C++ - Pre-lecture 8

Polymorphism and abstract base classes

Caterina Doglioni, PHYS30762 - OOP in C++ 2023

MANCHESTER 1824
The University of Manchester

Part 0.1: in this lecture

In this lecture

- **Part 1:** A recap of OOP and inheritance, intro to polymorphism [video 1]
- **Part 2:** Polymorphism [video 2]
- **Part 3:** Abstract classes [video 3]
- **Part 4:** Examples of polymorphism and abstract classes [video 4]
- **Part 5 (optional)**: Unified Modelling Language [video 6]

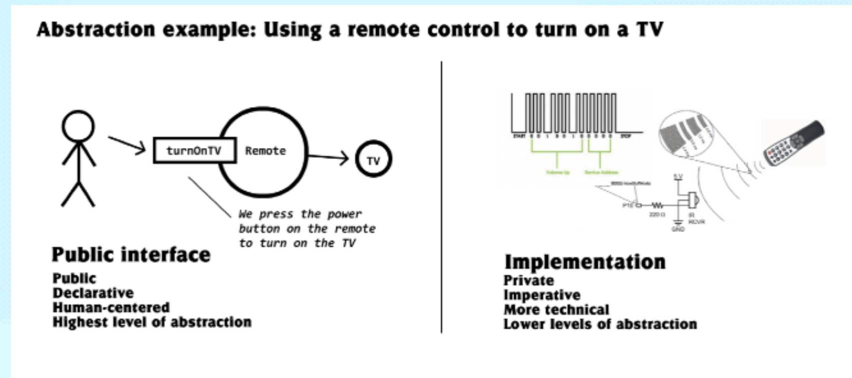
Y3 teaching review: *we were asked to split lectures in shorter videos, but we are working on a "playlist" option if you want to watch everything at once (UofM video service doesn't provide this). We only have one set of slides per pre-lecture to avoid possible confusion with links.*

MANCHESTER 1824
The University of Manchester

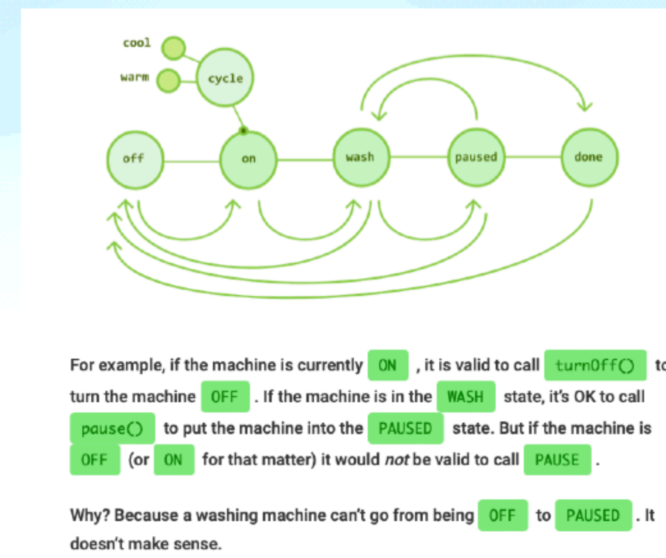
Part 1 / Video 1:
recap of OOP and inheritance, introduction to polymorphism

What is Object-Oriented Programming?

Most describe it in terms of these 4 principles:



- **Abstraction**
  separate interface
  and implementation
  (remote control example)



- **Encapsulation:**
  keep data private, alter properties
  via methods only (washing machine example)

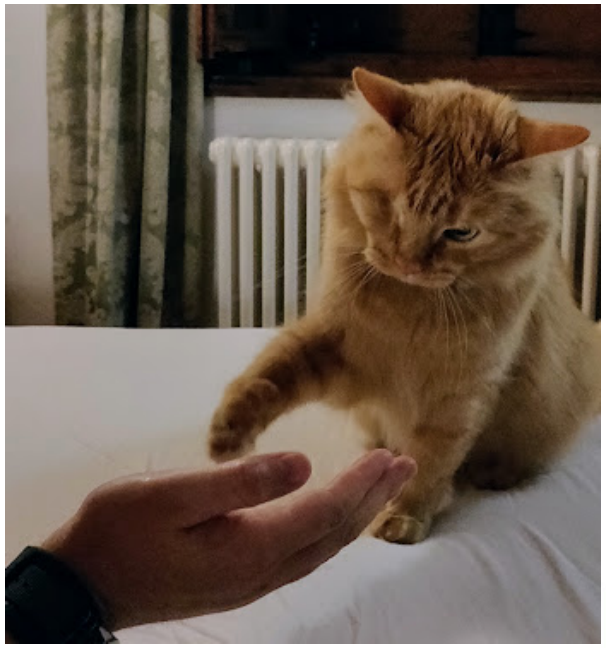
- **Inheritance:**
  classes can be based on other classes to avoid code duplication

- **Polymorphism [today's lecture]:**
  can decide **at run-time** what methods to invoke for a certain class, based on the object itself

MANCHESTER 1824
The University of Manchester

The real-life example

- Class name: **cat**
  - properties = ***data members***
    - ***name***: *Bob*
    - ***fur color***: *ginger*
    - ***eye(s)***: *yellow*
  - functionalities = ***member functions***
    - ***sleep()***
    - ***high_five_with_claws()***



MANCHESTER 1824
The University of Manchester

Inheritance

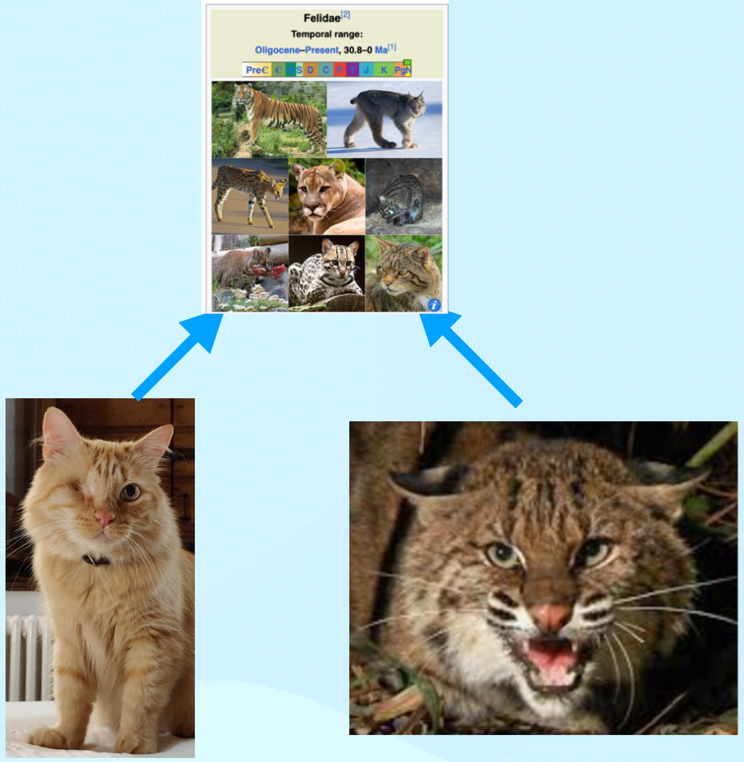
- Base class name: **cat** (member of the felidae family)
  - ***data members*** *(*name, fur color, eye(s)*)*
  - ***member functions*** *(*sleep()*)*

Why this arrow?
We'll see at the end
of this pre-lecture…

- Derived class name: **domestic_cat** (felis catus)
  - ***same*** *data members / functions as base class*
    - *functions can be* ***overridden***
      - Bob the cat calls the `snore()` function inside sleep(), other derived feline classes don't
  - *can also* ***add specialized functions (***high_five_with_claws***)***
    - a quieter / more polite feline would not do this

- When you design your code (& before writing it!), think of
  - What is common → base class
    - You can also decide to do something different in the derived class's function (overriding), but the action is the same
  - What is not specific → derived class

Polymorphism



- Base class name: **cat**
  - ***data members*** *(*name*,* fur color*,* eye(s)*)*
  - ***member functions (***sleep(), **react_to_being_pet()***)*
- Derived class name: **domestic_cat**
- Derived class name: **non_domestic_cat**

} They will most likely react differently to human touch:
domestic_cat will most likely accept pets
non_domestic_cat may see the human as food

- What if you want to let the user of your class decide whether to instantiate a domestic or non-domestic cat *at runtime*?

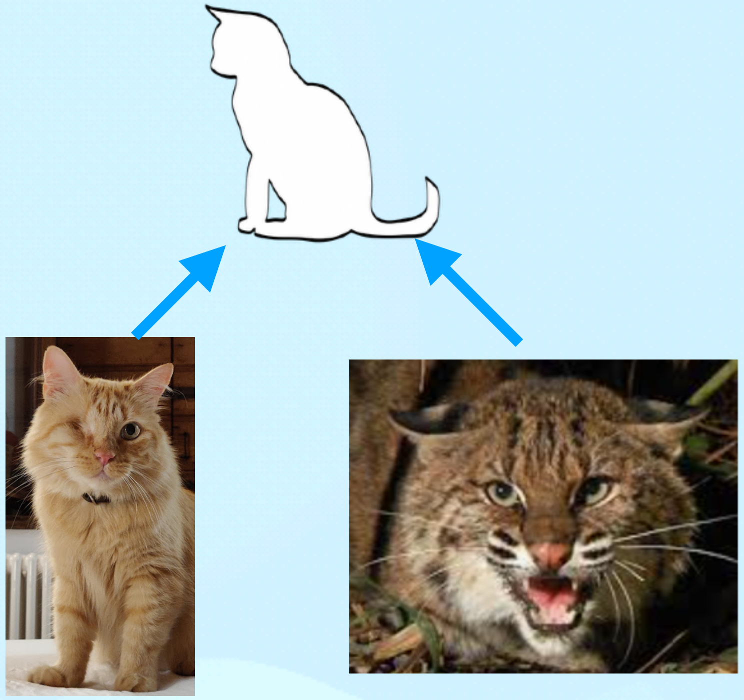
  - real life: you don't know if someone tamed those bobcats roaming in your yard, they look really cute and you want to (literally) try your hand at petting them

  - programming life: you want to have the freedom to decide what behaviour your class will have depending on its type, e.g. when you are filling a vector which can only have one type

- This is where **polymorphism** becomes useful

MANCHESTER 1824
The University of Manchester

Abstract base classes



- Base class name: **cat**
  - ***data members*** *(*name, fur color, eye(s)*)*
  - ***member functions (***sleep(), react_to_being_pet()*)*
- Derived class name: **domestic_cat**
- Derived class name: **non_domestic_cat**
- A cat is either a domestic_cat or a non_domestic_cat
  - (in this perspective, farm/feral cats are domestic cats)
  - no need to instantiate a generic “cat” object
    → make the base class **abstract** so it’s **only an interface**

In the rest of the lecture, we will use particles as examples

Part 2 / Video 2:
polymorphism in practice

Polymorphism in general

- Polymorphism: poly (**many**) and morph (**form**)
- It is a pillar of Object Oriented Programming
- It gives us the ability to create classes with the **same structure** (e.g. function names) but with **different methods**
- It makes use of **inheritance** and **function overriding** (not overloading)
- Of central importance is the concept of the base class pointer
- As usual, easiest to understand by example

MANCHESTER 1824
The University of Manchester

Elements of polymorphism:

The base class, no pointer yet

- Let's have a base class (particle) and a derived class (ion)

```cpp
// PL8/baseclasspointer.cpp
// Demonstrates base class pointers
// Niels Walet, Last modified 03/12/2019
#include<iostream>
class particle
{
protected:
  double charge{};
public:
  particle(double q) : charge{q} {}
  void info(){std::cout<<"particle: charge="<<charge<<"e"<<std::endl;}
};

class ion : public particle
{
private:
  int atomic_number;
public:
  ion(double q, int Z) : particle{q}, atomic_number{Z} {};
  void info()
  {
    std::cout<<"ion: charge="<<charge
        <<"e, atomic number="<<atomic_number<<std::endl;
  }
};
int main()
{
  particle particle_1{1}; // proton
  ion ion_1{2,2}; // helium nucleus
  particle_1.info();
  ion_1.info();
  return 0;
}
```

- ion has more properties than particle, and inherits properties and functions from particle
- when calling the info() function, its behaviour depends on the type of object that is instantiated

```
urania277@medram Prelecture8 % ./baseclasspointer
particle: charge=1e
ion: charge=2e, atomic number=2
```

- this is function **overriding**

Code on GitHub at: Prelecture8/baseclasspointer.cpp

Elements of polymorphism:

Basic base class pointers

```cpp
// PL8/baseclasspointer2.cpp
// Demonstrates use of baseclass pointer for polymorphism
// Niels Walet, Last modified 03/12/2019
#include<iostream>
class particle
{
protected:
  double charge;
public:
  particle(double q) : charge{q}{}
  void info(){std::cout<<"particle: charge="<<charge<<"e"<<std::endl;}
};

class ion : public particle
{
private:
  int atomic_number;
public:
  ion(double q, int Z) : particle{q}, atomic_number{Z} {}
  void info()
  {
    std::cout<<"ion: charge="<<charge
        <<"e, atomic number="<<atomic_number<<std::endl;
  }
};
int main()
{
  particle particle_1{1}; // proton
  ion ion_1{2,2}; // helium nucleus
  particle_1.info();
  ion_1.info();
  particle *particle_pointer; // pointer to particle
  particle_pointer=&particle_1; // point to particle_1
  particle_pointer->info();
  particle_pointer=&ion_1; // point to ion_1 (allowed!)
  particle_pointer->info();
  return 0;
}
```

- this time, we also make use of the pointer to the objects
  - note: you can use a base class pointer to describe a derived class
  - test: can you use a base class pointer to call a specialised function in the derived class object? No!
- however, this does **not** call the overridden function for the derived class, it calls the base class function!

```
urania277@medram Prelecture8 % ./baseclasspointer2
particle: charge=1e
ion: charge=2e, atomic number=2
particle: charge=1e
particle: charge=2e
```

Code on GitHub at: Prelecture8/baseclasspointer2.cpp

MANCHESTER 1824
The University of Manchester

Elements of polymorphism:
Base class pointers and virtual functions

- In order to call the derived class function, we need to make the base class function `virtual`
  - this is a C++ keyword that will appear later as well (*pure virtual functions*)
  - if a function is virtual in a base class, then it is possible for the derived class's function will be called from a pointer that points to an object of the derived class
  - test: can you use a base class pointer to call a specialised function in the derived class object? No, but you can still implement this behaviour with a virtual function
- this solves the restriction of having only one type in a vector:
  - can fill the vector with pointers of the base class…
  - …allocated dynamically to be pointing to derived class objects
  - important: for each `new` there must be a `delete`

MANCHESTER 1824
The University of Manchester

Elements of polymorphism:

Base class pointers and virtual functions

```cpp
// PL8/baseclasspointer3.cpp
// Demonstrates the use of a baseclass pointer
// Niels Walet, Last modified 03/12/2019
#include<iostream>
using namespace std;
class particle
{
protected:
  double charge;
public:
  particle(double q) : charge{q}{}
  virtual void info(){std::cout<<"particle: charge="<<charge<<"e"<<std::endl;}
};

class ion : public particle
{
private:
  int atomic_number;
public:
  ion(double q, int Z) : particle{q}, atomic_number{Z}{}
  void info(){
    std::cout<<"ion: charge="<<charge
        <<"e, atomic number="<<atomic_number<<std::endl;
  }
};
int main()
{
  particle particle_1{1}; // proton
  ion ion_1{2,2}; // helium nucleus
  particle_1.info();
  ion_1.info();
  particle *particle_pointer; // pointer to particle
  particle_pointer=&particle_1; // point to particle_1
  particle_pointer->info();
  particle_pointer=&ion_1; // point to ion_1 (allowed!)
  particle_pointer->info();
  return 0;
}
```

Code on GitHub at: Prelecture8/baseclasspointer3.cpp

- this is the same code as before, but we added the keyword `virtual` in the base class, before the declaration of the function that will be overridden by the derived class
  - to compare with later part of the lecture: you can also not have an implementation of this function in the derived class, and when calling it the compiler will default to the base class. Convince yourself by commenting `ion::info` out…
- using the base class pointer to the derived class object, this **does** call the overridden function for the derived class

```
urania277@medram Prelecture8 % ./baseclasspointer3
particle: charge=1e
ion: charge=2e, atomic number=2
particle: charge=1e
ion: charge=2e, atomic number=2
```

Elements of polymorphism:
Virtual destructors

- Recap: **destructors** are called whenever an object goes **out of scope**
    - This usually happens at the end of the function where the object is first instantiated

- Destructors should be used to **delete memory** when using dynamic arrays in classes

- **Advice:** when using base class pointers, make sure your **base class destructor is virtual**

```
virtual ~particle(){std::cout<<"Calling base class destructor"<<std::endl;}
~ion(){std::cout<<"Calling derived class destructor"<<std::endl;}
```

    - That way, the appropriate destructor is called when object (accessed with base class pointer) goes out of scope
    - if this is not done, and the derived destructor contains `delete`s, only the base class destructor will be called ⟹ ☠️memory leak☠️

- If base class destructor is not a virtual function, this will always be called in preference to any derived class destructor!

Code on GitHub at: Prelecture8/baseclasspointer3.cpp

MANCHESTER 1824
The University of Manchester

Polymorphism summary

- We have just demonstrated **polymorphism** in action!
    - Used **inheritance** to create base and derived classes
    - Used **function overriding** to change the action of `info` in derived class
    - Defined a **base class pointer** to point to either type of object
    - Made `info` a **virtual function** to access correct version of `info` with pointer
- This is **run-time polymorphism**:
  only while running the code can we decide what version of info to call
- *Note*: polymorphism relies on **overridden virtual members** (otherwise base class pointer always refers to base class member function)
- **Summary**: **action depends on which object base class pointer is pointing to in hierarchy**
- Classes used in this way (with virtual functions) known as polymorphic classes

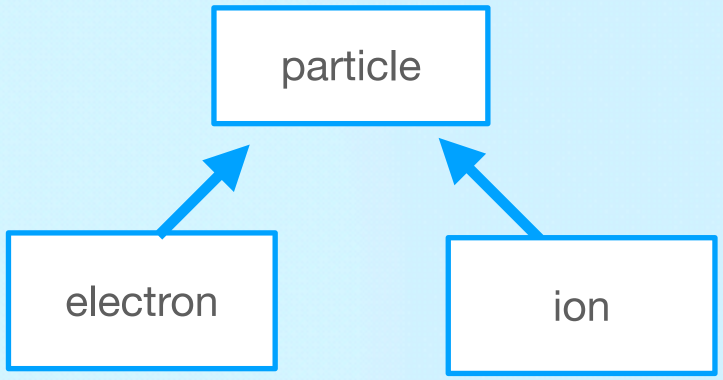
Code on GitHub at: Prelecture8/baseclasspointer3.cpp

MANCHESTER 1824 The University of Manchester

Part 3 / Video 3: abstract base classes

Abstract base class

How to make a base class abstract



- In the previous example, objects could be instantiated from either the base or derived class
- But base class was special: it contains the virtual functions, and its type is used when declaring base class pointer
- We can take this further: we can (and sometimes should) use the **base class as interface only**

  1. Use base class to declare **virtual functions only** (*pure virtual functions*)
  2. In the derived class we now **must** override the virtual functions and define their action– otherwise the derived class is also abstract (which may be what you want, but you cannot instantiate an abstract class)
  3. The derived classes can still contain their own data and member functions

- A base class that only declares existence of virtual functions is known as an abstract base class
  - Formally: a base class becomes abstract base class when converting at least one virtual function to a pure virtual function
- Let’s see how ...

MANCHESTER 1824 The University of Manchester

Our first abstract class

The particle

```
5   class particle
6   {
7   public:
8     virtual ~particle(){} // Need this!
9     virtual void info()=0; // pure virtual function
10  };
```

Find this code at: Prelecture8/abstract.cpp

- Only abstract functions in base class (which now can't be instantiated)

```
32  int main()
33  {
34    particle abstractParticle;
```

```
/Users/urania277/Work/PHYS30762/prelecture-codes/Prelecture8/abstract.cpp: In function 'int main()':
/Users/urania277/Work/PHYS30762/prelecture-codes/Prelecture8/abstract.cpp:34:12: error: cannot declare variable 'abstractPa
rticle' to be of abstract type 'particle'
   34 |   particle abstractParticle;
      |            ^~~~~~~~~~~~~~~~
/Users/urania277/Work/PHYS30762/prelecture-codes/Prelecture8/abstract.cpp:5:7: note:   because the following virtual functi
ons are pure within 'particle':
    5 | class particle
      |       ^~~~~~~~
/Users/urania277/Work/PHYS30762/prelecture-codes/Prelecture8/abstract.cpp:9:16: note:     'virtual void particle::info()'
    9 |   virtual void info()=0; // pure virtual function
      |                ^~~~

Build finished with error(s).
```


MANCHESTER 1824
The University of Manchester

Using abstract classes

```
#include<iostream>
class particle
{
public:
  virtual ~particle(){} // Need this!
  virtual void info()=0; // pure virtual function
};
class electron : public particle
{
private:
  int charge;
public:
  electron() : charge{-1} {}
  ~electron() {std::cout<<"Electron destructor called"<<std::endl;}
  void info() {std::cout<<"electron: charge="<<charge<<"e"<<std::endl;}
};
class ion : public particle
{
private:
  int charge,atomic_number;
public:
  // Note constructor short-hand!
  ion(int q, int Z) : charge{q},atomic_number{Z} {}
  ~ion() {std::cout<<"Ion destructor called"<<std::endl;}
  void info() {std::cout<<"ion: charge="<<charge
            <<"e, atomic number="<<atomic_number<<std::endl;}
};

int main()
{
  particle *particle_pointer = new ion{1,2};
  particle_pointer->info();
  delete particle_pointer;
  particle_pointer = new electron;
  particle_pointer->info();
  delete particle_pointer;
  return 0;
}
```

Find this code at: Prelecture8/abstract.cpp

- **Pure virtual functions** have no method in base class: **must be implemented in derived classes**
- We use `particle` to declare what functions common to all derived classes (and as name of base class pointer)
- All objects can be accessed using a base class pointer through particle - known as an **interface**
- Derived classes define **members specific** of each class type
  - they must contain an implementation of the pure virtual functions
- this is another example of polymorphism:
  - **one interface**
  - **multiple methods**

Polymorphism summary

- We have just demonstrated **polymorphism** in action!
  - Used **inheritance** to create base and derived classes
  - Used **function overriding** to change the action of `info` in derived class
  - Defined a **base class pointer** to point to either type of object
  - Made `info` a **virtual function** to access correct version of `info` with pointer
- This is **run-time polymorphism**:
  only while running the code can we decide what version of info to call
- *Note*: polymorphism relies on **overridden virtual members** (otherwise base class pointer always refers to base class member function)
- **Summary**: **action depends on which object base class pointer is pointing to in hierarchy**
- Classes used in this way (with virtual functions) known as polymorphic classes

Code on GitHub at: Prelecture8/baseclasspointer3.cpp

MANCHESTER 1824 The University of Manchester

Summary of polymorphism

"Buzzword summary"

- We want to design classes for a **set of related objects**
- We create a **base class** that contains **members** (data and functions) **applicable to all objects within the set**
- We make those **functions** we wish to **override** (same name/parameters different method) **virtual functions**
- If we **do not need to create objects of the base class** (and use it solely as an **interface**), we make our virtual functions **pure virtual functions**, assigning them to zero in the base class
- Our base class is now known as an **abstract base class**;
  it is only accessible to derived classes
- We can call each object's virtual member functions with a **single base class pointer**

MANCHESTER 1824
The University of Manchester

Part 4 / Video 4:
more polymorphism and abstract classes by example

Y3 teaching review: *There are concerns that it is hard to see how the content presented can be applied in different coding challenges, and how to choose the most suitable technique for solving a problem. Some more explicit examples on how to apply the coding skills learned that stress where it is appropriate to use them would help.*

Where is polymorphism useful?

Some examples of *dynamic runtime behaviour*

- When you need to loop over derived classes of different sorts
  - You can do this using a vector, where you allocate the content dynamically

```cpp
int main()
{
  // Array of base and derived objects, one particle and one ion
  particle *particle_array[2];
  particle_array[0] = new particle{2}; // He
  particle_array[1] = new ion{1,2};    // He+
  particle_array[0]->info(); // print info for particle
  particle_array[1]->info(); // print info for ion
  delete particle_array[0]; particle_array[0]=0;
  delete particle_array[1]; particle_array[1]=0;
  return 0;
}
```

Code on GitHub at: Prelecture8/mixedarray.cpp

Where is polymorphism useful?

Some examples of *dynamic runtime behaviour*

- When you have a function that can take different derived classes as argument
  - With polymorphism, you don't need to make overloaded versions of the functions with different argument for each type of derived class (because code duplication is bad)

```cpp
void prettyParticlePrinter (particle* theParticle) {

   //add some ascii art
   std::cout << "_U__U__U__U__U__U__U__U__U__U__U__U__U_" << std::endl;
   //then call the function
   theParticle->info();

}

int main()
{
  // Array of base and derived objects, one particle and one ion
  particle *particle_array[2];
  particle_array[0] = new particle{2}; // He
  particle_array[1] = new ion{1,2};    // He+

  prettyParticlePrinter(particle_array[0]);
  prettyParticlePrinter(particle_array[1]);

  delete particle_array[0]; particle_array[0]=0;
  delete particle_array[1]; particle_array[1]=0;
  return 0;
}
```

Code on GitHub at: Prelecture8/mixedfunction.cpp

Where is polymorphism needed?

Virtual destructors

- If you have derived class data members that are dynamically allocated, you need a **virtual destructor**
  - Otherwise, memory leak
  - In general, good practice to use virtual destructors when making use of inheritance
- Let's take the previous example and modify it together…

MANCHESTER 1824
The University of Manchester

Where are abstract classes useful?

- Where you have a common concept for an object, but the actual objects you're going to use have different behaviours

- You can use abstract base classes for looping in both arrays and vectors

- In general, this is (like) polymorphism, it is the design of your overall code that is different → thinking about why / how you are designing something becomes very important for good OOP software!

```cpp
int main()
{
  // Array of 2 base class pointers
  particle **particle_array = new particle*[2];
  particle_array[0] = new ion{1,2};
  particle_array[1] = new electron;
  particle_array[0]->info(); // print info for electron
  particle_array[1]->info(); // print info for ion
  // clean-up
  delete particle_array[0];
  delete particle_array[1];
  delete[] particle_array;
  return 0;
}
```

Code on GitHub at: Prelecture8/polymorphicarray.cpp

```cpp
int main()
{
  std::vector<particle*> particles;
  particles.push_back(new ion{1,3});
  particles.push_back(new electron);
  particles[0]->info();
  particles[1]->info();
  std::cout<<"particles has size "<<particles.size()<<std::endl;
  for (auto particle_it=particles.begin();
       particle_it<particles.end();
       ++particle_it) delete *particle_it;
  particles.clear();
  std::cout<<"particle_vector now has size "<<particles.size()<<std::endl;
  return 0;
}
```

Code on GitHub at: Prelecture8/polymorphicvector.cpp

MANCHESTER 1824 The University of Manchester

Bonus content / Video 5
Unified Modelling Language

[not required for assignments/projects, use it as you wish - I find it rather useful]

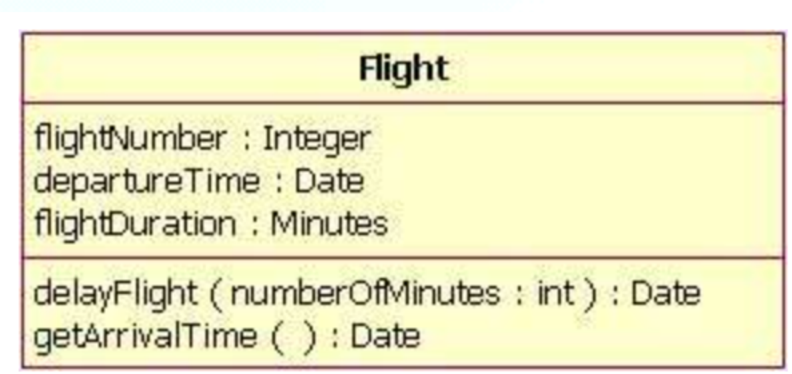
Unified Markup Language (UML)

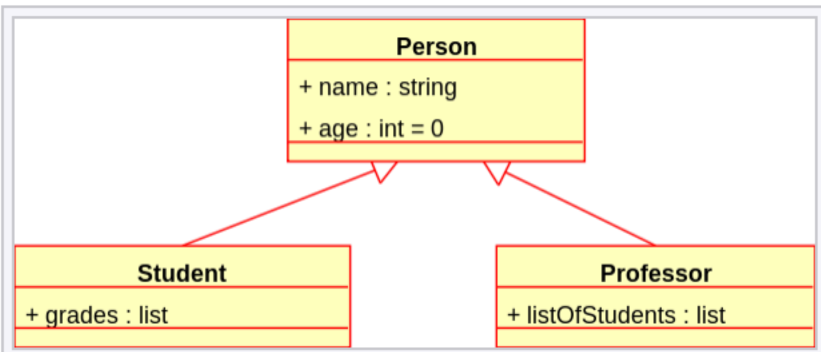
Useful for designing before coding

Wikipedia: The **Unified Modeling Language** (**UML**) is a general-purpose, developmental modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system.[1]

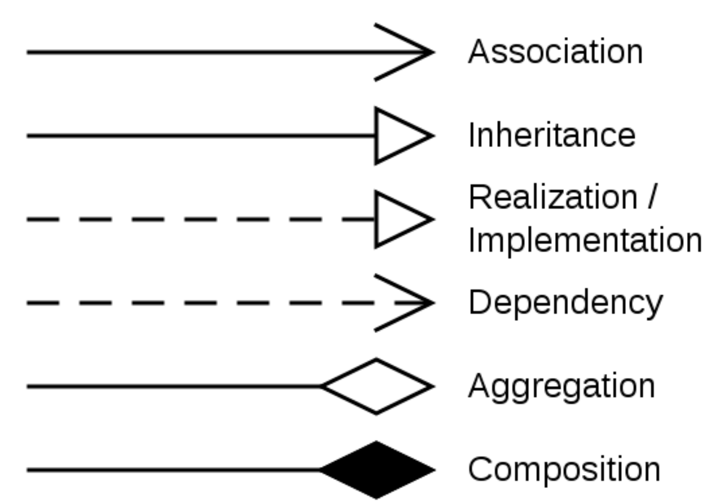
We can use it in OOP C++ to represent the structure of our code in terms of classes and relationships - **class diagrams**

A good tutorial: https://developer.ibm.com/articles/the-class-diagram/





Class diagram showing generalization between the superclass *Person* and the two subclasses *Student* and *Professor*



https://developer.ibm.com/articles/the-class-diagram/

https://en.wikipedia.org/wiki/Class_diagram

MANCHESTER 1824
The University of Manchester

Let’s draw!

Many alternatives out there, but a free online one (as long as you don’t have too many projects) is at: https://www.lucidchart.com/pages/

MANCHESTER 1824
The University of Manchester