PHYS30762: C++ Particle Catalogue

Lewis Dean - 10823674

School of Physics and Astronomy

University of Manchester

May 2024

Abstract

This report aims to convey the effective use of C++ and its object-orientated paradigm when implementing a catalogue of elementary particles, each being an instance of their respective classes[1]. Their specific properties are encoded within their data members, whilst deriving common properties from an abstract particle base class. The catalogue class wraps a std::multimap, allowing numerous particle objects to be associated with user-defined keys, and it features methods for neatly printing the properties of any stored particles. The report closes with the resulting user experience, and a discussion of possible extensions.

[1]Hyperlink to GitHub release

Contents

1 Introduction

The Standard Model is the pinnacle of modern physics, enabling all known elementary particles to manifest as excitations of their underlying quantum fields. The model is divided according to particle spin; those with spins that are integer multiples of $\hbar$ are labelled bosons, whilst those with half-integer multiples are labelled fermions. The vector bosons, i.e. the non-zero spin bosons, function as the mediators of the 3 fundamental interactions: the electromagnetic, strong and weak interactions. The only scalar boson, the Higgs boson, belong to the Higgs field, with the latter being responsible for the elementary particle masses. The fermions are further split into the quarks and leptons, where quarks are characterised as fermions with colour charge.

Note that, when considering particle states, an accurate, relativistic, treatment requires working with their 4-momenta. Thus, a class for such objects is shown within Section 3.1. Using natural units, the 4-momentum vector of a Cartesian coordinate system has components $P^\mu = (E, P_x, P_y, P_z)$, where E denotes energy, and P_i is the i^{th} component of linear momentum. Lorentz scalars can be constructed from 4-vectors via the Minkowski product,

$$A^\mu B_\mu = A_0 B_0 - \mathbf{A} \cdot \mathbf{B}\,, \tag{1}$$

where the summation convention has been used, and the negative term is a regular dot product. Vitally, all real particles have a rest mass equal to their pole mass, and hence, by evaluating the Minkowski product of a particle's four momentum with itself within its rest frame, we conclude,

$$m^2 = E^2 - P^2\,. \tag{2}$$

This is coined the particle's invariant mass, and it restricts the available phase space such that certain decay modes could be prohibited. Additionally, the conservation of certain quantum numbers by a given fundamental interaction can also forbid decays. The implementation of particle classes must respect the theoretical nuances above if to be of any use. Hence the associated user-input validation is explained throughout Section 3.

The relationships realised between the many particle classes intend to make the user's experience as intuitive as possible, the details of which are in Section 5. Example users of this code could be those interested in a reliable framework to build Monte-Carlo simulations of particles upon; these could then be analysed in conjunction with experimental data to test the fit of theoretical models [3].

2 Universal coding practices

Some general design choices are made consistently throughout all classes. One such example is the use of constant references to variables within function arguments so as to prevent unnecessary copying. Moreover, all classes bar Photon have both split interface/implementation and complete, custom rule-of-fives. Appropriate header guards protect these against repeating definitions during compilation. Every file includes common.h, a header file with all other include directives and using-declarations required. To limit memory allocated, modifiable data members are assigned as floats where appropriate, with doubles being reserved for user-inputted decimal numbers. Similarly, static containers are occasionally implemented as C style arrays to reduce overhead. The try/throw/catch structure is used during validation of user inputs, with the explicit use of the macro EXIT_FAILURE for improved readability. Finally, any pointers are implemented as smart pointers since practicing RAII prevents memory leaks.

3 Particle zoo implementation

Figure 1 is a UML diagram for illustrating the relationships between classes. The Standard Model occupies a 3-level inheritance hierarchy here.

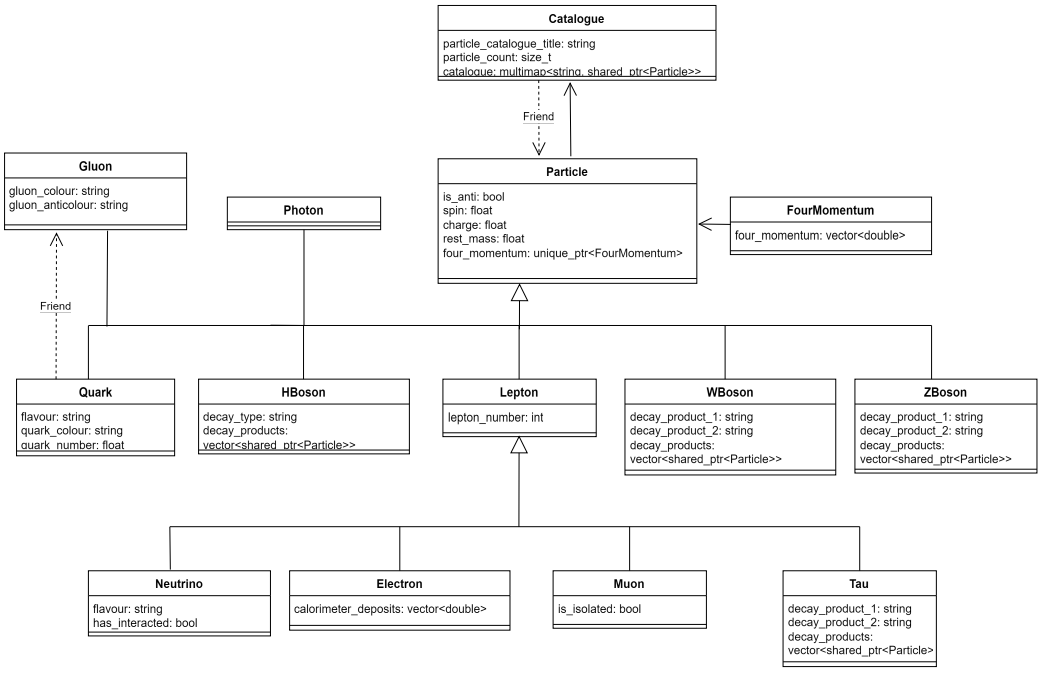

Figure 1: UML diagram summarising class inheritance/dependencies.

3.1 FourMomentum class

All particles objects are associated with an instance of the FourMomentum class; such instances feature a private std::vector containing the particle's 4-momentum components. Wrapping this vector, in the spirit of encapsulation, allows for proper validation of its elements. Instances can be built with either a default or parameterised constructor. Then, these components are accessed via getters and the purpose-built setters guarantee any modifications are physically valid. For example, assigned energies cannot be negative. The friend functions included are the overloaded +/- operators, for summing and subtracting the 4-vectors, and "dotProduct" which calculates the Minkowski product of two inputted FourMomentum objects. Extra methods include "get_invariant_mass()" and "get_3-momentum_magnitude()".

3.2 Particle class

The peak of the particle zoo hierarchy is an abstract base class labelled Particle, from which all standard model particles publicly inherit. The declared protected data members are described in Table 1. This class has no default constructor, but a parameterised constructor that is used in derived classes. Note, an energy cannot be inputted to this constructor; instead, 3 momentum components are, and the particle's rest mass is then used to initialise that particle on-shell. The class' data member four_momentum is a unique smart pointer to a FourMomentum object. Setters associated with 4-momentum components have additional validation, such as the code exiting if the inputted energy is below the particle's rest mass energy. Furthermore, when setting a given component, the other 3 components are scaled to maintain a magnitude matching the

invariant mass. For setting momentum components, a template shown in listing 1 is employed to adjust the particle's energy accordingly.

Listing 1: Template for setting momentum components.

```
template<typename setter> void Particle::set_i_momentum(
      const double& i_momentum_input, setter&& setter_input)
{
    (four_momentum.get()->*setter_input)(i_momentum_input);
    if(four_momentum->get_invariant_mass() != rest_mass)
    {
      double required_energy = std::sqrt(pow(rest_mass, 2) +
            pow(four_momentum->get_3_momentum_magnitude(), 2));
      four_momentum->set_energy(required_energy);
    }
}
```

Since this class is polymorphic, it requires a virtual destructor, but it also declares a virtual function for printing a particle's data. The latter is then overridden in the derived classes to print their additional data members too. The print function has a boolean argument, with a default value of false, which determines if a full printout of all decay product data is outputted.

Data Member	Data Type	Description
`is_anti`	boolean	Stores if particle/antiparticle
`spin`	float	Stores particle's spin
`charge`	float	Stores particle's electric charge
`rest_mass`	float	Stores particle's pole mass
`four_momentum`	unique_ptr<FourMomentum>	Points to particle's four momentum

Table 1: The protected data members of the Particle base class.

3.3 Quark class

This is derived from Particle and allows all 6 quark flavours (and their antiquarks) to be instantiated. Its extra private data members are the strings colour and flavour, as well as the float quark number. These all have getters, and colour has a setter. The default constructor creates a red up quark. The parameterised constructor accepts a flavour and colour from the user; these are converted to lowercase by a std::transform lambda function and subsequently searched for within static, constant, unordered sets of valid input strings. Note that Quark is a friend class of Gluon, as this contains the methods for validating inputted colours. These strings then serve as the keys of maps that hold the quark masses/charges, such that these fields can be initialised

correctly. The accepted values of said masses/charges were found at the Particle Data Group website [2]. Within the constructor, the ternary operator provides a succinct if-statement for flipping the charge when is_anti is set true. The spin is initialised as $\frac{1}{2}$, as well as for all other fermions, by definition.

3.4 Lepton classes

Lepton is a class derived from Particle yet is also abstract. It includes an additional int data member called lepton_number, which is +1 for leptons and -1 for anti-leptons. From this, four classes are derived that are described below.

3.4.1 Neutrino

This class allows all 3 flavours of neutrino (and their antiparticles) to be constructed. They are distinguished by the is_anti field from Particle, but also a neutrino-specific string data member called flavour. The user-inputted flavours are validated in a manner analogous to that detailed in Section 3.3. In addition, the boolean data member has_interacted is introduced, alongside associated setter/getters. It represents whether the neutrino object is known to have interacted or not.

3.4.2 Electron

This class builds upon Lepton with a 4 element std::vector of doubles called calorimeter_deposits. Its purpose is storing the amount of energy transferred to each of four different calorimeter layers as the electron passes through a detector. The first 2 layers, EM_1 and EM_2, detect electromagnetic particles, whereas the last 2 layers, HAD_1 and HAD_2, detect incident hadrons. The energy deposited to each layer is inputted to the Electron parameterised constructor, however they are validated to ensure they total to the energy stored in its associated FourMomentum object (within a 10^{-3} tolerance). When adjustment is necessary, if zero is inputted to all layers, the electron's energy is distributed equally, otherwise the momentum components are scaled, maintaining the proportion in each spatial component.

3.4.3 Muon

This is a simple extension to the lepton parent class that adds one boolean data member: is_isolated. This stores whether a muon is deemed isolated or not. The data member has both a setter and getter defined.

3.4.4 Tau

The Tau class extends Lepton by adding a private std::vector called decay_products for storing shared pointers to Particle objects. These represent the particles this tau would decay to; note that various conservation laws restrict the set of possible decays, and thus inputted decay

product strings are thoroughly validated before the code is allowed to continue. These strings are stored in Tau data members called decay_product_1 and decay_product_2. The valid decay modes are looped through and if the input strings match one, that index is passed to "update_tau_decay_products(const size_t& decay_index)" or "update_antitau_decay_products(const size_t& decay_index)", depending on whether it is instantiated as an anti-lepton or not. Within these functions, an enumeration is defined to aid readability and a switch statement allows the corresponding particles to be constructed and stored in decay_products. The vector stores shared pointers to remove the need to copy an object whenever its extracted from the class by its getter. However, since the pointer will always point to that specific particle type, the validation regarding valid decay modes cannot be bypassed outside of the class as one cannot replace the valid object at that address with an invalid one.

3.5 Boson classes

The only scalar boson is the Higgs boson, hence it is assigned a spin of 0. The rest are spin-1 vector bosons. Besides the W^+/W^- bosons, they are all their own antiparticles. Here, the boolean is_anti argument being true creates a W^+ by convention.

3.5.1 Photon

This has no additional data members. It is always given zero mass and a spin of 1, thus the parameterised constructor only accepts 3 momentum components.

3.5.2 Gluon

Gluon has 2 extra string data members called gluon_colour and gluon_anticolour, which are defined by the user in the parameterised constructor, but have default values of "red" and "antired" respectively. As mentioned previously, Quark is defined as a friend class to Gluon to share its colour validation methods with the Quark instances.

3.5.3 ZBoson, WBoson, HBoson

All 4 of these bosons (Z, W^+, W^-, H) have one extra data member, a vector called decay_products, containing shared pointers to Particle objects with identical implementation to that described for the Tau class. However, each of these bosons has its own unique set of allowed decay modes. The sets can be seen at the top of these classes' .cpp files (link to GitHub Release within title page footnote).

4 Catalogue implementation

This class is a wrapped STL multimap between user-defined string keys and shared pointers to Particle objects. An associative container was employed to allow users to create their

own meaningful categories when grouping particles inside the catalogue. However, beyond getting or printing stored particles via a key, template methods allow the same functionality but based upon particle type. So, pointers to specific particle types can be returned or their data printed, regardless of their associated key. An example snippet is given in listing 2. A private field called particle_count is updated for every added object, and "get_particle_count()" returns this number. The static string data member particle_catalogue_title is an ASCII art title for a print_catalogue() method [1]. Another method is "get_total_four_momentum()", which returns the total 4-momentum of the catalogue. This requires Catalogue to be a friend class of Particle.

Listing 2: Template for printing catalogue members of a given particle type.

```
template<typename type_input> void Catalogue::print_particles_of_type()
{
  for(auto iterator = catalogue.begin(); iterator != catalogue.end();
      ++iterator)
  {
    if(typeid(*iterator->second) == typeid(type_input))
    {
      iterator->second->print_particle_data();}
    }
  cout<<endl;
}
```

5 User interface

A user who has included the necessary particle catalogue files in their code could begin by instantiating a default constructed Catalogue object. Then, they could use the parameterised constructors of particle varieties to build physically valid instances of said particles (note these users should see the class interfaces to ensure correct construction). After this, the Catalogue method "add_particle()" can be employed to append shared pointers to these particles to the catalogue, along with an associated string key. An example of this procedure is depicted in Figure 2.

```cpp
int main()
{
  Catalogue particle_catalogue;

  // Examples of adding quarks to catalogue
  particle_catalogue.add_particle("up quark", make_shared<Quark>("up", "green", false, 2, 3, 4));
  particle_catalogue.add_particle("antidown quark", make_shared<Quark>("down", "antired", true, 5, 6, 7));

  // Examples of adding charged leptons to catalogue, electron will have their calorimeter energy deposits modified
  particle_catalogue.add_particle("electron", make_shared<Electron>(false, 100, 200, 300, 1, 2, 3, 4));
  particle_catalogue.add_particle("antimuon", make_shared<Muon>(true, true, 0, 0, 0));
  particle_catalogue.add_particle("tau", make_shared<Tau>(false, 10 , 20 , 30, "anticharm", "strange"));
  particle_catalogue.add_particle("antitau", make_shared<Tau>(true, 10 , 20 , 30, "antimuon", "muon neutrino"));

  // Examples of adding neutrinos to catalogue
  particle_catalogue.add_particle("electron neutrino", make_shared<Neutrino>("electron", false, false, 100, 200, 500));
  particle_catalogue.add_particle("muon neutrino", make_shared<Neutrino>("muon", false, true, 100, 200, 500));
  particle_catalogue.add_particle("antitau neutrino", make_shared<Neutrino>("tau", true, false, 100, 200, 500));

  // Examples of adding bosons to catalogue
  particle_catalogue.add_particle("photon", make_shared<Photon>(1, 2, 3));
  particle_catalogue.add_particle("gluon", make_shared<Gluon>("green", "antiblue", 10, 20, 30));
  particle_catalogue.add_particle("z boson", make_shared<ZBoson>(100, 200, 300, "antiup", "up"));
  particle_catalogue.add_particle("w- boson", make_shared<WBoson>(0, 100, 200, 300, "antiup", "down"));
  particle_catalogue.add_particle("w+ boson", make_shared<WBoson>(1, 100, 200, 300, "antielectron", "electron neutrino"));
  particle_catalogue.add_particle("higgs boson", make_shared<HBoson>(100, 200, 300, "w bosons"));
```

Figure 2: Several demonstrations of simultaneously instantiating a particle variety whilst adding to a catalogue object a shared pointer to that particle. The string keys chosen here simply convey the particle being stored.

Perhaps the user intends to manipulate particles stored within the catalogue. An example of such manipulation is given in Figure 3a. First, it shows the syntax required for the extraction of all particles of a given type from the catalogue, all quarks in this case.

Its possible a more unique grouping of particles has been implemented too, say all particles with low momentum transverse to a given direction were of interest and thus stored together. This could be conveyed through a user-defined key. An example of how to extract by key is also shown in Figure 3a. Next, a Tau object is selected and the energy modified using the Particle class setter; this tau particle was initially on-shell, thus its 3-momentum components will be scaled to keep it on-shell with the updated energy. The properties of the tau particle are subsequently printed to the terminal. Notice the optional boolean argument to "print_particle_data()", meaning the properties of this tau's decay products (specified when instantiated) will also be printed. The output of this is shown explicitly in 3b.

Figure 3a ends with an example of printing the catalogue, along with how one can access the total number of particles stored within the catalogue.

```
79  // Example of extracting particles from catalogue by their type
80  vector<shared_ptr<Particle>> quarks_in_catalogue{particle_catalogue.get_particles_of_type<Quark>()};
81
82  // Extracting from catalogue by key, then showing set_energy corrects momenta to be on-shell,
83  // and then using optional boolean argument into print_particle_data to print full decay product info
84  vector<shared_ptr<Particle>> particles_extracted{particle_catalogue.get_particles_with_key("tau")};
85  shared_ptr<Particle> extracted_tau{particles_extracted.at(0)};
86  extracted_tau->set_energy(2e3);
87  extracted_tau->print_particle_data(true); cout<<endl;
88
89  particle_catalogue.print_catalogue();
90  cout<<"\nNumber of particles in the catalogue is "<<particle_catalogue.get_particle_count()<<".\n"<<endl;
```

(a) Example code snippet.

```
tau properties:
-----------------------
Lepton number = 1
Rest mass (MeV) = 1.8e+03
Charge (e) = -1
Spin = 0.5
Energy (MeV) = 2e+03
Momentum [P_x, P_y, P_z] (MeV) = [233, 466, 699]
Decay products =  tau neutrino, anticharm, strange

---------------
DECAY PRODUCTS:
---------------


tau neutrino properties:
-----------------------------
Lepton number = 1
Rest mass (MeV) = 0
Charge (e) = 0
Spin = 0.5
Energy (MeV) = 0
Momentum [P_x, P_y, P_z] (MeV) = [0, 0, 0]
Interaction status = 0


anticharm quark properties:
-----------------------
Quark number = -0.333
Colour charge = antired
Rest mass (MeV) = 1.3e+03
Charge (e) = -0.667
Spin = 0.5
Energy (MeV) = 1.3e+03
Momentum [P_x, P_y, P_z] (MeV) = [0, 0, 0]


strange quark properties:
-----------------------
Quark number = 0.333
Colour charge = red
Rest mass (MeV) = 95
Charge (e) = -0.333
Spin = 0.5
Energy (MeV) = 95
Momentum [P_x, P_y, P_z] (MeV) = [0, 0, 0]
```

(b) Printed tau and decay product properties.

Figure 3: Here is a snippet of example code, described in detail in the main body, illustrating how a user may wish to operate on particle objects. The terminal output given in (b) reflects the calling of "print_particle_data()" on line 87 of (a).

The beginnings of the terminal output once "print_catalogue()" is called is shown by Figure 4. Note this figure matches the output for a particle catalogue built to hold the complete particle zoo, without any repeating particle types. The order in which particle properties are printed corresponds to ascending alphabetical order of the keys associated with the particles.

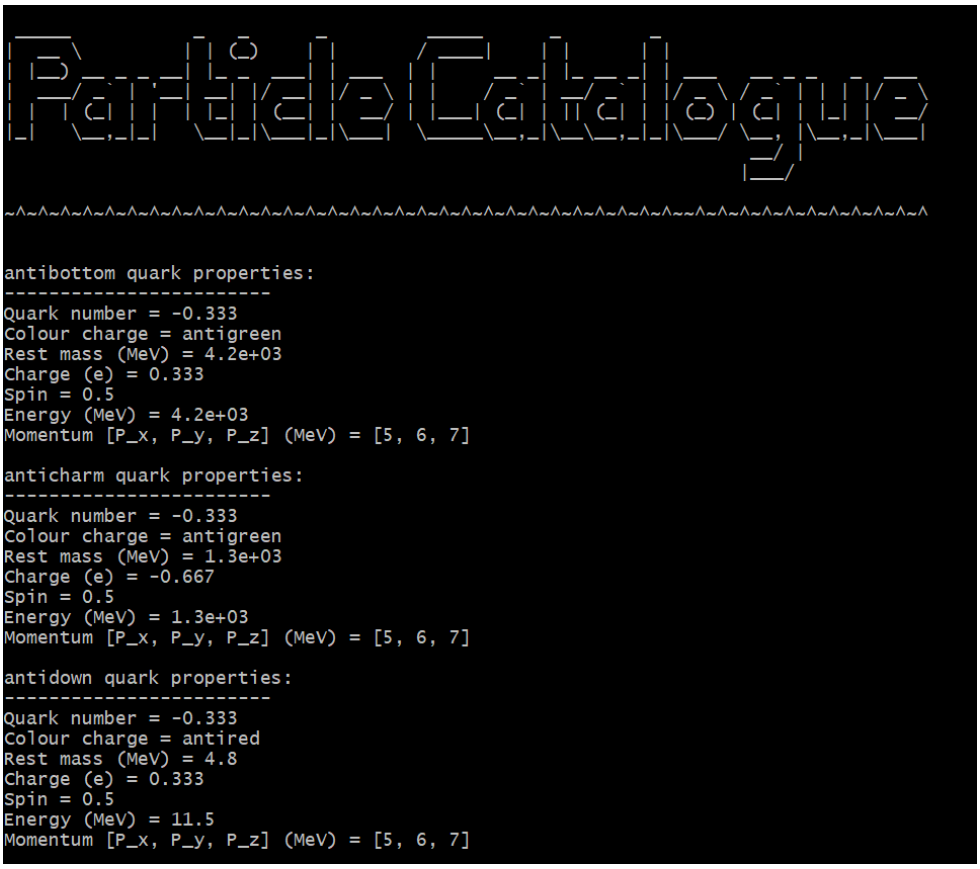

Figure 4: First 3 outputs of "print_catalogue()" to the terminal for a catalogue containing all Standard Model particles once each.

6 Discussion

Overall, the user experience currently is rather straightforward, and the implementation of the Standard Model in C++ has been successful. However, there are a range of improvements which could have been made to this project if time permitted. For example, despite full rule-of-five implementation, there is an opportunity for particle properties to bypass validation when move semantics are invoked. The null state of the particle data was moved from includes masses/charges of zero, despite this being inappropriate for some derived classes.

Also, the decay products data member of Tau, WBoson, ZBoson, HBoson could be improved. Namely, there is no setter for decay products yet, beyond that established in the parameterised constructor. Perhaps, an algorithm could be implemented allowing the linear momentum of the parent particle to be shared amongst the decay products, respecting its conservation post-decay.

Finally, it could be useful to implement a method of the Catalogue class which enables the removal of particles from the catalogue associated with a particular key.

References

[1] *ASCII Art Archive*. en. URL: https://www.asciiart.eu (visited on 05/10/2024).

[2] *Particle Data Group*. en. URL: https://pdg.lbl.gov/2023/listings/contents_listings.html (visited on 05/10/2024).

[3] N Srimanobhas. "Introduction to Monte Carlo for Particle Physics Study". en. In: ().