C++ - Pre-lecture 8

Polymorphism and abstract base classes + enumerators and switches

Caterina Doglioni & Charanjit Kaur, PHYS30762 - OOP in C++ 2024

MANCHESTER 1824
The University of Manchester

Part 0.1: in this lecture

In this lecture

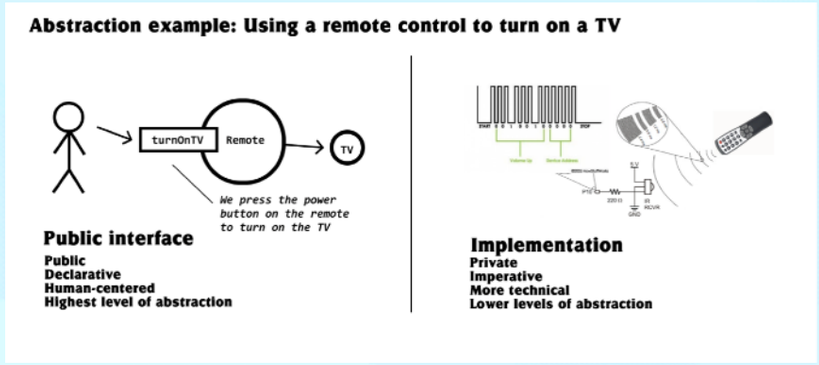
- **Part 1:** A recap of OOP and inheritance, intro to polymorphism [video 1]
- **Part 2:** Polymorphism [video 2]
- **Part 3:** Abstract classes [video 3]
- **Part 4:** Examples of polymorphism and abstract classes [video 4]
- **Part 5:** Enumerators and switches [video 5]
- **Part 6 (optional, but can lead to bonus marks in report)**: Unified Modelling Language [video 6]

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

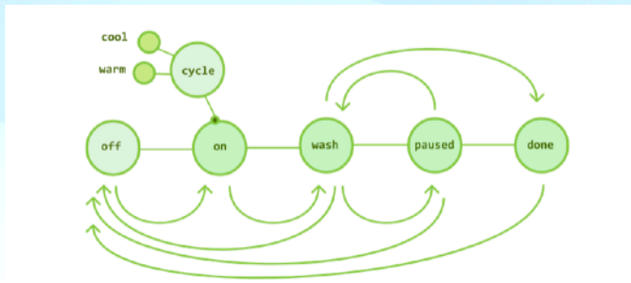
The University of Manchester

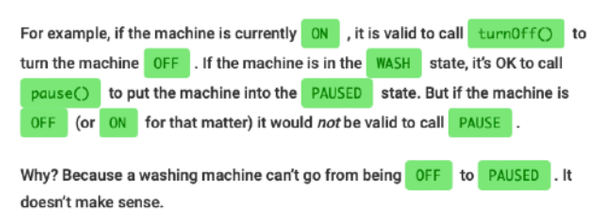
What is Object-Oriented Programming?

Most describe it in terms of these 4 principles:



- Abstraction
  separate interface and implementation (remote control example)

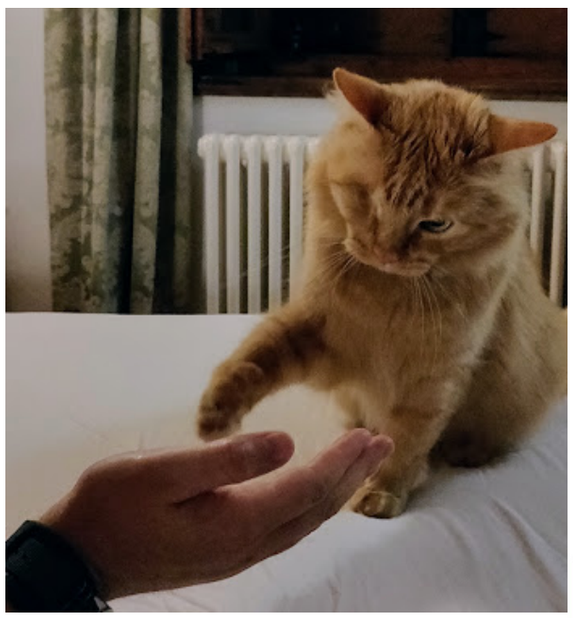




- Encapsulation:
  keep data private, alter properties via methods only (washing machine example)
- Inheritance [lecture 7]:
  classes can be based on other classes to avoid code duplication
- Polymorphism [today's lecture]:
  can decide **at run-time** what methods to invoke for a certain class, based on the object itself

The real-life example

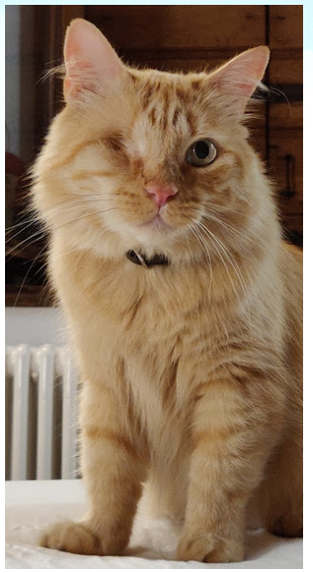
- Class name: **cat**
  - properties = ***data members***
    - ***name****: Bob*
    - ***fur color****: ginger*
    - ***eye(s)****: yellow*
  - functionalities = ***member functions***
    - ***sleep()***
    - ***high_five_with_claws()***



Inheritance

- Base class name: **cat** (member of the felidae family)
  - ***data members*** *(*`name, fur color, eye(s)`*)*
  - ***member functions*** *(*`sleep()`*)*
- 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
  - *More specific cats can also* ***add specialized functions (***`high_five_with_claws`***)***
    - a quieter / more polite feline would not do this

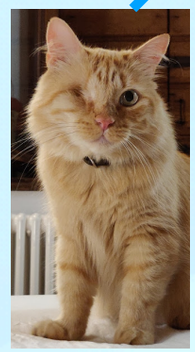
Why this arrow?
We'll see at the end of this pre-lecture…



- 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 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

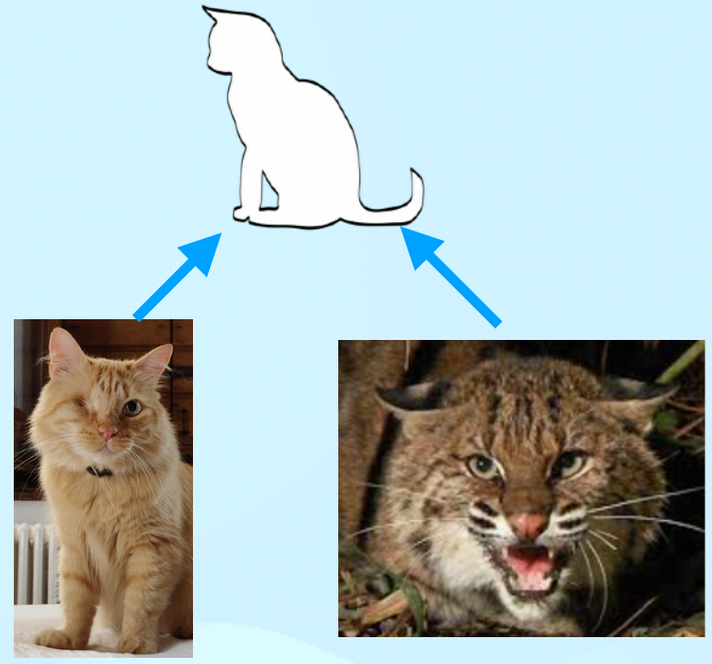
- What if you want to let the user of your class decide whether to instantiate a domestic or non-domestic cat *at runtime*?

  - 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

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

MANCHESTER 1824
The University of Manchester

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 and parameters) but with **different functionalities**
- It makes use of **inheritance** and **function overriding** (not overloading)
- Of central importance is the concept of the **base class pointer**
  - When you use a base class pointer you can **decide at run-time what functions to call** (they can be from the base or from the derived class)
  - It's important to know what the compiler will do by default, and to use the **virtual** keyword when necessary (see abstract classes in video 3)

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 by yourself: 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

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 on your own: 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 if you use raw pointers (in general you shouldn't!): for each `new` there must be a `delete`

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
  - *safety warning when using raw pointers and not smart pointers:* 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!

Polymorphism summary so far

- 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

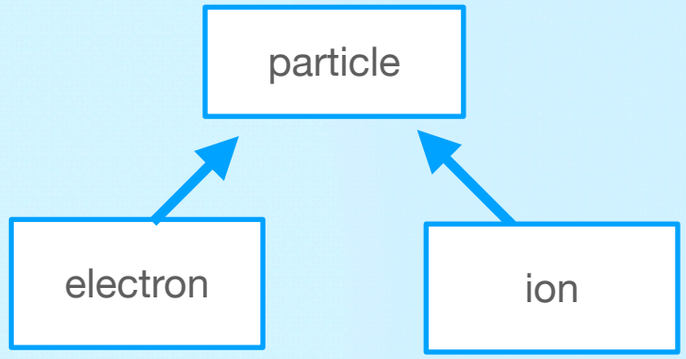
Code on GitHub at: Prelecture8/baseclasspointer3.cpp

Part 3 / Video 3: abstract base classes

MANCHESTER 1824
The University of Manchester

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 the 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 ...

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).
```

Using abstract classes

- **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**

Find this code at: Prelecture8/abstract.cpp

Using abstract classes: example

```cpp
// PL8/abstract.cpp
// The use of an abstract base class
// Niels Walet, last updated 06/01/2022
#include<iostream>
#include<memory>

class particle
{
public:
  //the print-out here is to show you that the destructor of the base class is called after that of the derived class
  virtual ~particle(){std::cout << "Particle destructor called" << std::endl;} // 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;}
};
```

```cpp
int main()
{
  // Creates a ion but with a pointer to the abstract base class 'particle'
  std::unique_ptr<particle> particle_pointer_for_ion = std::make_unique<ion>(1,2);
  particle_pointer_for_ion->info();
  //
  std::unique_ptr<particle> particle_pointer_for_electron = std::make_unique<electron>();
  particle_pointer_for_electron->info();
  //
  // Try to uncomment this line and see what kind of error you get...
  // You cannot instantiate an abstract class!
  // std::unique_ptr<particle> particle_pointer_for_electron = std::make_unique<particle>(-1);

  return 0;
}
```

Find this code at: Prelecture8/abstract.cpp

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**

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

Trying to simultaneously answer Y3 2023 teaching review “want more examples and fewer concepts” + Y3 2024 teaching review “want more concepts and fewer examples”…

The University of Manchester

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
  std::unique_ptr<particle> particle_array[2];
  particle_array[0] = std::make_unique<particle>(2); // generic particle with charge q
  particle_array[1] = std::make_unique<ion>(1,2);    // He+
  particle_array[0]->info(); // print info for particle
  particle_array[1]->info(); // print info for ion
  //no need for delete, memory management is done for you by the smart pointer!
  return 0;
}
```

Code on GitHub at: Prelecture8/mixedarray.cpp

Where is polymorphism useful?

Some examples of *dynamic runtime behaviour*

- Use case: you want 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 (const 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
  std::unique_ptr<particle> particle_array[2];
  particle_array[0] = std::make_unique<particle>(2); // He
  particle_array[1] = std::make_unique<ion>(1,2);    // He+
  //note below: get() does not change who owns the pointer,
  //so the smart pointer will still be responsible for returning the memory (deleting) as usual
  prettyParticlePrinter(particle_array[0].get());
  prettyParticlePrinter(particle_array[1].get());
  //no need for delete, memory management is done for you by the smart pointer!
  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
    - The “rule of 3/rule of 5” from lecture 6 still applies for smart pointers!
- Let’s take the previous example and modify it…

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!

Exception to the vector/smart pointer rule: arrays and raw pointers

```
36  int main()
37  {
38      // Array of 2 base class pointers
39      particle **particle_array = new particle*[2];
40      particle_array[0] = new ion{1,2};
41      particle_array[1] = new electron;
42      particle_array[0]->info(); // print info for electron
43      particle_array[1]->info(); // print info for ion
44      // clean-up
45      delete particle_array[0];
46      delete particle_array[1];
47      delete[] particle_array;
48      return 0;
49  }
```

Code on GitHub at: Prelecture8/polymorphicarray.cpp

Vectors and smart pointers: much better!

```
int main()
{
  std::vector<std::unique_ptr<particle>> particles;
  particles.push_back(std::make_unique<ion>(1,3));
  particles.push_back(std::make_unique<electron>());
  particles[0]->info();
  //particles[1]->info();
  std::cout<<"particles has size "<<particles.size()<<std::endl;
  return 0;
}
```

Code on GitHub at: Prelecture8/polymorphicvector.cpp

Part 5 / Video 5

Enumerators and switches

MANCHESTER 1824
The University of Manchester

Enumerators

- In your assignments (and in real programming life), you often have characteristics of your class / choices that the user makes that correspond to string but are “quantised” (only a limited number of choices)
  - Example: particle types, names of calorimeter layers…
  - There are many ways to implement this, but one of the simplest / most efficient ones that is built-in C and C++ is the **enumerator**
- Enumerator (`enum`): special type of variable that associates a string to an int
  - You can use it in vectors as an `int` can correspond to indices
  - You shouldn’t mix hardwired `int`s and `enum`s in your code for the same things, since enums are there to make things more readable

Enumerators: example

```cpp
// PL8/enum.cpp
// Demonstrate use of enum
// Caterina Doglioni, Last modified 14/03/2024
#include<iostream>
#include<vector>
#include<string>

int main () {

    enum particleType {
        ELECTRON = 0, //note the comma, not the semicolon
        PROTON = 1, //the ints here can be anything you choose
        MUON = 2,
    };

    //simple example: assume that you will always want particles in this order,
    //and want a short-hand for indices that is not numbers
    std::vector<std::string> vector_of_particle_names(3);
    vector_of_particle_names[ELECTRON] = "Electron";
    vector_of_particle_names[PROTON] = "Proton";
    vector_of_particle_names[MUON] = "Muon";

    std::cout << "The index of vector_of_particle_names[ELECTRON] is " << ELECTRON << std::endl;
    std::cout << "Its content is: " << vector_of_particle_names[ELECTRON] << std::endl;

}
```

Code on GitHub at: Prelecture8/enum.cpp

Switches

- What about nested if loops?
  - Nothing wrong with them (`if...else...`)
  - But there is a way to make them look “nicer”
  - When you have a number of choices and can only take one of them
    - There are many ways to implement this, but one of the simplest / most efficient ones that is built-in C and C++ is the **switch**
- Enumerator: construct that lets you decide what to do depending on a number of **cases** (usually associated to an int or an enumerator)

Switch: syntax and considerations

- `c1`, `c2`, `c3`: conditions
- `statements1...n`: code to execute
- The `break` statement is not mandatory but it is recommended (C++17 compiler will warn you)
  - Compiler will automatically fall through the next case if the previous condition is not satisfied, and end up in the default case

This is pseudo-code

Syntax

```
switch(identifier) {
  case c1 : statements1; break;
  case c2 : statements2; break;
  case c3 : statements3; break;
  ...
  default : statementsn; break;
}
```

```cpp
#include<iostream>
#include<vector>
#include<string>

int main () {

  // the reason why I use CamelCase here is that
  // there is also an enum class (out of the scope of this course)
  // so I identify an enum more with a class than with variable
  enum LanguageType {
        ENGLISH = 0, //note the comma, not the semicolon
        GERMAN = 1, //the ints here can be anything you choose
        FRENCH = 2,
  };

  int language = GERMAN;

  switch (language) {
    case ENGLISH:
      std::cout << "Good morning" << std::endl;
      break;
    case GERMAN:
      std::cout << "Guten Tag" << std::endl;
      break;
    case FRENCH:
      std::cout << "Bonjour" << std::endl;
    break;
    default:
      std::cout << "I do not speak your language";
  }

}
```

Code on GitHub at: Prelecture8/switch.cpp

Bonus content / Video 5

Unified Modelling Language

[can get you some challenge marks in the project report, and I find it rather useful when designing code]

Unified Markup Language (UML)

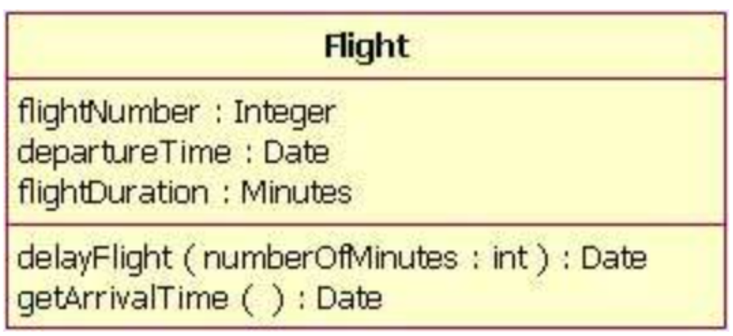
Useful for designing before coding

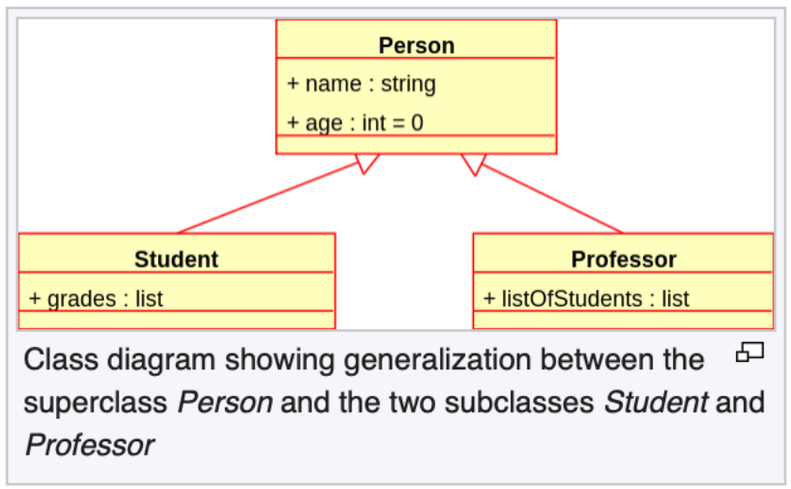
Wikipedia: "The **Unified Modeling Language** (**UML**) is a general-purpose, developmental modeling language in the field of software engineering that is intended to provide a standard way to visualise the design of a system."

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

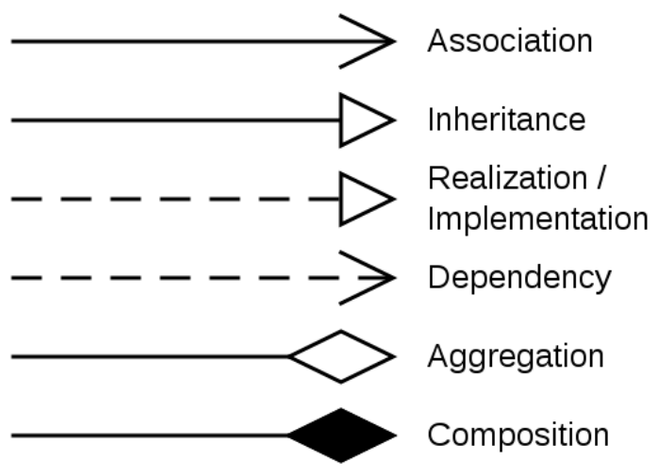
We can use it in OOP C++ to represent the structure of our code in terms of classes and relationships - **class diagrams**

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

Let’s read a UML diagram…

https://www.ge.infn.it/geant4/training/ptb_2009/day3/figures/physics_hadrons.jpg

What are we looking at? https://indico.cern.ch/event/776050/contributions/3237925/attachments/1789252/2914238/PhysLists.pdf

Tip: you can start with this when you are thinking of your project

Let’s draw! This is useful when you’re thinking about the design of your code

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

Tutorial at: `https://www.lucidchart.com/pages/uml-class-diagram`

Let's *automatically* draw a UML diagram! This is useful when you have code and you want to inspect it

Download doxygen here: `https://www.doxygen.nl/`

Quick-start here: `https://www.doxygen.nl/manual/doxywizard_usage.html`