C++ - Pre-lecture 4

Introduction to classes and OOP concepts

Caterina Doglioni, PHYS30762 - OOP in C++ 2023

Part 0.1: in this lecture

In this lecture

- We will take our first steps using *Objects* in Object-Oriented Programming
- [video 1] intro **examples**, leading to a C “object”: **the struct**
- [video 2] a C++ object: **the class**
  - **data members**, public/private
  - **accessors** and **member functions**
- [video 3] the class, refined:
  - separating **interface** and **implementation**
  - functions returning vectors, and iterating over vectors
- [video 4] How to design a class? **Object-oriented programming principles**

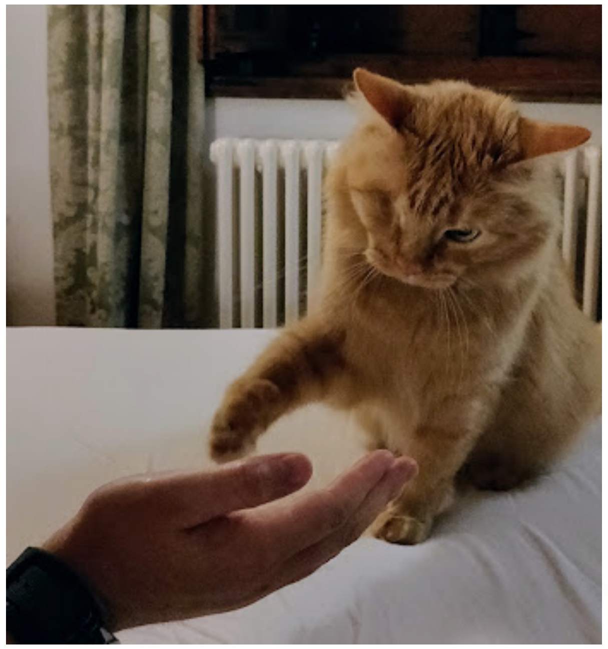
Video 1: examples, and the `struct`

What is an object (a class), conceptually

- Objects ⇔ Classes
  - Think of real objects:
    - defined by their **properties** (nouns)
    - defined by their **functionality** (verbs)
  - Extending this concept to storing and manipulating data:
    - properties = ***data members***
    - functionalities = ***member functions***

A 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()*** $\longrightarrow$



Our example object: a particle

- **Type**: electron
- **Electric charge**: -1
- **Mass**: 511 MeV
- **Momentum**: xxx
- **Energy**: yyy
- **β-factor**: zzz

  (for non-physics-students: this has to do with special relativity, if the particle is travelling at close to the speed of light then this factor is close to 1)

MANCHESTER 1824
The University of Manchester

A struct

A C-inspired structure that holds properties

- A simple way to capture all the properties of an object (which actually originates in the C language) is the `struct`

- Example: consider a *particle object*.
  - We can define a struct to hold its properties (data)

```
struct particle {
  std::string type;
  double mass;
  double momentum;
  double energy;
}
```

  - Its properties are accessed using its name and the “dot” (.) notation

```
particle p; //make an object of type “particle”
p.type=“electron”;
p.mass=0.511;
```

struct, functions and data

How to do something with the struct's properties

- Let's have a function to print out the data

```
void print_data(const struct particle &p) {
std::cout.precision(3); // 2 significant figures
std::cout<<"Particle: [type,m,p,E] = ["<<p.type<<","<< p.mass
<<“,"<<p.momentum<<","<<p.energy<<"]"<<std::endl; return;
}
```

- or calculate the Lorentz factor

```
double gamma(const struct particle &p) {
return p.energy/p.mass; }
```

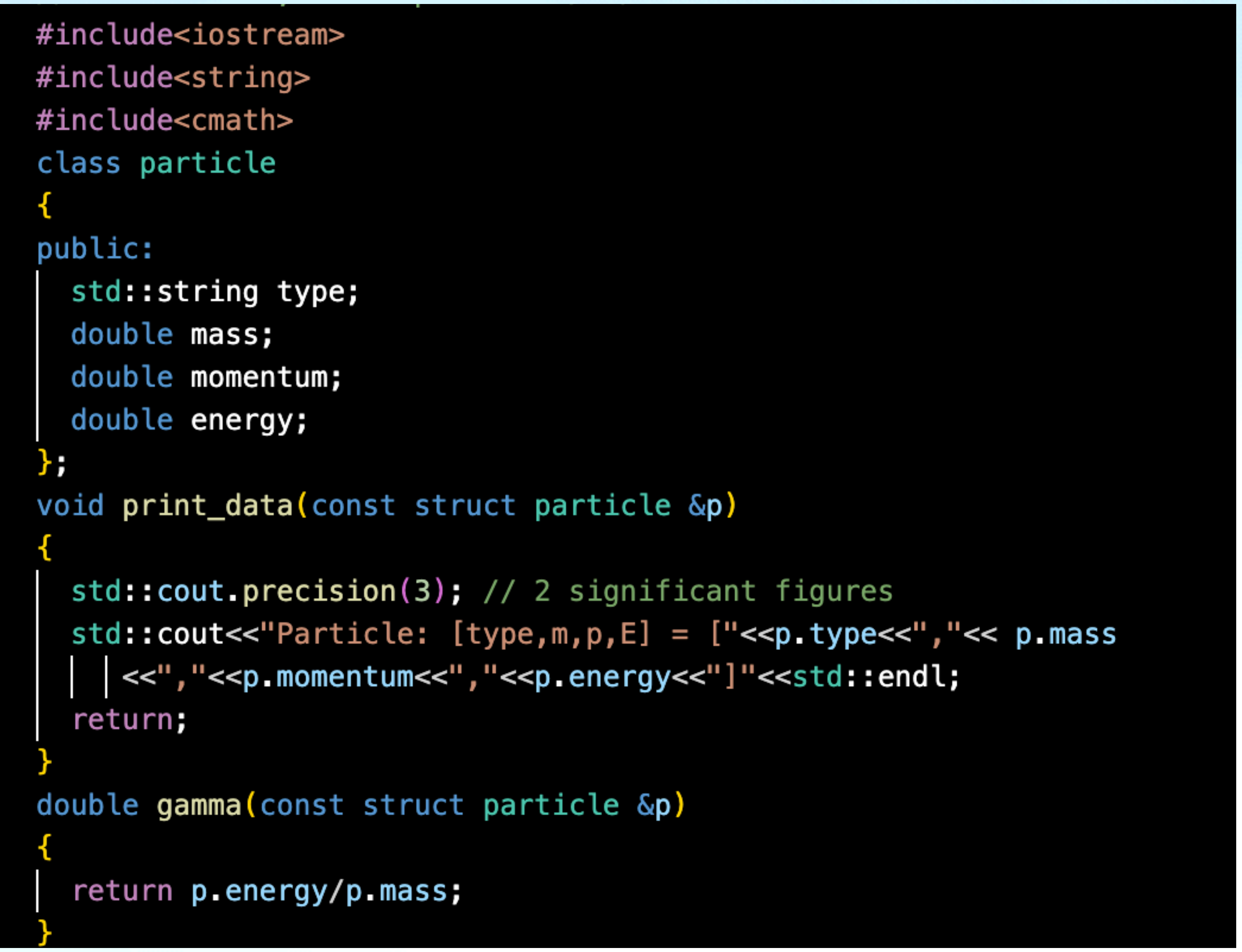
- Those functions can't be stored in the struct (even though they logically belong to it), they can only use the struct as input
- Another disadvantage: the struct declaration can't set up or keep default values for its data members
- Enter **C++ classes**, combining data and functions

Part 2: the class

Our first class

The particle

Note: the only reason this is a screenshot is ease of formatting
Please don't send anyone screenshots of code!



```
#include<iostream>
#include<string>
#include<cmath>
class particle
{
public:
  std::string type;
  double mass;
  double momentum;
  double energy;
};
void print_data(const struct particle &p)
{
  std::cout.precision(3); // 2 significant figures
  std::cout<<"Particle: [type,m,p,E] = ["<<p.type<<","<< p.mass
      <<","<<p.momentum<<","<<p.energy<<"]"<<std::endl;
  return;
}
double gamma(const struct particle &p)
{
  return p.energy/p.mass;
}
```

Find this code on GitHub at: Prelecture4/class1.cpp

- This code works exactly as the previous struct
- Advantages:
  - we can decide whether the data in the class is accessible and modifiable by the outside world (*public*) or not (*private*)
    - a struct is a class where all data members are public
  - we can also add functions to this class

Public/private (for protected, see later lectures)

The principle of least privilege (or: C++'s GDPR)

- Public: data members are accessible from "the outside" (e.g. main function)

```
class particle
{
public:
std::string type;
double mass;
double momentum;
double energy;
};
```

Find code that does not compile at: [Prelecture4/class2.cpp]() and modify it so that it does!

Public/private (for protected, see later lectures)

The principle of least privilege (or: C++'s GDPR)

- Private: cannot access data members from “the outside”

```
class particle
{
private:
std::string type;
double mass;
double momentum;
double energy;
//How to access data members: ‘accessor’ functions
//These must be inside the class!
public:
// Function to set type of particle
void set_type(const string &ptype) {type=ptype;}
// Function to print type of particle
void print_type() {cout<<"Particle is of type “<<type<< endl;}
};
```

Find code that does not compile at: [Prelecture4/class2.cpp]() and modify it so that it does!

Public/private (for protected, see later lectures)

The principle of least privilege (or: C++'s GDPR)

- Public: can access all data members from "the outside"

```
class particle
{
public:

std::string type;
double mass;
double momentum;
double energy;

};
```

- Private: cannot access data members from "the outside"

```
class particle
{
private:

std::string type;
double mass;
double momentum;
double energy;

//How to access data members: 'accessor' functions
//These must be inside the class!

public:
// Function to set type of particle
void set_type(const string &ptype) {type=ptype;}
// Function to print type of particle
void print_type() {cout<<"Particle is of type "<<type<< endl;}

};
```

- **Principle of least privilege** - *elements of a class (data or functions) should be private unless proven to be needed to be public*
  - This also means that users should not need to rely on / look at the implementation of a class: use **interface** only
  - Advantage: you can change the internal behaviour of a class without affecting its users, as **they only use the public interface and functions**
  - See why in the last video for this lecture - Object Oriented Programming concepts

Access functions / accessors

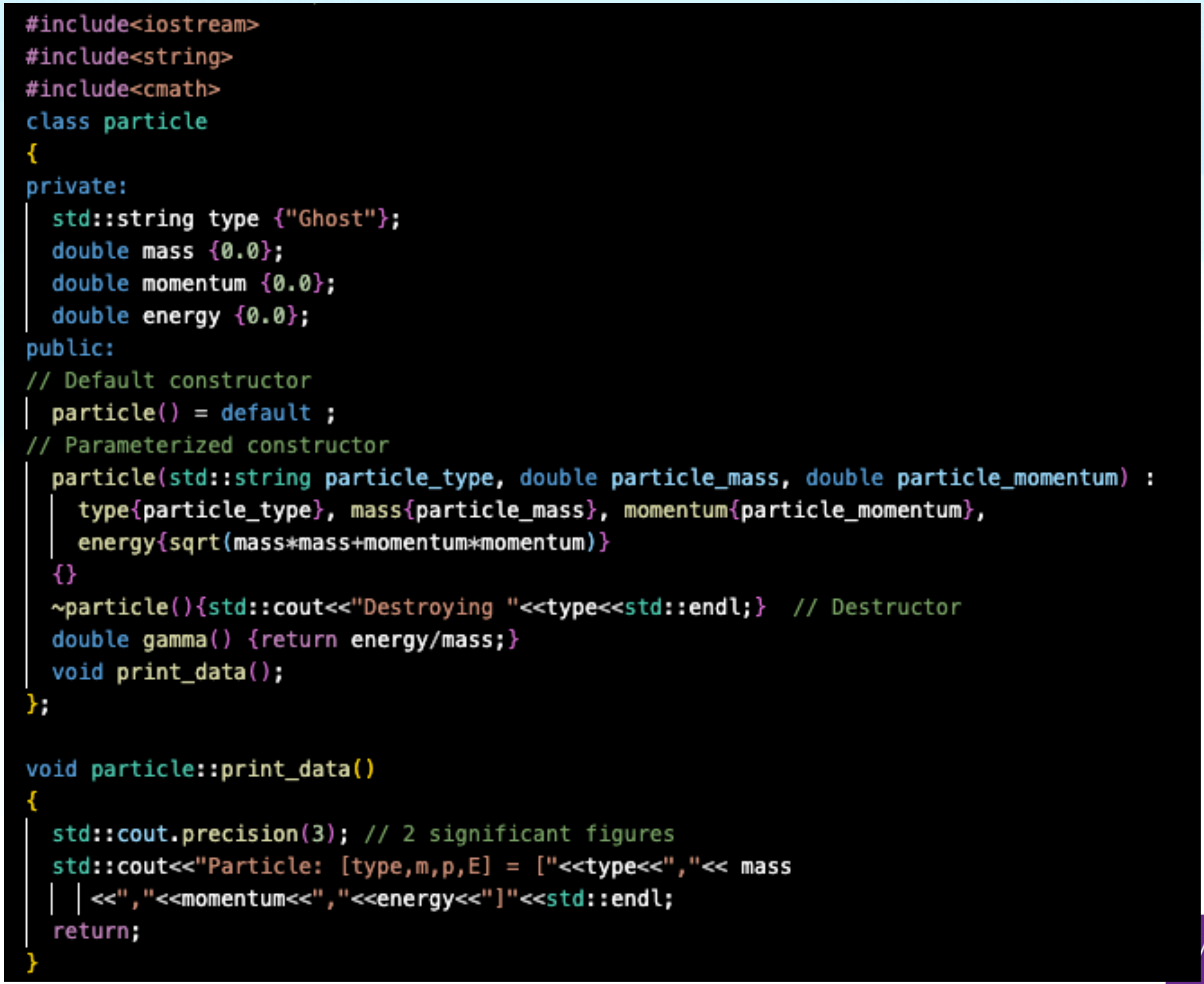
How to set/get private member data

- In the previous code, we have two **public functions**. This is because we wish to access these functions from outside the class.
  - When a new object is created, we use the functions to refer to that particular object.
  - We access these functions in a similar way to accessing the object's (public) data: `myObject.myFunction(myArgument)`; making clear that the function is associated with the object
  - Example for earlier code (this would go in the `main()` function):

    ```
    string type("electron"); particle p1;
    p1.set_type(type); p1.print_type();
    ```
- We only allow access to the data through access functions/accessors. We can protect our data from any undesirable consequences in designing these functions (see more in last video).

Our particle class

Now with member functions



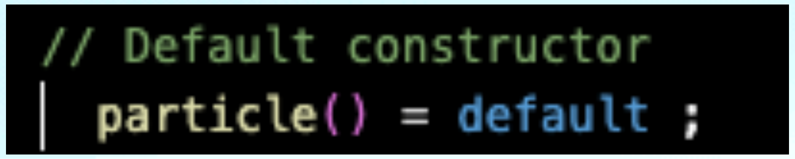
```cpp
#include<iostream>
#include<string>
#include<cmath>
class particle
{
private:
  std::string type {"Ghost"};
  double mass {0.0};
  double momentum {0.0};
  double energy {0.0};
public:
// Default constructor
  particle() = default ;
// Parameterized constructor
  particle(std::string particle_type, double particle_mass, double particle_momentum) :
    type{particle_type}, mass{particle_mass}, momentum{particle_momentum},
    energy{sqrt(mass*mass+momentum*momentum)}
  {}
  ~particle(){std::cout<<"Destroying "<<type<<std::endl;}  // Destructor
  double gamma() {return energy/mass;}
  void print_data();
};

void particle::print_data()
{
  std::cout.precision(3); // 2 significant figures
  std::cout<<"Particle: [type,m,p,E] = ["<<type<<","<< mass
      <<","<<momentum<<","<<energy<<"]"<<std::endl;
  return;
}
```

Find this code at: Prelecture4/class4.cpp

Constructors and destructors

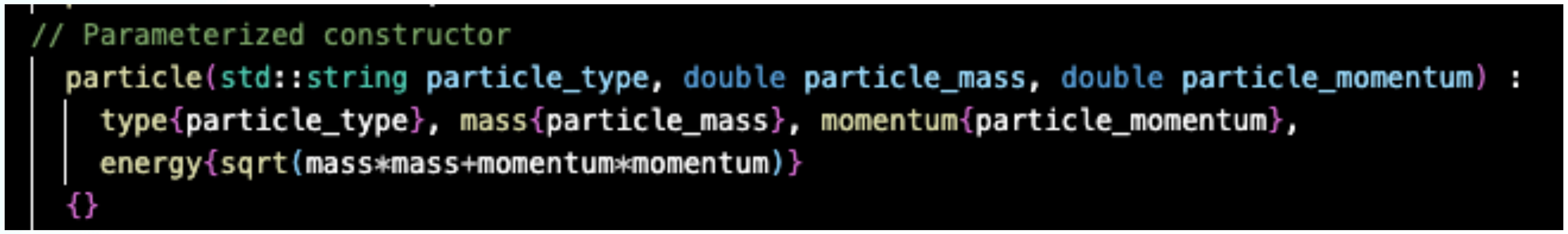
Special member functions

- Constructors are required in C++ classes to create new objects
  - they must have the **same name as the class**
  - they also **initialise data members**
  - there is a **default** constructor (can "do nothing, leaving things as defaults)…



```
// Default constructor
  particle() = default ;
```

  - as well as **constructors that can take arguments**, same as any other function in C++
    - the syntax using : and { } assigns data members (including calculations!)



```
// Parameterized constructor
  particle(std::string particle_type, double particle_mass, double particle_momentum) :
    type{particle_type}, mass{particle_mass}, momentum{particle_momentum},
    energy{sqrt(mass*mass+momentum*momentum)}
  {}
```

    - this is done using overloading - to be seen in later lectures

Find this code at: Prelecture4/class4.cpp

Constructors and destructors

Special member functions

- Constructors are required in C++ classes to create new objects
  - they must have the same name as the class
  - they also initialise data members
  - there is a default constructor (can “do nothing, leaving things as defaults)…

```
// Default constructor
  particle() = default ;
```

  - as well as constructors that can take arguments, same as any other function in C++
    - the syntax using : and { } assigns data members (including calculations!)

```
// Parameterized constructor
  particle(std::string particle_type, double particle_mass, double particle_momentum) :
    type{particle_type}, mass{particle_mass}, momentum{particle_momentum},
    energy{sqrt(mass*mass+momentum*momentum)}
  {}
```

    - this is done using overloading - to be seen in later lectures
- Destructors ‘destroy’ the class object

```
~particle(){std::cout<<"Destroying "<<type<<std::endl;}  // Destructor
```

  - They are called when the object of that class goes out of scope, or they can be called by the user
  - Implementation particularly important if dynamical allocation of memory in the class, otherwise memory leak

Find this code at: [Prelecture4/class4.cpp]()

MANCHESTER 1824 The University of Manchester

Functions in classes

And function prototypes inside the class

- So far, all functions were defined within the class itself (e.g. constructors), but we have not specified the details for `print_data`!
- Such a larger member function, included in full detail, can make the code look clumsy
- Solution: put implementation of such member functions outside of the class (or even in a separate file…see bonus content)
- Important note: member functions must be prototyped inside the class
- Example: define a function to print an object's data.
  - We first declare its existence inside class using function prototype
  - And define what it does outside the class, remembering the scope resolution operator ::
    - Common compiler error if that is forgotten: function cannot access data members

Summary of classes so far

“Buzzword summary”

- A **class** is the set of rules used to **define our C++ objects**. It specifies which types of **data** and **functions** are created and their **scope** (private or public)
- An **object** is an **instance of a class**. Each object will have specified its own set of data (values of the data members).
- A **member** refers to either **data** or a **function** belonging to a particular class, e.g. a constructor will be a member function. Member functions are sometimes called **methods**
- A **constructor** is a special function called when a class is instantiated, usually to **initialise** an object’s member data. If not user generated, generated by compiler.
- A **destructor** is the function called when an object is **destroyed** (usually automatically when exiting a function; we say “the object goes **out of scope**”– this happens when we can no longer access the object). If not user generated, generated by compiler.

Part 3: refinements to our class

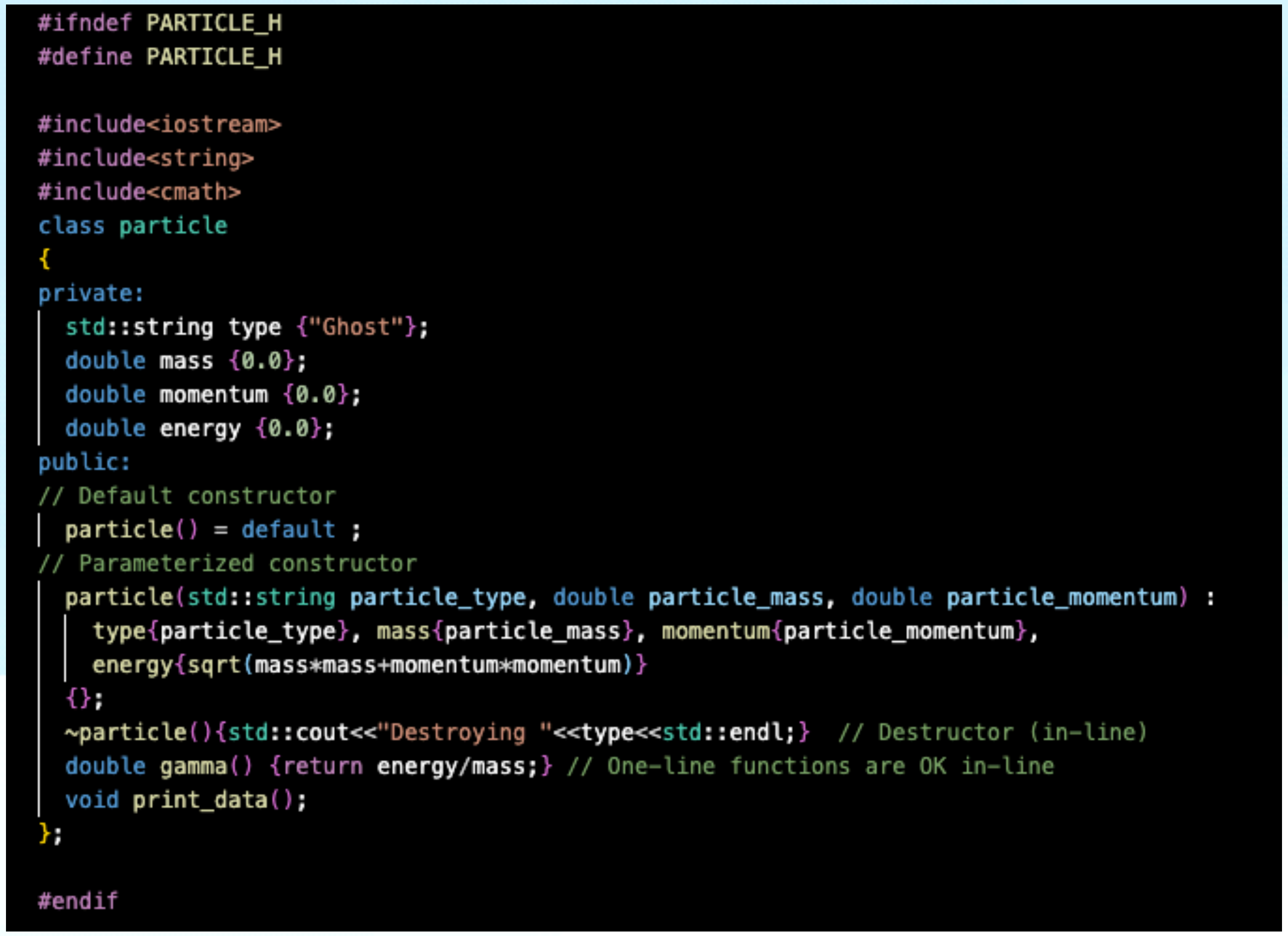
Headers and implementation

Why we do this: more in video #3

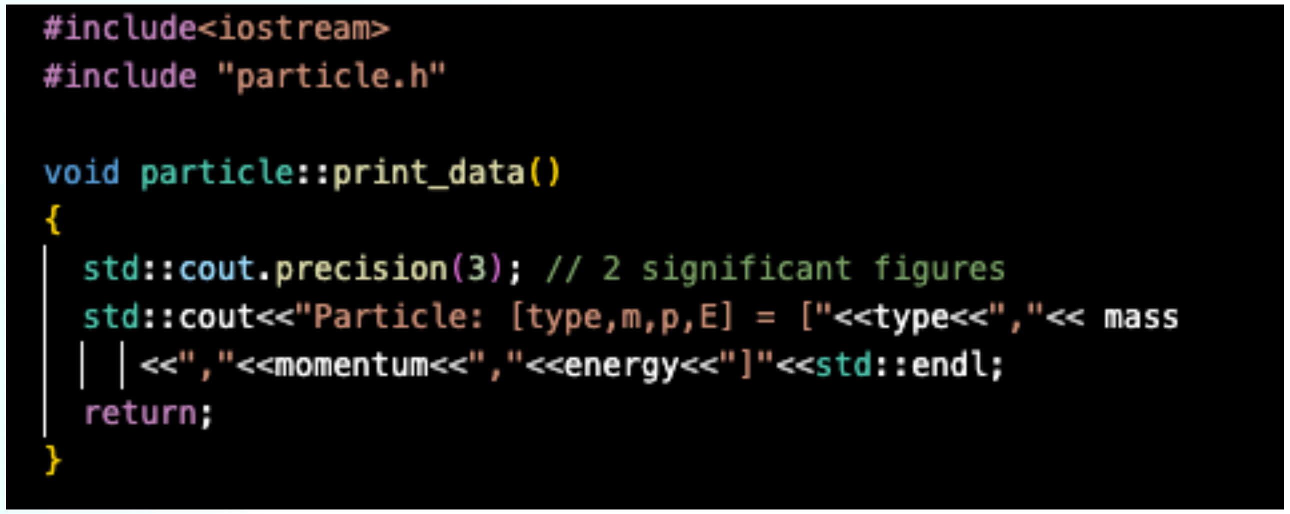
- C++ classes are usually split into two different parts, often in different files
  - *Header* (file extension: .h, .hpp) = where things are defined
    - Think of it as the index of a book
    - Usually headers contain *interfaces* that help you get an overview of what something (e.g a function) does, without the clutter of the full implementation
  - *Implementation* (file extension: .cxx, .cpp) = where things are implemented
    - Think of it as the book content
    - The implementation of the class's functions go here

Our particle class, split in .cxx and .h

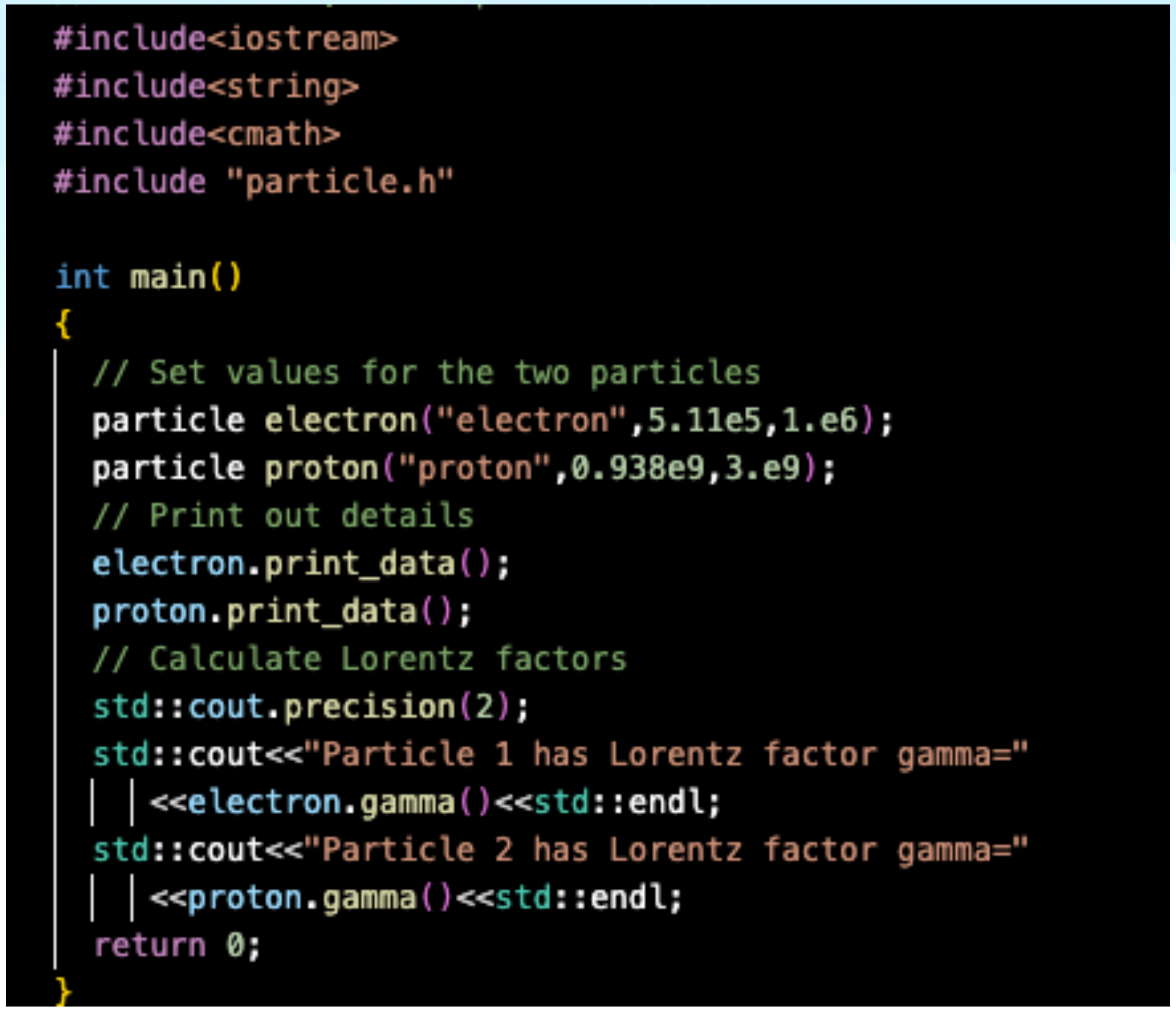
Compilation will have to change slightly…



```cpp
#ifndef PARTICLE_H
#define PARTICLE_H

#include<iostream>
#include<string>
#include<cmath>
class particle
{
private:
  std::string type {"Ghost"};
  double mass {0.0};
  double momentum {0.0};
  double energy {0.0};
public:
// Default constructor
  particle() = default ;
// Parameterized constructor
  particle(std::string particle_type, double particle_mass, double particle_momentum) :
    type{particle_type}, mass{particle_mass}, momentum{particle_momentum},
    energy{sqrt(mass*mass+momentum*momentum)}
  {};
  ~particle(){std::cout<<"Destroying "<<type<<std::endl;}  // Destructor (in-line)
  double gamma() {return energy/mass;} // One-line functions are OK in-line
  void print_data();
};

#endif
```



```cpp
#include<iostream>
#include "particle.h"

void particle::print_data()
{
  std::cout.precision(3); // 2 significant figures
  std::cout<<"Particle: [type,m,p,E] = ["<<type<<","<< mass
    <<","<<momentum<<","<<energy<<"]"<<std::endl;
  return;
}
```



```cpp
#include<iostream>
#include<string>
#include<cmath>
#include "particle.h"

int main()
{
  // Set values for the two particles
  particle electron("electron",5.11e5,1.e6);
  particle proton("proton",0.938e9,3.e9);
  // Print out details
  electron.print_data();
  proton.print_data();
  // Calculate Lorentz factors
  std::cout.precision(2);
  std::cout<<"Particle 1 has Lorentz factor gamma="
    <<electron.gamma()<<std::endl;
  std::cout<<"Particle 2 has Lorentz factor gamma="
    <<proton.gamma()<<std::endl;
  return 0;
}
```

Find these codes at:
[Prelecture4/particle.h]()
[Prelecture4/particle.cpp]()
[Prelecture4/main_class4.cpp]()

MANCHESTER 1824
The University of Manchester

How do we compile this?

Putting <u>everything in one line</u> makes sure all .cpp are compiled, the .h are included automatically via *preprocessor directives*

Careful in copying/pasting this, it should be a 'minus sign' (-) not a dash (—)

```
[yourCompilerDir]/g++-11 -std=C++17
```

make sure C++17 standard is selected

```
-fdiagnostics-color=always
```

Add some color to your life

```
-g [yourdir]/main_class4.cpp
   [yourdir]/particle.cpp
```

The -g option tells the compiler that those are your source files

```
-o [yourdir]/main_class4
```

The -o option tells the compiler how to call the executable
This is the program you should run

Note: VS Code's "Build / Run" will have to be modified to do this, but for now you can start by exercising the command line more!

MANCHESTER 1824 The University of Manchester

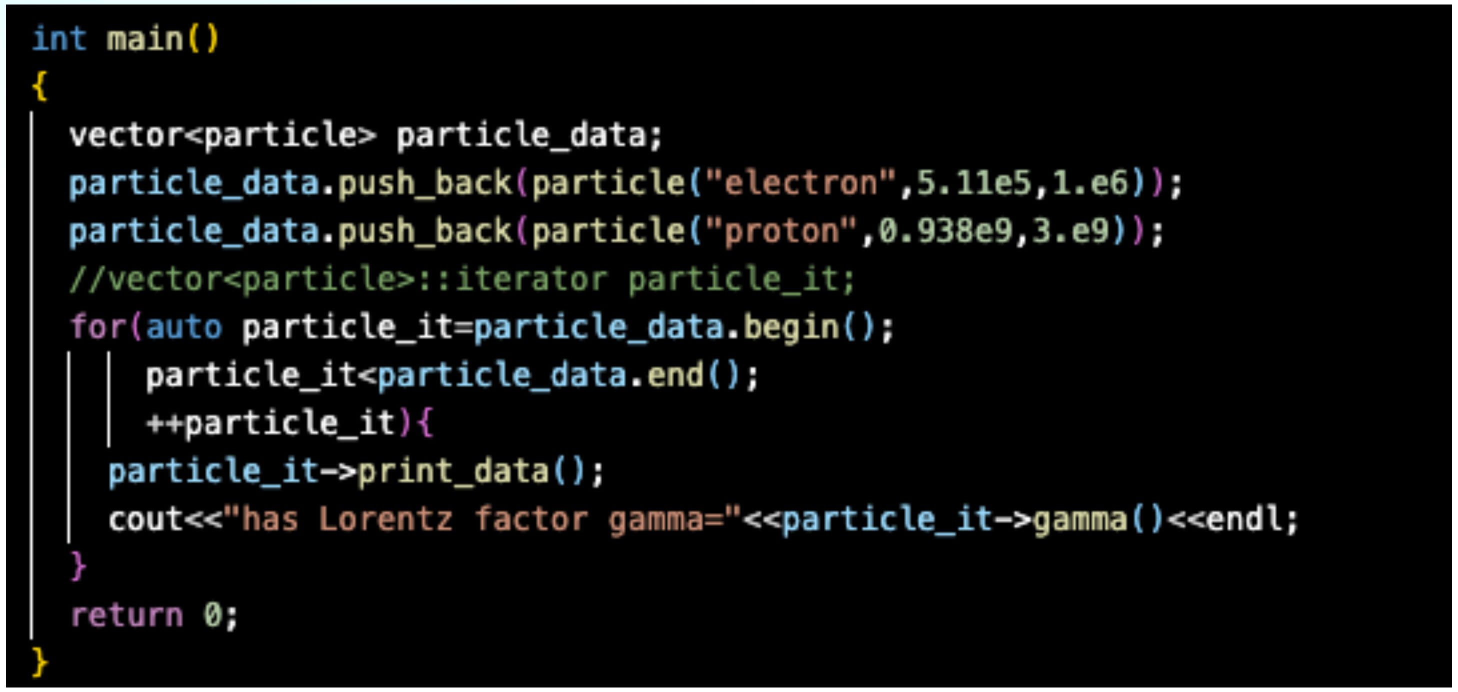
Refinement of classes

Return values for functions, using vectors

- If we have a large number of particle objects, we can store them into a vector
    - Then we can iterate on the vector using iterators

both lines work!

auto pointers:
we are keeping the best
for lecture 8…
which will be released
before the Easter teaching break



```
int main()
{
  vector<particle> particle_data;
  particle_data.push_back(particle("electron",5.11e5,1.e6));
  particle_data.push_back(particle("proton",0.938e9,3.e9));
  //vector<particle>::iterator particle_it;
  for(auto particle_it=particle_data.begin();
      particle_it<particle_data.end();
      ++particle_it){
    particle_it->print_data();
    cout<<"has Lorentz factor gamma="<<particle_it->gamma()<<endl;
  }
  return 0;
}
```

Find this code at: Prelecture4/class5.cpp

Refinement of classes

Return values for functions, using vectors

- This code used a few refinements. If we have a large number of particles, it is much easier to use a vector to contain all of them
- We can then use iterators over the data to output all the information
- Here we use the arrow -> operator to get a class member of a dereferenced pointer `particle_it->print_data();`
    - particle_it->print_data() is the same as (*particle_it).print_data(), but easier to read.
- Remember, the **iterator** particle_it is like a pointer!

Find this code at: Prelecture4/class5.cpp

Part 4: first elements of Object Oriented Programming (OOP)

What is Object-Oriented Programming?

Most describe it in terms of these 4 principles:

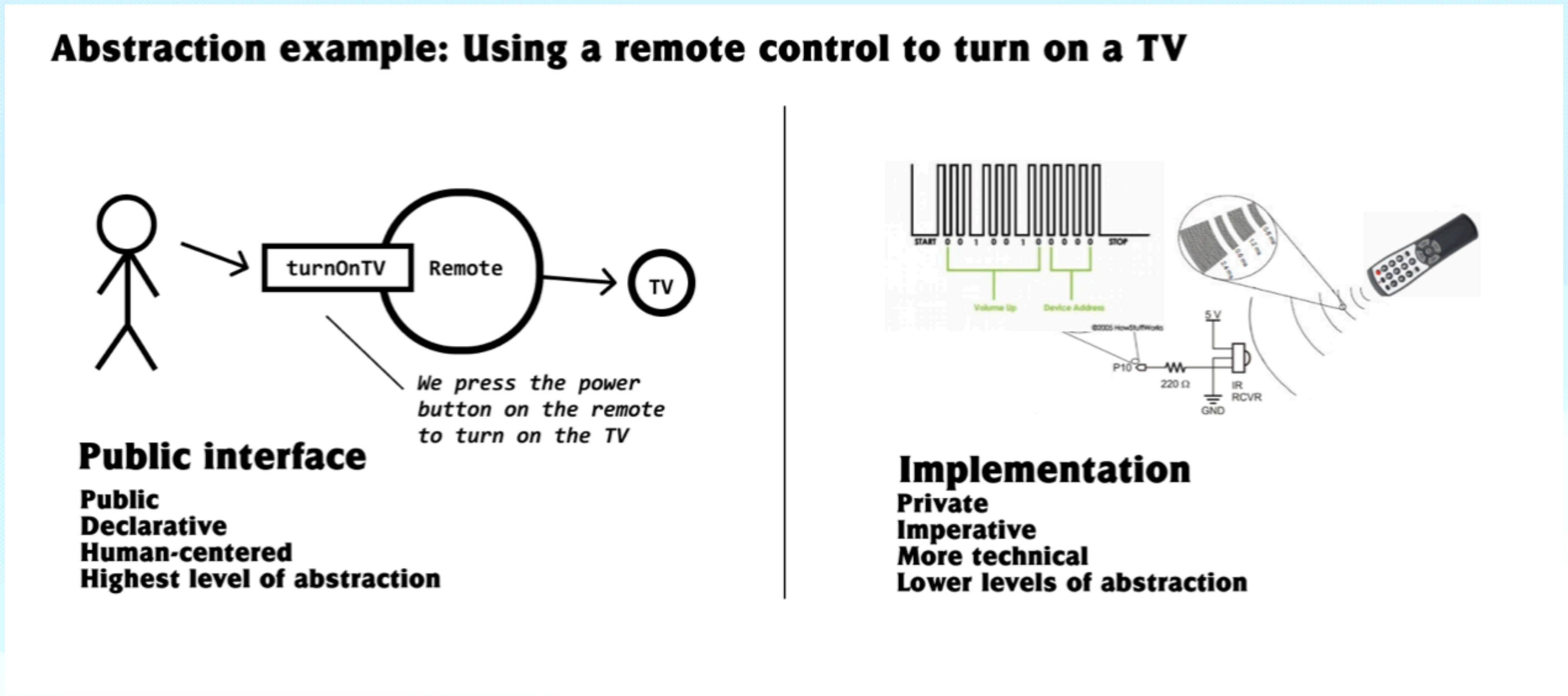
Talking about these two today

- Abstraction
  separate interface and implementation (see: *interfaces*)
- Encapsulation:
  keep data private, alter properties via methods only
- Inheritance
  classes can be based on other classes to avoid code duplication
- Polymorphism
  can decide at run-time what methods to invoke for a certain class, based on the object itself

MANCHESTER 1824
The University of Manchester

Abstraction

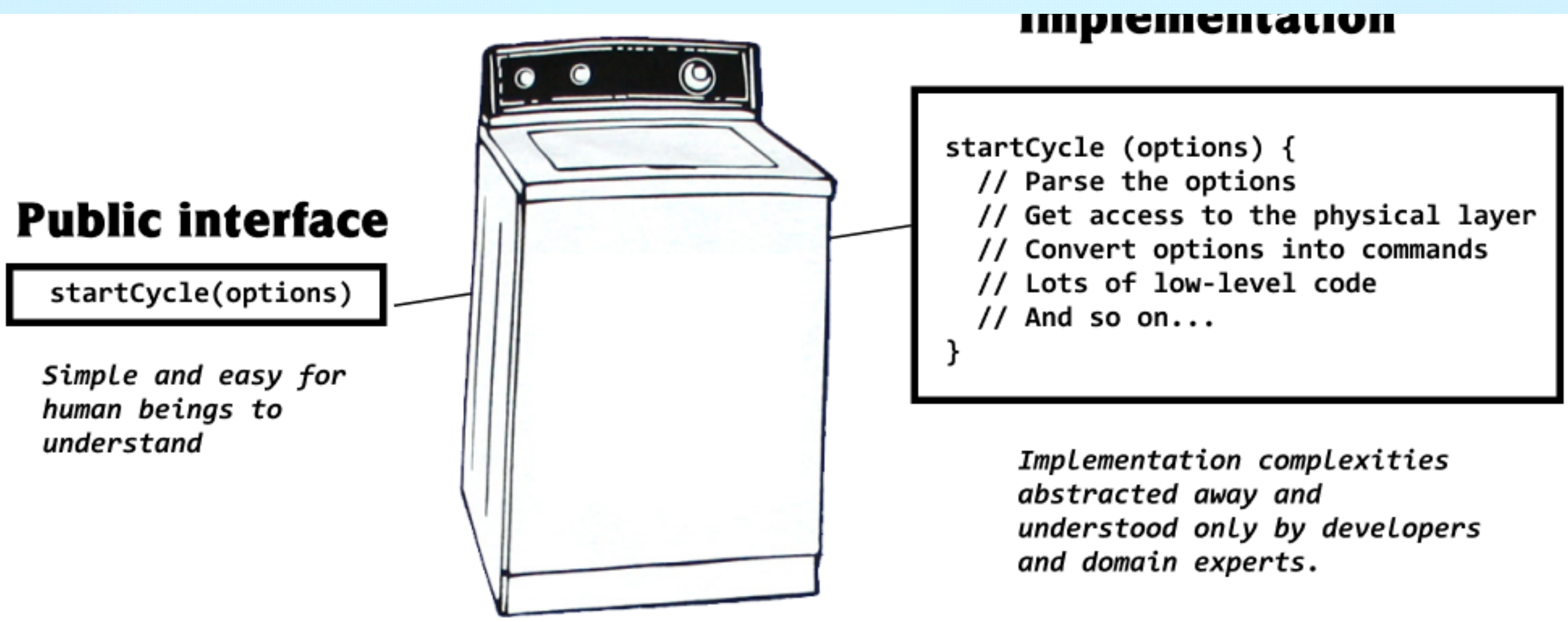
Images from this website:


Abstraction example: Using a remote control to turn on a TV
turnOnTV
Remote
TV
We press the power button on the remote to turn on the TV
Public interface
Public
Declarative
Human-centered
Highest level of abstraction
Volume Up
Device Address
5 V
P10
220 Ω
IR RCVR
GND
Implementation
Private
Imperative
More technical
Lower levels of abstraction

- interface and implementation are separate
  - the user doesn't need to know how the remote control works to turn on the TV
  - the interface should be sufficient for the user to understand how to turn on the TV
- interface contains method declaration, implementation contains everything that is needed for its execution

Another example of abstraction

Images from this website:



```
startCycle(options)
```

Simple and easy for human beings to understand

```
startCycle (options) {
  // Parse the options
  // Get access to the physical layer
  // Convert options into commands
  // Lots of low-level code
  // And so on...
}
```

Implementation complexities abstracted away and understood only by developers and domain experts.

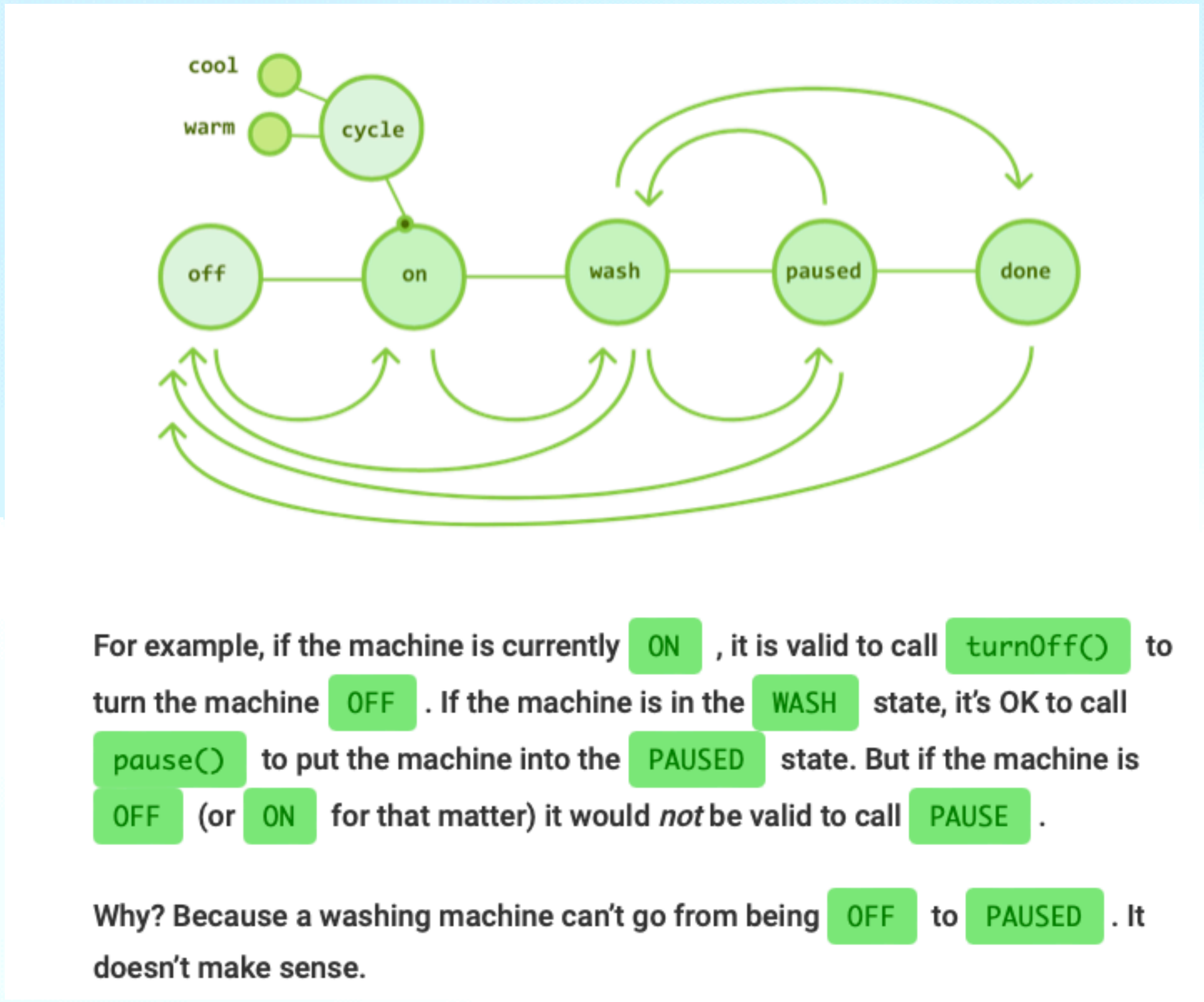
- Paul Graham (tech entepreneur) has suggested that OOP's popularity within large companies is due to "large (and frequently changing) groups of mediocre programmers." According to Graham, the discipline imposed by OOP prevents any one programmer from "doing too much damage." I just see this as a very negative take on good practice!

Software Sustainability Institute

MANCHESTER 1824 The University of Manchester

Encapsulation

Images from this website:

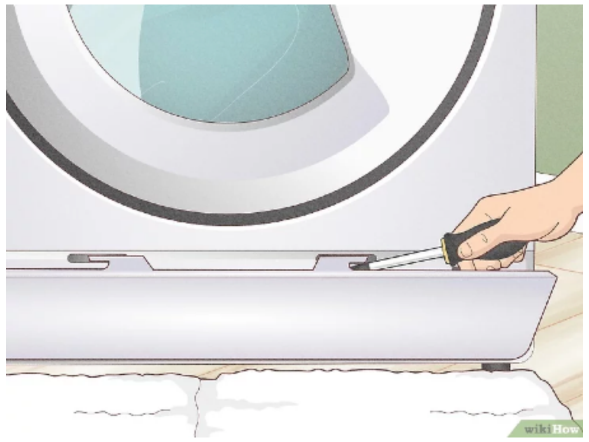


For example, if the machine is currently `ON`, it is valid to call `turnOff()` to turn the machine `OFF`. If the machine is in the `WASH` state, it's OK to call `pause()` to put the machine into the `PAUSED` state. But if the machine is `OFF` (or `ON` for that matter) it would *not* be valid to call `PAUSE`.

Why? Because a washing machine can't go from being `OFF` to `PAUSED`. It doesn't make sense.

similarly, opening the door of the washing machine while it is doing the washing cycle is a bad idea

- data members are private, use accessor/mutator (or: getter/setter) functions to modify them
  - the user isn't allowed to change the state (data member) of the washing machine
  - there are functions that do that, and they check that everything makes sense



users can be full of bad ideas
source: wikihow

Encapsulation in practice

- accessors and mutator (setter/getter) functions provide a **direct way to change** or just **access** private variables
- They must be written with the utmost care
  - …after all these are providing the protection for the data in the first place!
  - *So, add checks in member functions to make sure your washing machine doesn't go in a funny state*
- Remember what we said about a class:
  - data members are hidden in the <u>private</u> section
  - they can only be modified by public member functions (which dictate allowed changes)