

Part VIII

More classes

Outline

29 Inheritance

30 Virtual functions

31 Polymorphism

32 Classes and static

Inheritance

- In OOP, we may have objects which are of some general type, but are also of some specialised type
- For example, we may have a Vehicle class with various member functions:

```
Vehicle v;  
v.setNumberPassengers(4);  
v.startEngine();
```

- However, there are types of vehicle with features not shared by all other vehicles:

```
car.openBoot();
```

doesn't apply to a ferry, for example

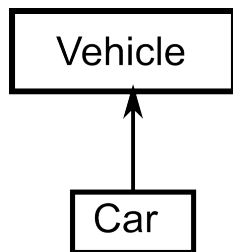
Inheritance

- We would like to have a `Car` be a specialized type of `Vehicle`:

```
class Car : public Vehicle
{
public:
    void openBoot();
};
```

- Now `Car` inherits the member functions and data of `Vehicle`, and can be regarded as being of type `Vehicle`:

```
Car car;
car.openBoot();
car.startEngine();
Vehicle v;
v.openBoot(); // Compile-time error
```



Protected inheritance for data

When `Car` derives from `Vehicle`, we may want it to have access to some of `Vehicle`'s data:

```
class Vehicle{
public:
    void setNumPassengers(int);
    int getNumPassengers() const;
protected:
    int numPassengers;
private:
    int vehicleData;
};
```

Now, a member-function of `Car` can access `numPassengers`, but not `vehicleData`.

Protected inheritance for functions

- The same applies to member-functions of a parent class:

```
class Vehicle{  
protected:  
    void runStarterMotor();  
};
```

- This is then not accessible from outside `Vehicle` (only called through a public function such as `turnIgnitionOn`).
- However, it is accessible from member functions of `Car` because `Car` inherits `Vehicle` as a public class

Inheritance rules

Suppose we have a classes X and Y:

```
class X{
    public:
        int pub;
        int m();
    protected:
        int pro;
    private:
        int pri;
};
```

```
class Y : public X {
    public:
        int f();
};
```

```
int main() {
    X x;
    x.pub=0; // OK
    x.pro=0; // Error
    x.pri=0; // Error
}
```

```
int X::m() {
    pub = 0; // OK
    pro = 0; // OK
    pri = 0; // OK
}
```

```
int Y::f() {
    pub = 0; // OK
    pro = 0; // OK
    pri = 0; // Error
}
```

Constructing base classes

- In order to initialize members of a base class in a constructor, you can either initialize them directly:

```
Child::Child(int a, int b) {  
    m_parentData = a;  
    m_childData = b;  
}
```

or by calling the base class constructor:

```
Child::Child(int a, int b) : Parent(a) {  
    m_childData = b;  
}
```


Outline

29 Inheritance

30 Virtual functions

31 Polymorphism

32 Classes and static

Virtual functions

- What happens if we want a **Car** to have a maximum number of passengers, but most **Vehicles** do not have this restriction?
- We could introduce a new data-member, `maximumPassengers`, to the **Vehicle** class:

```
void Vehicle::setNumberPassengers(int p) {  
    if( p > maximumPassengers) {  
        std::cout << "Crowded" << std::endl;  
    }  
}
```

- However, this represents a leakage of information into the base class
- Anyway, we would potentially have to do this every time we wanted to add extra information to a **Car**

Virtual functions

- Virtual functions allow definitions of functions in a base-class to be re-implemented by a derived class

```
class Vehicle{
public:
    virtual void setNumberPassengers (int);
};

void Vehicle::setNumberPassengers (int p) {
    passengers = p;
}

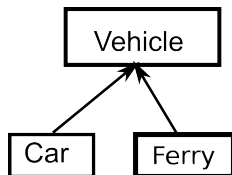
class Car : public Vehicle{
    void setNumberPassengers (int) override;
};

void Car::setNumberPassengers (int p) {
    if ( p > 6 ) {
        std::cout << "Crowded" << std::endl;
    }
    passengers = p;
}
```

Virtual functions ctd.

- If `setNumberPassengers()` is called on a `Vehicle`, then the first version is called.
- If `setNumberPassengers()` is called in a `Car`, then the second version is called.

```
Ferry f;  
f.setNumberPassengers(10); //  
    Will not print error message  
Car c;  
c.setNumberPassengers(10); //  
    Will print error message
```



Abstract classes

- Consider the function `openBonnet`:
- There may be no sensible way of defining the function `openBonnet` for a generic `Vehicle`, but all `Vehicles` must implement this function
- This can be expressed as follows:

```
class Vehicle{  
public:  
    virtual void openBonnet() = 0;  
};
```

Abstract classes ctd

- `openBonnet` is now a pure virtual function
- `Vehicle` is an abstract class
- No object of type `Vehicle` can now exist
- All classes derived from `Vehicle` must implement `openBonnet`
- Trying to create an object of type `Vehicle` will fail.

```
class Car : public Vehicle{
public:
    void openBonnet() override;
};

void Car::openBonnet() {
    // Code here
}

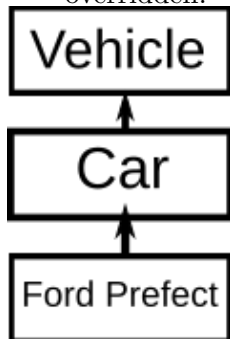
Vehicle v; // Compile-time failure
Car c; // OK
Vehicle* c2 = new Car; // OK
c2->openBonnet(); // Calls Car's version.
```

Abstract classes ctd

- It is an error to specify **override** for a function that is not overriding another one.
- The main reason for this syntax is clarity for the developer about the intent of the class/function.

Final functions

- Sometimes we want to prevent virtual functions from being overridden.



```
class Car : public Vehicle{
public:
    virtual void turnIgnition(bool) const final;
};
class FordPrefect : public Car{
public:
    void turnIgnition(bool) const override; // Error
```


Outline

- 29 Inheritance
- 30 Virtual functions
- 31 Polymorphism**
- 32 Classes and static

Polymorphism

- Polymorphism allows objects of one type to be referred to as objects of another type, if their inheritance allows:

```
Vehicle* v = new Car;  
v->setNumberPassengers(4);
```

- will call `Vehicle`'s version of `setNumberPassengers`
- or `Car`'s, if `Car` overrides `setNumberPassengers` as a virtual function

```
std::array<Vehicle*, 4> v;  
v[0] = new Car;  
v[1] = new Ferry;  
v[0]->setNumberPassengers(4); // Calls Car's version  
v[1]->setNumberPassengers(10); // Calls Ferry's version
```

Polymorphism ctd.

- The function that will be called is only determined at run-time.
- This does not contradict static type-checking
- Assigning an object of type `Car` to a `Vehicle` pointer causes the compiler to check that `Car` derives from `Vehicle`.
- For a class with virtual functions, the compiler will generate a “v-table”, which allows it, for a given derived class, to determine which function it should jump to.
- This could be thought of as a class containing a set of function pointers for each of its virtual functions
- Calling a virtual function therefore requires a very tiny amount of extra overhead compared to a simple member function call.

Losing polymorphism (Slicing)

- Object slicing occurs when a derived object is assigned to a variable of base-class type:

```
DerivedClass d;  
BaseClass b = d;
```

will compile.

- However, the assignment only copies data relevant to `BaseClass` from `d` to `b` (the rest has been sliced off).
- The same problem would occur in the following:

```
DerivedClass *d = new DerivedClass;  
BaseClass b = *d;
```

- Here, `b` is only of type `BaseClass`
- This behaviour is nearly always undesirable.

Downcasting

- So far, we have seen examples of up-casting, where a derived class pointer is converted to a base-class pointer
- Consider the following:

```
void maintain(Vehicle* v) {  
    v->checkChain();  
}
```

and we want to call this function for a collection of **Vehicles**

- However, only a **Bicycle** has a chain, so this will fail to compile!
- (Here, **maintain** should probably be a member function of **Vehicle**, but ignore this for the purposes of argument.)
- How can we check at run-time whether **v** is a **Bicycle**?

Dynamic-casting

- The following is valid:

```
Bicycle* b = dynamic_cast<Bicycle*>(v);
```

and will convert `v` to be of type `Bicycle` if possible.

- If not possible, then the `dynamic_cast` returns `nullptr`:

```
if( b ){  
    b->checkChain();  
}
```

is perfectly valid.

- Almost all uses of `dynamic_cast` are the result of bad design and are better replaced by a member function:

```
bool Vehicle::isBicycle() const;
```

(or just making `maintain` a virtual member function of `Vehicle`)

- However, there are cases where it is needed

Multiple inheritance

- It is also possible for a class to derive from multiple classes

```
class EvilWizard : public EvilCreature,  
                  public MagicUser{}
```

since not all `EvilCreatures` can wield magic,
and not all `MagicUsers` are evil.

- An `EvilWizard` can therefore be used as an `EvilCreature` or as a `MagicUser`
- We can now use

```
EvilCreature* e = new EvilWizard;  
if( dynamic_cast<MagicUser*>(e) )
```

to determine whether an `EvilCreature` is also a `MagicUser`.

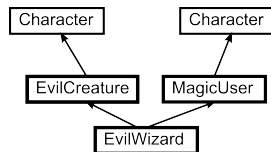


Virtual inheritance

- What happens in the previous example if we have a general **Character** class?
- Obviously every **EvilCreature** is a **Character** and so is every **MagicUser**:

```
class MagicUser : public  
    Character{};  
class EvilCreature : public  
    Character{};
```

- But now **EvilWizard** has two **Character** bases
- Anything stored in **Character** will be duplicated, and referring to the **Character** base of **EvilWizard** is ambiguous.



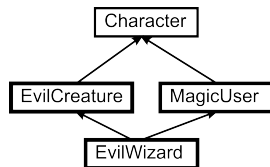
Virtual inheritance

- We can correct this by using **virtual inheritance**

```
class MagicUser : public  
    virtual Character{};  
class EvilCreature : public  
    virtual Character{};
```

- Now **EvilWizard** has a unique **Character** base
- It can be consistently cast to something of type **Character**
- Further, the following is now valid:

```
Character* c = new EvilWizard;
```



Outline

- 29 Inheritance
- 30 Virtual functions
- 31 Polymorphism
- 32 Classes and static**

Static in member functions

- Recall that **static** variables are local to a function, but are initialized only once and persist throughout the program's run
- The same applies to **static** variables in a class's member-function
- These are not unique to each class instance:

```
int A::f(int x) {  
    static int firstVal = x;  
    return firstVal;  
}
```

```
A a,b;  
int x = a.f(5); // x == 5  
int y = b.f(6); // y == 5
```

Static member data

- It is permissible to have **static** member data within a class
- This data is then available to all instances of the class, similar to global variables, but with class access-permissions
- Ordinary member data has space allocated when an instance of the class is allocated
- Static member data must be allocated exactly once per program
- This is done outside the class definition:

```
class A{  
    static double myData;  
};  
  
double A::myData = 9.80665;
```

- For constant static variables of integer or enumeration type only, the initialization can be done inside the class definition:

```
class A{  
    static const int myVal = 9;  
};
```

Static member functions

- Class member functions may also be static.
- This means that they do not need a class object on which to be called:

```
class A{  
    static int f();  
};  
  
int A::f() {  
    return 9;  
}  
  
int x = A::f();
```

- Static member functions may not have **const** or virtual qualifiers

Use of static

- One use for **static** functions is the following:
- Suppose we have a class which must only be instantiated once in a program
- We could use a global variable, but this fails to prevent other functions from creating new instances, and means that the initialization of the variable is situated away from the class
- This is known as a Singleton, and can be implemented as:

```
class SimlParams{
public:
    static const SimlParams* inst() {
        static SimlParams* s = new SimlParams;
        return s;
    }
private:
    SimlParams();
};
```

Use of the Singleton

- Since the constructor is private, this prevents other functions from creating a different instance of `SimlParams`.
- Since `s` is static, only a single instance of `SimlParams` is created, and this is returned every time `inst` is called.
- This construct is used as:

```
const SimlParams* params = SimlParams::inst();
```

- Note that we do not require an object of type `SimlParams` to call `inst`.
- This construct can be used anywhere in the code to reference the unique instance of `SimlParams`
- Any misuse of the class, such as multiple instantiations, is prevented.