PARTICLE CATALOGUE REPORT

Clive Marvelous (10916086)

Abstract: This report presents the code architecture and implementation of a particle cataloging system. This system is designed to categorise and manage various particles from the Standard Model of particle physics. The system employs the principles of object-oriented programming- encapsulation, abstraction, inheritance and polymorphism- to provide a robust and accurate cataloging system. A three-level class hierarchy forms the code architecture. Starting with an abstract particle base class, which branched into more specialised classes for leptons, quarks and bosons. It further divides into even more specialised derived classes for individual particles like electrons and muons. This core structure also relies on an external four-momentum class. The code implementation involves a structured storage system using a map container that organises particles by category and type. There are 36 particle/antiparticle objects constructed at the end of this project.

1 Introduction

This report details the design, implementation, and results of a particle cataloging system. The main objective of this project is to provide a robust framework and a standardised user interface to catalog Standard Model particles, in agreement with the laws of physics [1]. The system's architecture is designed on the principles of object-oriented programming, specifically encapsulation, abstraction, inheritance, and polymorphism [2]. Encapsulation ensures private data within objects is protected from unauthorised access. Abstraction simplifies interactions with the systems by hiding complex implementation details behind the scene. Inheritance allows the creation of a hierarchical class structure where derived classes inherit data and behaviours from their base classes, enabling code reusability and scalability. Polymorphism enables the same function to perform different tasks when accessed from objects of different classes.

Briefly, the design of this system incorporates a three-level class hierarchy. The base class, `particle`, provides a common interface and defines shared functionalities for all particles. This base class is inherited by intermediate classes of `lepton`, `quark`, and `boson`, which then are specialised further into specific particle classes such as `electron`, `muon`, and `photon`. This design demonstrates the four fundamental principles of object-oriented programming. Firstly, encapsulation is demonstrated through safeguarding of particle data within class structures and abstraction is shown by providing a simple and standardised user interface. Furthermore, not only does this architecture takes advantage of inheritance to distribute shared properties but also polymorphism to handle diverse particle behaviors.

To implement this framework effectively, a map container within the main function is used to store the constructed particles. This container maps particle categories (lepton, quark or boson) and particle type (particle or antiparticle) pair with base class pointer vectors of the relevant particle objects. This database system allows for an efficient data management, operations and categorisation.

In summary, this report will explore the design of this system and why it is designed the way they are. At the end, the result will show how this system has achieved its objective.

2 Architecture

The structure of this project consist of three-level inheritance class hierarchy as shown in Figure 1. The first level consists of particle abstract base class. The second level consists of

the lepton, quark and boson derived classes. The third level consists of all the specific particle derived classes such as electron, muon and photon. Furthermore, the base class depends on an external four momentum class.

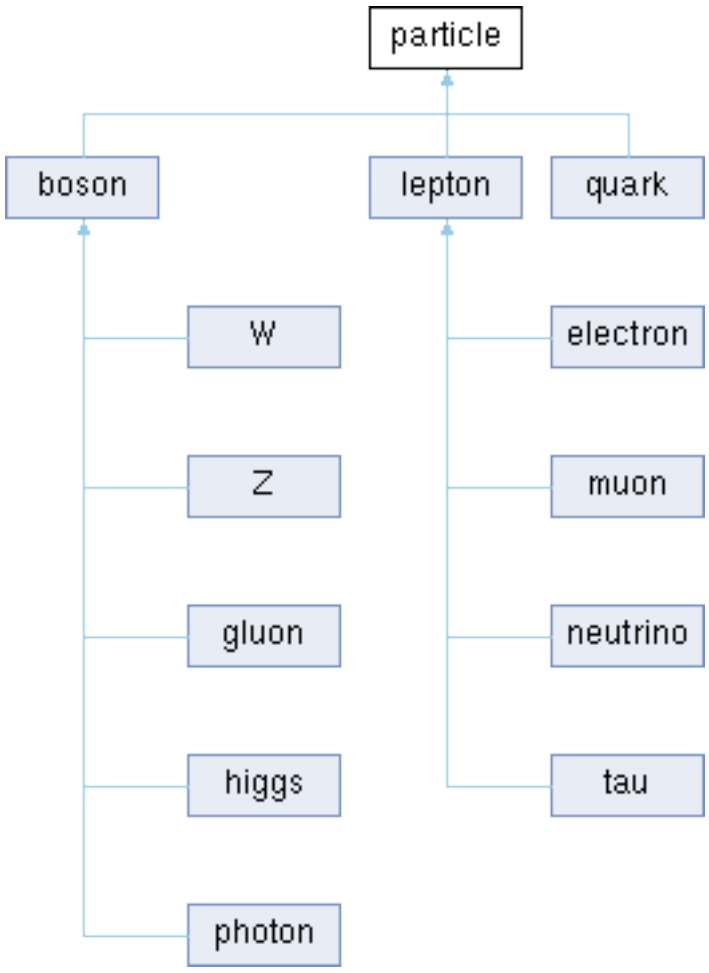

Figure 1: Class diagram showing generalisation between particle class and various derived classes. Figure was generated using Doxygen [3].

2.1 Level One: Particle Abstract Base Class

The `particle` abstract base class acts as the foundation of the entire system. It also provides a standard user interface for all particle objects.

Firstly, this class encapsulates common data members such as particle name and spin value. One important data member is a pointer to an external class `fourMomentum`, which stores the particle's four-momentum data. Moreover, there is a static data member `particle_count` that keeps track of the total number of particle objects constructed (static count is present in all distinct particle classes). Another data member worth mentioning is rest mass. This attribute is not a requirement and the user has no direct access to it. However, the rest mass value is used in the four-momentum validation function to ensure that the four-momentum inputs make physical sense. The data members and the validation function mentioned are illustrated below in Code 1.

```
protected:
  // Data members
  string particle_name;
  ParticleType particle_type;
```

```
  ParticleClass particle_class;
  float charge;
  float spin;
  double rest_mass; // MeV/c^2
  std::shared_ptr<fourMomentum> four_momentum;
  static int particle_count;

  // Validation functions
  void particle::validate_four_momentum_general(double E) // For all
   particles
  {
  // Check for realistic energy input
  if(E < 0) throw std::logic_error("Energy >= 0 for particles");
  }
  virtual void validate_four_momentum_mass(double E, double px, double py,
   double pz);  // For mass particles
  {
  // Calculate magnitude of momentum square
  double magnitude{magnitude_square(px, px, py, py, pz, pz)};
  // Tolerance for comparison
  const double tolerance = rest_mass * 0.2;
  // Check for realistic momentum inputs for mass particles
  if(!(std::sqrt(E*E - magnitude) <= rest_mass + tolerance)  || !(std::sqrt(
   E*E - magnitude) >= rest_mass - tolerance))
  {
    std::cout<<"Default values applied since input values not physically
   possible"<<std::endl;
    // Assign default values
    four_momentum->set_E(rest_mass);
    four_momentum->set_px(0);
    four_momentum->set_py(0);
    four_momentum->set_pz(0);
  }
}
```

Code 1: Data member and validation function of particle base class

Code 1 also shows that if the input four-momentum values are physically impossible, such as when the energy is negative, an exception will be thrown or the default four-momentum values will be assigned to that particle. This validation functions are called in the parameterised constructor of `particle` class or subsequent derived classes. This ensures a robust system.

The five foundational class features - default constructor, parameterised constructor, virtual destructor, copy and move semantics - are clearly defined in this class. Virtual destructor is crucial to prevent double deletion of memory when derived class object is destroyed. This class also provides complete getter and setter functions that give user access to the data members. To streamline particle categorisation into particle or antiparticle, this class utilises `enum class ParticleType` as shown in Code 2 below.

```
enum class ParticleType{PARTICLE, ANTIPARTICLE};

// Particle class constructor
particle(string name_in, ParticleType particle_type_in, ParticleClass
    particle_class_in, float charge_in, float spin_in, double E, double px,
    double py, double pz);
```

Code 2: enum class ParticleType to differentiate particle and antiparticle

There are several methods defined in this class, mainly to perform various physical calculations. Firstly, `restMmass()` function to calculate the rest mass of particles. Moreover, there

are also three friend functions defined in this class to perform various calculations involving the four-momentum values, they are: `sum(const particle& p1, const particle& p2)`, `subtract (const particle& p1, const particle& p2)` and `dotProduct(const particle& p1, const particle& p2)`. Lastly, `print_data()` function prints all the data stored in the object.

Furthermore, to provide a standardised interface for the user, there are numerous pure virtual functions defined within this class. These functions are intended for specific behaviours in the derived classes. For example, `get_muon_isolation()` is intended to be implemented in `muon` subclass. While most of these functions are intended only for the specific derived classes, it is still necessary to declare them in this class. This allows objects of derived classes to be managed through base class pointers. As a result, this design standardises user interface and thus encourages code overloading within the system.

It is important to mention that a pure virtual clone function is also defined in this class. This clone function will be overridden in every derived classes to return the object itself when called. This enables polymorphic deep copy of base class pointers without the need to know their specific class. This function will be extremely important for some derived classes like `tau` class.

2.2 Level Two:

The second level consists of three derived classes: lepton, quark and boson. The lepton and boson derived classes will be explained together due to their similar code structure, while the quark derived class will be explained separately.

2.2.1 Level Two: Lepton and Boson Derived Classes

Both the `lepton` and `boson` classes inherit from `particle` class and share a similar design and implementation. They are specialised to handle properties specific to leptons and bosons, respectively.

There are additional data members declared in the protected parts on these classes, such as the `lepton_number` variable and `decay_products` vector, to store the lepton number of lepton particles and decay products of boson particles, respectively.

Each class has its own specific parameterised constructor, virtual destructor, copy and move semantics. The constructor, copy and move semantics will first call the corresponding functions of the base class to ensure proper initialisation and resource management. Lepton class takes inputs of particle name, particle type (particle or antiparticle), charge, spin, four-momentum values and lepton number. Boson class takes the same inputs except for lepton number.

Both classes override the clone function. This function returns an object of the same class, ensuring that each class can produce a copy of itself accurately.

Furthermore, to ensure that both lepton and boson classes are not abstract, all inherited pure virtual functions from particle class must be overridden. Most of these functions are irrelevant for that particular class, hence, they will throw an exception when called inappropriately as shown in Code 3 below.

```
1 void lepton::set_colour(ColourCharge colour_in) {throw std::logic_error("
    set_colour is available only for boson class");}
```

Code 3: Overriding pure virtual class inside lepton class

2.2.2 Level Two: Quark Derived Classes

The `quark` class, derived from the `particle` class, has a distinct design and implementation compared to the`lepton` and `boson` classes. Unlike these classes, `quark` class does not serve as a

base for further derived classes because quark particles share the same fundamental behaviours and only differ in their property values. Therefore, it is more efficient to immediately categorise them into different particles when the quark class is constructed.

There are additional data members declared in the protected parts on this class. The `baryon_number` and `qurk_colour` variables are used to store the baryon number and colour charge of quark particles respectively.

Quark class has its own specific parameterised constructor, virtual destructor, copy and move semantics. Similar to the other classes, the constructor, copy and move semantics will first call the corresponding functions of the base class. However, unlike lepton and boson classes, quark class takes inputs of quark particle, particle type, colour charge and four-momentum values. Based on the quark particle input, the parameterised constructor will automatically assign relevant values to the data members. This is achieved through a structured approach involving a `struct`, a `static std::map`, and `enum class` for efficient categorisation and updating of the properties. The relevant code structure is illustrated below in Code 4.

```
private:
  // Simple struct to streamline information update
  struct quarkInformation
  {
    // Data members
    std::string name;
    float charge;
    float baryon_n;
    double rest_mass_value; // MeV
    // Constructor
    quarkInformation(string n, float c, float b, double rm) : name{n},
   charge{c}, baryon_n{b}, rest_mass_value{rm} {}
  };

  // Static map container to streamline data member update
  static const std::map<std::pair<QuarkParticle, ParticleType>,
   quarkInformation> information
  {
  {{QuarkParticle::UP, ParticleType::PARTICLE}, {"Up", 2.0/3.0, 1.0/3.0,
   2.2}},
  {{QuarkParticle::UP, ParticleType::ANTIPARTICLE}, {"Anti-Up", -2.0/3.0,
   -1.0/3.0, 2.2}},
  ...
  }

  // Update function
  void update_information()
  {
  // Find and update data
  auto information_it = information.find({quark_particle, particle_type});
  if(information_it != information.end())
  {
    particle_name = information_it->second.name;
    charge = information_it->second.charge;
    baryon_number = information_it->second.baryon_n;
    rest_mass = information_it->second.rest_mass_value;
  }
  };

public:
  // Constructor
  quark(QuarkParticle quark_particle_in, ParticleType particle_type_in,
```

```
ColourCharge quark_colour_in, double E, double px, double py, double pz);
```

Code 4: Quark class design and implementation

Code 4 demonstrates that the `quarkInformation` struct centralises quark-specific information: name, charge, baryon number, and rest mass. The static map `information` maps the 'key' (`QuarkParticle` and `ParticleType` pairs) to the corresponding `quarkInformation`. This map is static and shared among all instances to prevent redundant instantiation of the same map. During construction, the constructor will call `void update_information()` function to assign the appropriate values based on the inputs provided. This allows each quark object to be initialised with a complete set of accurate physical properties. This exact mechanism is also applied for quark particle setter function.

Similar to the`lepton` and `boson` classes, the `quark` class also overrides all inherited pure virtual functions to ensure that this class is not abstract. Furthermore, this class also override the clone function to return an object of the same class.

2.3 Level Three: Specific Particle Derived Classes

Third-level subclasses derived from either the `lepton` or the `boson` classes. Moreover, they all have a similar code design and specially structured to manage specific particles. As such, some of their data member values are already predefined when the parameterised constructor of the base class is called. Code 5 below shows the parameterised constructor of `electron` class.

```
electron::electron(ParticleType particle_type_in, double E, double px,
    double py, double pz, double EM_1, double EM_2, double HAD_1, double
    HAD_2) :
  lepton("electron", particle_type_in, -1, 0.5, 1, E, px, py, pz)
  {
    rest_mass = 0.511; // MeV/c^2
    // Validate energy and momentum inputs
    validate_four_momentum_mass(E, px, py, pz);
    // Check for antiparticle
    if(particle_type == ParticleType::ANTIPARTICLE)
    {
      // Update data members
      particle_name = "antielectron";
      charge = 1;
      lepton_number = -1;
    }
    // Validate and store energy deposit values into vector
    validate_energy_deposit(E, EM_1, EM_2, HAD_1, HAD_2);
    energy_deposited_in_calorimeter = {EM_1, EM_2, HAD_1, HAD_2};
    // Update electron count
    ++electron_count;
  }
```

Code 5: Parameterised constructor of electron class

The code above also highlights further validation of four-momentum inputs in each subclasses, except for the `neutrino` class. This validation utilises the rest mass values of the particles to ensure the invariant mass of the four-momentum will return the rest mass of the particles within 20% (for mass particles) and 10% (for massless particles) margin of error. The `neutrino` class has no further input validation because its rest mass is negligibly small yet they are not massless. Hence, it is difficult to predefined its rest mass.

A significant feature in some of these derived classes is overloaded parameterised constructors. The selection of these constructors depend on the decay products of the particles, as illustrated in Code 6 below, taken from the `Z` class:

```
protected:
// Update functions
...
void update_quark_decay(QuarkParticle quark_decay_product_in) // For decay to quark
{
  // Declare decay products
  decay_products.push_back(std::make_shared<quark>(quark_decay_product_in, ParticleType::PARTICLE, ColourCharge::RED, 0, 0, 0, 0));
  decay_products.push_back(std::make_shared<quark>(quark_decay_product_in, ParticleType::ANTIPARTICLE, ColourCharge::RED, 0, 0, 0, 0));
}

public:
Z(LeptonParticle lepton_decay_product_in, double E, double px, double py, double pz) : ... // For decay to lepton (Other than neutrino)
Z(Flavour neutrino_decay_product_in, double E, double px, double py, double pz) : ... // For decay to neutrino
Z(QuarkParticle quark_decay_product_in, double E, double px, double py, double pz) : boson("Z", ParticleType::PARTICLE, 0, 1, E, px, py, pz)
{
  ...
  // Update data members
  update_quark_decay(quark_decay_product_in);
  ...
}
```

Code 6: Overloaded parameterised constructor of Z class

Code 6 above demonstrates the polymorphic advantage of C++. Each constructor is design for a specific final decay products: lepton, neutrino or quark. For the quark decay, the constructor will initialise a Z object and call `update_quark_decay(quark_decay_product_in)` update function. This function when called will automatically construct quark-antiquark pair decay products and store them inside the vector. Using overloaded constructors allows a seamless construction of Z boson in accordance with the principles of particle physics.

2.4 External: Four Momentum Class

The final component of this system is the `fourMomentum` class. This class specialises in storing and handling the four-momentum vectors associated with each particle. This class is crucial for performing physical calculations essential in particle physics, which are adding and subtracting four-momentum vectors, computing dot products, and calculating invariant masses.

Additionally, the `fourMomentum` class incorporates a template function to compute the square magnitude of three-momentum vectors. This template is shown below in Code 7.

```
template<class c_type> c_type magnitude_square(c_type x1, c_type x2, c_type y1, c_type y2, c_type z1, c_type z2)
  {return x1*x2 + y1*y2 + z1*z2;}
```

Code 7: Template function to calculate square magnitude of three momentum

This generic functionality is integrated into many functions within the system, including the `dotProduct` friend function in the `fourMomentum` class and the `validate_four_momentum_mass` function in the `particle` class. Making this function abstract enables code re-usability throughout the system.

3 Implementation and Result

All classes explained above are instantiated within the main function, where particles constructed are organised inside a map container. This container maps pairs of particle categories (lepton, quark, or boson) and particle types (particle or antiparticle) with vectors containing the relevant particle objects. This organisation is demonstrated in Code 8 below.

```
enum class ParticleCategory{LEPTON, QUARK, BOSON};
enum class ParticleType{PARTICLE, ANTIPARTICLE};

int main()
{
std::map<std::pair<ParticleCategory, ParticleType>, std::vector<std::
    shared_ptr<particle>>> particle_catalogue
{
    {{ParticleCategory::LEPTON, ParticleType::PARTICLE},
    {
      std::make_shared<electron>(ParticleType::PARTICLE, 0.511, 0, 0, 0, 0,
    0, 0, 0),
      std::make_shared<muon>(ParticleType::PARTICLE, true, 0, 0, 0, 0),
      ...
    }}
}
...
}
```

Code 8: Particle catalogue map container inside main function

Utilising a map container simplifies the management and organisation of various particle objects, as it allows data extraction and management based on specific criteria. Some of these functions, which facilitate data extraction, management and printing, are presented below in Code 9.

```
void print_particles_by_key(const ParticleCatalogueMap& catalogue,
    ParticleCategory category, ParticleType type); // Print all particles in
    the same category
void print_particles_by_name(const ParticleCatalogueMap& catalogue, const
    std::string& name); // Print particle data by name
int total_particles_in_catalog(const ParticleCatalogueMap& catalogue); //
    Count total particle in catalogue
int count_particles_by_type(const ParticleCatalogueMap& catalogue,
    ParticleType type); // Count particle by type
int count_particles_by_category(const ParticleCatalogueMap& catalogue,
    ParticleCategory category); // Count particle by category
std::vector<std::shared_ptr<particle>> get_particles_by_type(const
    ParticleCatalogueMap& catalogue, ParticleCategory category, ParticleType
    type); // Get sub-container
fourMomentum sum_four_momentum(const ParticleCatalogueMap& catalogue); //
    Sum four momentum of all particles in the catalogue
void catalogue_summary(const ParticleCatalogueMap& catalogue); // Print
    catalogue report
```

Code 9: Functions to manage particle_catalogue map container

Functions such as `print_particles_by_key` and `print_particles_by_name` allow users to retrieve particle data based on category and type, or name, respectively. Additionally, the `get_particles_by_type` function return vector containing relevant objects from the specified input categories without the need for deep copy.

The `particle_catalogue` map container holds a total of 36 particle and antiparticle objects. This count exceeds the initial requirement of 32 due to multiple instantiations of the same

particles. This is done to demonstrate the usefulness of overloaded constructors to generate various decay products for the same particle class. The function `catalogue_summary` provides a comprehensive report of the container's contents, listing the total number of particles in the container, as well as the number of particles, antiparticles, leptons, quarks, and bosons. Lastly, the `static_summary` function outputs the count of distinct particle types constructed, including those constructed as decay products. This function utilises the static count of each classes.

4 Discussion and Conclusion

In conclusion, this project successfully achieved its objective of cataloging the currently known particles within the Standard Model. This is achieved through providing a robust and organised system to manage particle information. However, the system has its limitation as it is not equipped to catalog particles that may be discovered in the future. Moreover, to enhance its applicability, this cataloging system can be integrated with a detector simulation software. This integration would allow for a seamless recording and organisation of the detected particles, including their four-momentum values and decay products. Therefore, advancing study of particle physics.

5 The Use of AI

Generative AI Disclosure: I used ChatGPT 4 to assist in idea generation and for feedback on grammar and content. I implemented some of its recommendations. ChatGPT was also used in writing the advanced C++ code. For instance, it recommended using a map container in the quark class for more efficient way to assign values to the data members instead of using a vector.

References

[1] Domain of Science Dominic Walliman. Map of fundamental particles. `https://www.flickr.com/photos/95869671@N08/`, 2024. Accessed: 2024-05-08.

[2] Geeks for Geeks. Object-oriented programming in c++. `https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/`, 2024. Accessed: 2024-05-08.

[3] Doxygen. Doxygen: The documentation system for c++, c, java, objective-c, python, idl (corba and microsoft flavors), fortran, vhdl, php, c#, d. `https://www.doxygen.nl/`, 2024. Accessed: 2024-05-08.

Overleaf word count: 2408