constructor
  - Move constructor
  - Copy assignment operator
  - Move assignment operator
  - Destructor
- If you do not specify them, the compiler creates them for you
- In the majority of cases the compiler-provided versions will work perfectly well, which is why we have not had to worry about them so far!

Aside: Compiler-provided functions

- So what are these "special functions" that we saw on the previous slide? And why have we not considered them before?
- They all handle resource management and are called:
  - Copy constructor
  - Move constructor
  - Copy assignment operator
  - Move assignment operator
  - Destructor



- If you do not specify them, the compiler creates them for you
- In the majority of cases the compiler-provided versions will work perfectly well, which is why we have not had to worry about them so far!

Aside: Compiler-provided functions

- So what are these "special functions" that we saw on the previous slide? And why have we not considered them before?
- They all handle resource management and are called:
  - Copy constructor
  - Move constructor
  - Copy assignment operator
  - Move assignment operator
  - Destructor

- If you do not specify them, the compiler creates them for you
- In the majority of cases the compiler-provided versions will work perfectly well, which is why we have not had to worry about them so far!

Aside: Compiler-provided functions

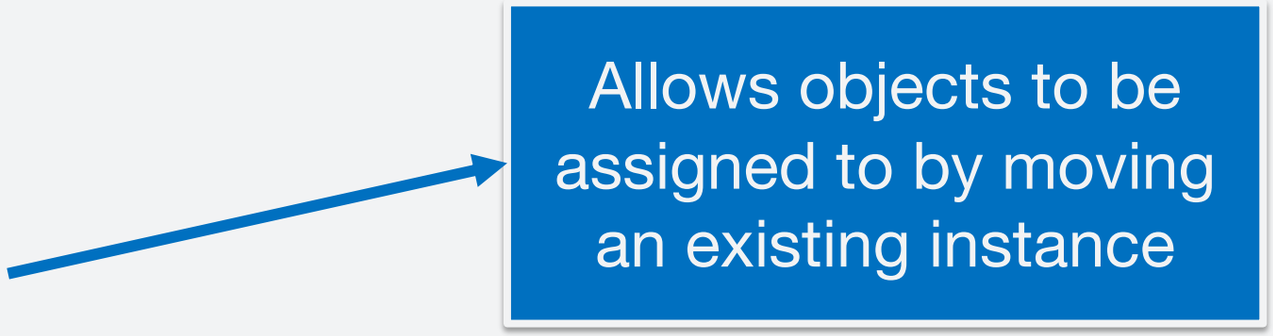
- So what are these "special functions" that we saw on the previous slide? And why have we not considered them before?
- They all handle resource management and are called:
  - Copy constructor
  - Move constructor
  - Copy assignment operator
  - Move assignment operator
  - Destructor

Allows objects to be assigned to by copying an existing instance

- If you do not specify them, the compiler creates them for you
- In the majority of cases the compiler-provided versions will work perfectly well, which is why we have not had to worry about them so far!

Aside: Compiler-provided functions

- So what are these "special functions" that we saw on the previous slide? And why have we not considered them before?
- They all handle resource management and are called:
  - Copy constructor
  - Move constructor
  - Copy assignment operator
  - Move assignment operator
  - Destructor



- If you do not specify them, the compiler creates them for you
- In the majority of cases the compiler-provided versions will work perfectly well, which is why we have not had to worry about them so far!

Aside: Compiler-provided functions

- So what are these "special functions" that we saw on the previous slide? And why have we not considered them before?
- They all handle resource management and are called:
  - Copy constructor
  - Move constructor
  - Copy assignment operator
  - Move assignment operator
  - Destructor

- If you do not specify them, the compiler creates them for you
- In the majority of cases the compiler-provided versions will work perfectly well, which is why we have not had to worry about them so far!

Aside: Compiler-provided functions

- However, in some cases there is the need to write custom versions
- Or, as we've done here, the compiler-provided destructor is fine but we need to specify that it should be virtual
- The problem is that if you specify one of these functions, the compiler assumes that you are taking over (to some extent or other) and will not provide some or all of these five functions
- So we have to say explicitly that we want the compiler to continue to provide these for us, which is what the "= default" does
- For more details see:

http://en.cppreference.com/w/cpp/language/rule_of_three

And links therein.

- We would generally advise to follow the "rule of all or nothing", i.e. to specify none of them when possible (the "rule of zero") but if you have to specify one you should do so for all five (the "rule of five")

Exercise: A pABC for a cipher interface

- Write a pABC called `Cipher` (put the declaration in MPAGSCipher/Cipher.hpp)
- It should have one pure virtual member function:
  - The function to encrypt and decrypt the supplied text:

```
virtual std::string applyCipher( const std::string& input,
const CipherMode mode ) const = 0;
```

- It will also need the lines shown on previous slide to obtain the `virtual` destructor (and keep all the other compiler-defined special functions)
- Make sure to add the appropriate documentation comments

Recap on Public Inheritance

- It is the mechanism by which we inform the compiler that our **concrete classes** - those from which we create object instances - also hold the type of the abstract base class

- The terms **derived class** and **base class** are often used to describe this inheritance relationship

- Public inheritance means that all the public methods in the base class remain public in the derived class (hence they share an interface)
    - C++ also allows private and protected inheritance, which result in sharing of implementation but not shared type
    - Composition/Aggregation are better design patterns in these cases

Writing Derived Classes

To derive a class, say `Car`, from our `Vehicle` base class, we append `": public Vehicle"` (as shown below) to the class declaration, to make `Car` publicly inherit from `Vehicle`.

The `Car` interface must include all the pure virtual function signatures from `Vehicle`, plus any functions specific to `Car` (like the constructor) and data members relevant to `Car`. The implementation of the virtual functions for `Car` follows just as for any other class.

Car.hpp (~/Documents/Teaching/2015-16/Day5-Slides) - VIM

```
1  #ifndef CAR_HPP
2  #define CAR_HPP
3
4  #include "Vehicle.hpp"
5
6  class Car : public Vehicle {
7    public:
8      Car( const int numberOfGears );
9
10     virtual int changeDownGear() override;
11     virtual int changeUpGear() override;
12
13     virtual bool startEngine() override;
14     virtual bool stopEngine() override;
15
16     virtual double currentSpeed() const override;
17
18   private:
19     bool engineOn_ = false;
20     const int numberOfGears_ = 6;
21     int currentGear_ = 0;
22     double currentSpeed_ = 0.0;
23 };
24
25 #endif // CAR_HPP
~
"Car.hpp" 25L, 504C written                    16,5          All
```

Notes

Like any other type, we have to ensure the declaration of the base class is present before we can inherit from it.

As the derived class will implement the virtual functions, these are still declared virtual in `Car`, but we replace the `=0` marker with `override`.

Exercise: Deriving from Cipher

- Now you can modify your concrete classes (`CaesarCipher`, `PlayfairCipher`, `VigenereCipher`) so that they inherit from your new base class (`Cipher`)

- Depending on your exact implementation of the concrete classes, it should be as straightforward as adding `": public Cipher"` to the class declaration and adding `override` to the end of the `applyCipher` function declaration

Dynamic Polymorphism

- We've now got implementations of three concrete types, `CaesarCipher`, `PlayfairCipher`, `VigenereCipher`, which also have the shared type `Cipher`

- Now we can take advantage of the combination of
  - **Type sharing** (via *public inheritance*)
  - **Dynamic binding** (via *virtual member functions*)

- So we can now address a set of objects that are instances of different derived classes through, e.g. a **reference** to an object of the base class type

Using Dynamic Polymorphism

We can now write functions that have a reference to an object of the base class type as an argument. As we can see below, via dynamic polymorphism, we can supply this function with instances of the different derived classes because they share type with the base class.

Each time we call the function, we can get different behaviour depending on the concrete type of the instance provided.

main.cpp (~/Documents/Teaching/2015-16/Day5-Slides) - VIM

```
1 #include <iostream>
2 #include "Car.hpp"
3 #include "Scooter.hpp"
4 #include "Lorry.hpp"
5
6 void printSpeed( const Vehicle& vehicle )
7 {
8   std::cout << vehicle.currentSpeed() << std::endl;
9 }
10
11 int main()
12 {
13   Car car( 5 );          // 5 gears
14   Lorry lorry( 8, 3 );  // 8 gears, 3 axles
15   Scooter scooter( 4 ); // 4 gears
16
17   printSpeed( car );
18   printSpeed( lorry );
19   printSpeed( scooter );
20
21   return 0;
22 }
~
~
~
~
"main.cpp" 22L, 387C written                    6,1           All
```

Notes

We can only achieve dynamic polymorphism via references or (smart) pointers (see later slides).

Exercise: using dynamic polymorphism in a test

- We can now write a function that can take any of our ciphers as a reference argument

- We can use this to simplify writing tests for our ciphers

- Within a new file (Testing/testCiphers.cpp), implement a function that takes a cipher, the encrypt/decrypt mode, the input text and the expected output text and which returns a boolean to indicate whether or not the actual output matches the expectation:

```
bool testCipher( const Cipher& cipher, const CipherMode mode,
const std::string& inputText, const std::string& outputText)
```

- Then write a test that uses this function to test all three of your ciphers