C++ - Assignment 5

Practicing inheritance, smart pointers and friendship

Marking scheme is 15 points total as this is a more challenging assignment

Charanjit Kaur & Caterina Doglioni, PHYS30762 - OOP in C++ 2023

MANCHESTER 1824
The University of Manchester

Part 1. Refine your particle class

- Instead of having your four-momentum as a vector data member in the particle class from assignment 4:
  - change it into a **four-momentum class**
    - the implementation of this class is up to you (you can keep a stl::vector as before), but you need setters and getters for each of the quantities
    - No need to do any physical values checks on the 4-momentum, unless you want to practice some more 1st year relativity - **just check that E>0**
    - **You will also need user-defined assignment and move operators, copy and move constructor and destructor at least for four-vector**
      - **In general, follow good coding practices from the pre-lectures even if the rubric/ slides don't ask you - in real life coding / in a job interview there is no rubric!**
  - have a **smart pointer** to a four-momentum object contained in each particle
    - Think about the following: *what smart pointer will you need? unique or shared?*
- *Marking scheme: 1 marks for four-momentum class, 1 marks for using smart pointer in particle class correctly (only 0.5 marks if use is not correct or functionalities/setters/getters missing)*

Class names clarification

- In our house style, we ask for **variable names** to be explicit in what they represent, using all small letters with underscores _ in between (*snake_case*)

- For class names, we don't have any house style specification

  - One suggestion is to use *CamelCase* (or *Pascal case*) https://en.wikipedia.org/wiki/Camel_case so that you can distinguish between a class name and a variable name

    - For file names, the class and the file can have the same name, including capitalisation

- We also approve of the style that calls the data members starting with `m_` (but again, it's a choice)

- See for example the Google C++ style guide: https://google.github.io/styleguide/cppguide.html

Part 2a. Build an inheritance chain for your leptons

- Turn your particle class into a base `lepton` class from which electrons and muons (derived classes) inherit
  - At this point, the “particle name” data member becomes redundant (unless you want to give first names to your particles, which you can do if you feel particularly creative)
  - Think by yourself:
    - what are the common data members that are shared by all objects of the same “particle” kind?
      - You can also set them all to the same value in the base class if you think it makes sense to do so
    - how do you implement particles and antiparticles efficiently?
- make at least one base class data member **protected** so that you can edit it directly from the derived class

Part 2a. Build an inheritance chain for your leptons

- Derived class members will be specific detector-related properties (they also need setters/getters)
    - for electrons, you will need to implement a vector of **energies that the electron deposited in each calorimeter layer** (see https://atlas.cern/Discover/Detector/Calorimeter for what these are if you haven't seen them in a particle physics course yet)
        - assume that a calorimeter has 4 layers, called `EM_1`, `EM_2`, `HAD_1`, `HAD_2`
        - make sure that the total energy deposited in the calorimeter layers is equal to the energy of the electron in the 4-vector (use closest acceptable values if users sets something inconsistently)
    - for muons, you will need to have the following **isolation information:**
        - a bool that tells you if the muon is isolated from other particles
        - This is not meant to be physical (we are not simulating anything at the moment, it's just to exercise inheritance), so it should just be 0 or 1 independent of where the muon comes from
- *Marking scheme: 2 mark total for electron and muon with all the data members mentioned above (marks for protected comes later), -0.5 points if no / partial input checking*

Part 2b. Add muon and electron neutrinos

- Neutrinos are the light-lepton counterparts of electrons and muons
  - They have a *flavour* (muon neutrino, electron neutrino, and respective antineutrinos)
  - They are invisible to the detector as they don't interact much!
    - The only way you can see them in a particle detector like the ones we described in assignment 2 is via **missing transverse momentum** (we won't do anything about this, in this assignment)
  - In your assignment, add a derived class member to class hierarchy, inheriting from the generic lepton class
    - Derived class specific data member: `bool hasInteracted` marking if it has interacted with a detector, even if unlikely (to be set when you instantiate the particle as either 0 or 1, no rule here)
  - *Marking scheme: 1 mark total for neutrino class implementations including the `hasInteracted` data member*

Part 2c. Add tau leptons

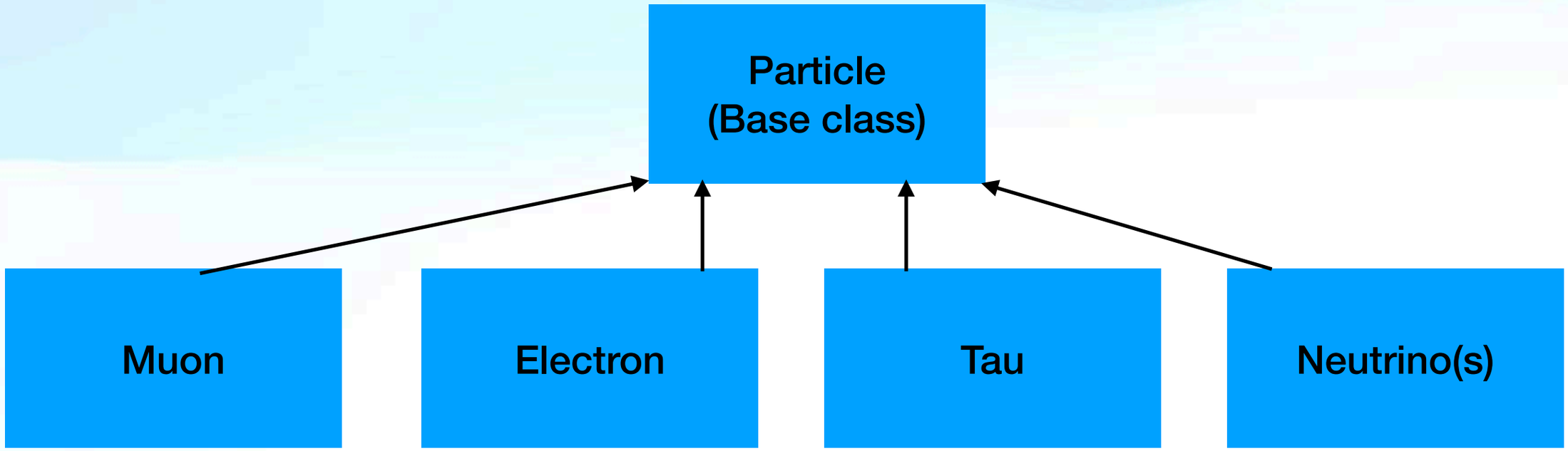
- Taus are the heaviest leptons
  - They are also unstable (so are muons, but we won't deal with muon decays in this assignment)
  - They can decay into:
    - hadrons and a tau neutrino
    - a lepton and two neutrinos (one tau, one of the same flavour of the lepton)
  - Add a derived class for taus, which should contain:
    - A flag for the kind of decay (hadronic/leptonic)
    - If the tau decays leptonically, you need smart pointers to the particles it decays into
      - Think about the following: *what smart pointer will you need here? unique or shared? How will you store these pointers?*
      - No need to do the same for the neutrino of the hadronic decay, just leave that as a flag
- You should also add a tau neutrino/antineutrino (same implementation as the other neutrinos)
- *Marking scheme: 2 marks total for tau class implementation, 1.5 if missing features*

Must-haves in your inheritance chain

- Two layers of hierarchy (lepton <- muon/tau/electron or neutrinos) *[1 mark]*
  - It is up to you to decide if neutrinos are different enough among themselves to warrant different classes for different flavours
- User-defined assignment operators, copy constructors and destructors at least for the class in the hierarchy that contains the four-vector *[1 mark]*
  - You can use the ones from assignment 4, even though with smart pointers you don't need to use "delete" anymore
- At least one protected data member in the base class *[0.5 marks]*
- Print function in base class and user-defined print classes for derived classes that together print all the characteristics of the particle in question *[0.5 marks]*
  - Suggestion: Call the base class function from the derived class one to avoid code duplication

Inheritance chain in picture

- We will see in lecture 8 (released this week or next Monday at the latest) how to make inheritance diagrams
- A simple one (as an example) for this assignment:



Part 3. Practice friend functions

- Write a friend function that is outside the particle class and it is its *friend* and does the following:
  - sums two four-vectors (that are hosted in a particle class)
  - takes the dot product of four-vectors (that are hosted in a particle class)
  - You can use the implementations from assignment 4, but you should be careful not to directly access members from the four-momentum from the particle class (otherwise you need more friend-ship…)
- ***Note**: in the spirit of coding and testing functionalities step by step, you can test these functions with four-vectors you instantiate outside the class first, to check that it all works, and then modify this to be a"friend" function to act on the four-vectors inside a particle*
  - *Marking scheme: 1 mark for friend functions, 0.5 if only one function or implementation problem*

Part 4a. Show how things work in main()

- Create the usual vector of particles, and add taus and neutrinos so you have:
  - two electrons
  - four muons
  - one antielectron
  - one antimuon
  - one muon neutrino
  - one electron neutrino
  - one tau decaying into a muon, a muon antineutrino and a tau neutrino
  - one antitau decaying into an antielectron, an electron neutrino and a tau antineutrino
- Call the print functions of all the particles in the vector in the loop
- *Marking scheme: no points, as this just makes sure that all the features you have implemented work correctly*

Part 4b. Practice friend functions and pointers

- *[0.5 marks]* Sum the four-vectors of the two electrons and print the result on screen
- *[0.5 marks]* Take the dot products of the antielectron and antimuon four-vector and print the result on screen
- *[1 challenge mark to get to 100%]:* Create a unique pointer for a new electron (outside the original vector) and then move its data to another electron using `std::move`
- *[1 challenge mark to get to 100%]:* Create a shared pointer for a tau lepton that is owned by multiple variables
  - Use case that may turn up in the project: this can happen when multiple detectors have a vector of particle pointers that passed through them

Marks for code compilation

- *Additional marks:*
  - *0.5 for use of git (commit or tag/release)*
  - *0.5 for splitting in interface and implementation*
- 1 mark will be deducted if your program produces any compilation warnings. **If your code does not compile, we will not debug/ mark it and you will get zero marks.** (This is the same as assignments 3 and 4, where in the slides we made suggestions on how to build your code up so that it does not happen)
- 1 mark will be deducted if your code does not follow the house style / for unclear code (variable/function names)

Link to join the GitHub repository:

https://classroom.github.com/a/tYbDl7r4

Note: for this assignment, we do not provide a template as we're getting close to the project where you won't have one.