```cpp
// Implementation of Particle.hpp - Luca Morelli 2021

#include "Particle.hpp"
#include "ParticleType.hpp"
#include "ResonanceType.hpp"

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>

// Initialization of static members
std::vector<ParticleType *> Particle::particleType_{};
int Particle::NParticleType_{0};

/*Member function definitions*/

// Returns the index of a particle or -1 if not found
int Particle::findParticle(std::string pName) {
  if (NParticleType_ == 0) {
    return -1;
  }
  for (int i{0}; i < NParticleType_; ++i) {
    if (particleType_[i]->getName() == pName) {
      return i;
    }
  }
  return -1;
}

// Particle constructor definition
Particle::Particle(std::string name, double Px, double Py, double Pz)
    : Px_{Px}, Py_{Py}, Pz_{Pz} {
  index_ = findParticle(name);
  if (index_ == -1) {
    std::cout << "ERROR: Particle " << name << " has still not been defined"
              << '\n';
  }
}

// Adds a new type of particle
void Particle::addParticleType(std::string name, double mass, int charge,
,                               double width) {
  if (NParticleType_ == maxNumParticleType) {
    std::cerr << "ERROR: reached maximum type number, can't add a new one\n";
  } else {
    if (findParticle(name) == -1) {
      if (width == 0) {
        particleType_.push_back(new ParticleType{name, mass, charge});
      } else {
        particleType_.push_back(new ResonanceType{name, mass, charge, width});
      }
      ++NParticleType_;
    }
  }
```

```cpp
}

// Sets the type of particle of a Particle object using the index of the type
void Particle::setParticle(int index) {
  if (index < NParticleType_ && index != -1) {
    index_ = index;
  } else {
    std::cerr << "ERROR: " << index
              << " is not a particle index alredy defined\n";
  }
}
// Sets the type of particle of a Particle object using the name of the type
void Particle::setParticle(std::string name) {
  setParticle(findParticle(name));
}

// Prints the types of particles already existing
void Particle::printParticleTypes() {
  for (auto &it : particleType_) {
    it->print();
    std::cout << '\n';
  }
}

// Prints the data of a Particle object
void Particle::printDetails() const {
  std::cout << "|Index:" << index_ << "|" << particleType_[index_]->getName()
            << "|Px:" << Px_ << "|Py:" << Py_ << "|Pz:" << Pz_ << '|';
}

// Gets data member and derived data
int Particle::getIndex() const { return index_; }
double Particle::getPx() const { return Px_; }
double Particle::getPy() const { return Py_; }
double Particle::getPz() const { return Pz_; }
double Particle::getMass() const { return particleType_[index_]->getMass(); }
int Particle::getCharge() const { return particleType_[index_]->getCharge(); }
double Particle::getEnergy() const {
  return sqrt(getMass() * getMass() + Px_ * Px_ + Py_ * Py_ + Pz_ * Pz_);
}
double Particle::invMass(Particle const &p2) const {
  double PxTot{Px_ + p2.getPx()};
  double PyTot{Py_ + p2.getPy()};
  double PzTot{Pz_ + p2.getPz()};
  return sqrt(pow(getEnergy() + p2.getEnergy(), 2) - PxTot * PxTot -
              PyTot * PyTot - PzTot * PzTot);
}
// Sets momentum vector
void Particle::setP(double Px, double Py, double Pz) {
  Px_ = Px;
  Py_ = Py;
  Pz_ = Pz;
}

// Operator Overload definition
std::ostream &operator<<(std::ostream &os, Particle const &particle) {
  particle.printDetails();
  return os;
}

// Management of the decay of a particle
int Particle::decay2body(Particle &dau1, Particle &dau2) const {
  if (getMass() == 0.0) {
    std::cout << "Decayment cannot be preformed if mass is zero\n";
    return 1;
  }

  double massMot = getMass();
  double massDau1 = dau1.getMass();
  double massDau2 = dau2.getMass();

  if (index_ > -1) {  // add width effect

    // gaussian random numbers

    float x1, x2, w, y1, y2;

    double invnum = 1. / RAND_MAX;
    do {
      x1 = 2.0 * rand() * invnum - 1.0;
      x2 = 2.0 * rand() * invnum - 1.0;
      w = x1 * x1 + x2 * x2;
    } while (w >= 1.0);

    w = sqrt((-2.0 * log(w)) / w);
    y1 = x1 * w;
    y2 = x2 * w;

    massMot += particleType_[index_]->getWidth() * y1;
  }

  if (massMot < massDau1 + massDau2) {
    std::cout << "Decayment cannot be preformed because mass is too low in "
                 "this channel\n";
    return 2;
  }

  double pout =
      sqrt(
          (massMot * massMot - (massDau1 + massDau2) * (massDau1 + massDau2)) *
          (massMot * massMot - (massDau1 - massDau2) * (massDau1 - massDau2))) /
      massMot * 0.5;

  double norm = 2 * M_PI / RAND_MAX;

  double phi = rand() * norm;
  double theta = rand() * norm * 0.5 - M_PI / 2.;
  dau1.setP(pout * sin(theta) * cos(phi), pout * sin(theta) * sin(phi),
            pout * cos(theta));
  dau2.setP(-pout * sin(theta) * cos(phi), -pout * sin(theta) * sin(phi),
            -pout * cos(theta));

  double energy = sqrt(Px_ * Px_ + Py_ * Py_ + Pz_ * Pz_ + massMot * massMot);

  double bx = Px_ / energy;
  double by = Py_ / energy;
  double bz = Pz_ / energy;

  dau1.boost(bx, by, bz);
  dau2.boost(bx, by, bz);

  return 0;
}

void Particle::boost(double bx, double by, double bz) {
  double energy = getEnergy();

  // Boost this Lorentz vector
  double b2 = bx * bx + by * by + bz * bz;
  double gamma = 1.0 / sqrt(1.0 - b2);
  double bp = bx * Px_ + by * Py_ + bz * Pz_;
  double gamma2 = b2 > 0 ? (gamma - 1.0) / b2 : 0.0;

  Px_ += gamma2 * bp * bx + gamma * bx * energy;
  Py_ += gamma2 * bp * by + gamma * by * energy;
  Pz_ += gamma2 * bp * bz + gamma * bz * energy;
}
```