

Сучасні мови та об'єктно-орієнтоване програмування в ядерній фізиці

Інкапсуляція та наслідування об'єктів в C++. Абстрактні класи. Інтерфейси.

Р.В. Єрмоленко

Зміст

1. Інкапсуляція. Модифікатори доступу.
2. Об'явлення класу та реалізація його методів.
3. Поняття наслідування.
4. Множинне наслідування.
5. Віртуальні функції.
6. Абстрактні класи. Інтерфейси та їх призначення.

Література

- Роберт Мартин. Чистый Код. Создание, анализ и рефакторинг.
- Бьярне Стауструп. Программирование: принципы и практика использования C++
- Эккель Брюс. Философия C++
- Герберт Шилдт. C++. Базовый курс
- Стенли Липпман, Жози Лажойе. C++ для начинающих
- Стивен Прата. Язык программирования C++
- Bjarne Stroustrup - The C++ Programming Language
- Deitel H. M., Deitel P.J. - C++: How to Program
- Bjarne Stroustrup - Programming: Principles and Practice Using C++

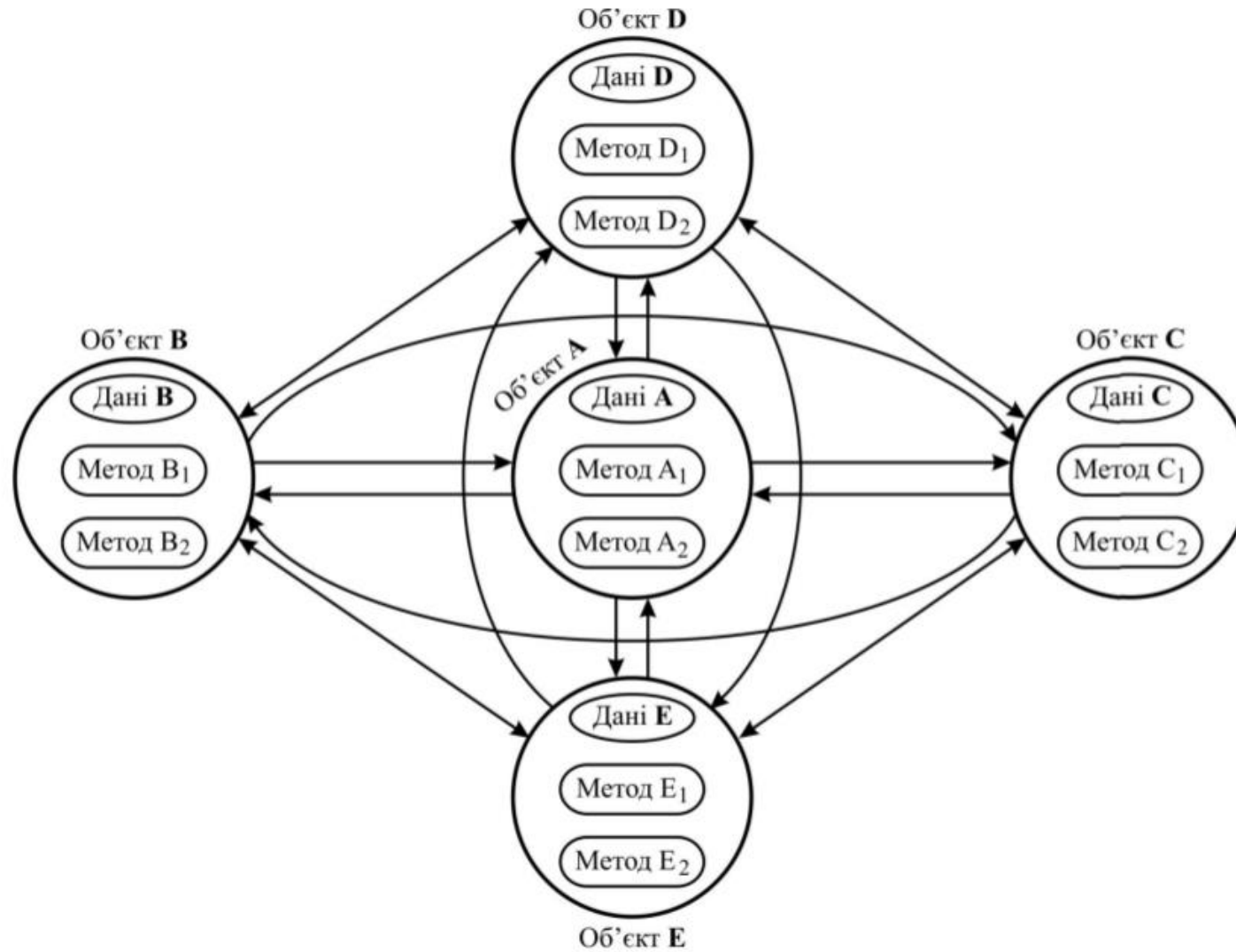
Питання з минулих лекцій

- В чому полягає відмінність між структурами в С та С++?
- Особливості організації пам'яті при в структурах.
- В чому відмінність між файлами *.cpp (*.c) та *.hpp (*.h)

ПОНЯТТЯ КЛАСУ

- Клас – основа об'єктно-орієнтованого підходу до програмування.
- Клас - новий тип даних, який задає формат об'єкта. Клас містить як дані, так і коди програм, призначені для виконання дій над ними.
- У мові програмування C++ специфікацію класу використовують для побудови об'єктів.
- Об'єкти – це примірники класового типу.
- Загалом, клас є набором планів і дій, які вказують на властивості об'єкта та визначають його поведінку.
- Важливо розуміти, що клас – це логічна абстракція, яка реально не існує доти, доки не буде створено об'єкт цього класу, тобто це те, що стане фізичним представленням цього класу в пам'яті комп'ютера.

ОБ'ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД ДО ВСТАНОВЛЕННЯ ЗВ'ЯЗКІВ МІЖ ДАНИМИ І МЕТОДАМИ (ФУНКЦІЯМИ)





ІНКАПСУЛЯЦІЯ. МОДИФІКАТОРИ ДОСТУПУ.

Варіант 1

```
class ім'я_класу {  
  private:  
    закриті дані та функції класу  
  public:  
    відкриті дані та функції класу  
} перелік_об'єктів_класу;
```

Варіант 2

```
class ім'я_класу {  
  private:  
    закриті дані та функції класу  
  public:  
    відкриті дані та функції класу  
};  
ім'я_класу перелік_об'єктів_класу;
```


ІНКАПСУЛЯЦІЯ. МОДИФІКАТОРИ ДОСТУПУ

```
inition.cpp ParticleDefinition.h Source.cpp
ParticleDefinition

#pragma once
#include <string>
class ParticleDefinition
{
private:
    std::string m_name;
    float m_mass;
    float m_Px;
    float m_Py;
    float m_Pz;
    float m_E;

public:
    float const& getEnergy();
    std::string const& getName(){ return m_name; }
    float const& getMass() { return m_mass; }
    void setMass(float const& mass);
    void setName(std::string const& name);
};
```

```
ParticleDefinition.cpp ParticleDefinition.h Source.cpp
Project1 ParticleDefinition

1 #include "ParticleDefinition.h"
2
3 inline float const& ParticleDefinition::getEnergy()
4 {
5     return m_E;
6 }
7
8 void ParticleDefinition::setMass(float const& mass)
9 {
10     m_mass = mass;
11 }
12 void ParticleDefinition::setName(std::string const& name)
13 {
14     m_name = name;
15 }
```

```
inition.cpp ParticleDefinition.h Source.cpp
(Global Scope)

#include <string>
#include <iostream>
#include "ParticleDefinition.h"

int main()
{
    ParticleDefinition p;
    p.setMass(1.f);
    p.setName("proton");
    std::cout << "mass = " << p.getMass() << std::endl;
    std::cout << "Name: " << p.getName() << std::endl;
    return 0;
}
```

КОНСТРУКТОРИ ТА ДЕСТРУКТОРИ

- **Конструктор** – це спеціальна функція-член класу, яка викликається при створенні об'єкта, а її ім'я обов'язково збігається з іменем класу.
- Автоматична ініціалізація членів-даних класу здійснюється завдяки використанню конструктора.
- В конструкторі також можливе динамічне виділення пам'яті для зберігання даних.
- **Деструктор** – це функція, яка викликається під час руйнування об'єкта.
- У багатьох випадках під час руйнування об'єкта необхідно виконати певну дію або навіть певні послідовності дій.
- Локальні об'єкти створюються під час входу в блок, у якому вони визначені, і руйнуються при виході з нього. Глобальні об'єкти руйнуються внаслідок завершення програми.
- Існує багато чинників, які вимагають використовувати деструктори. Наприклад, об'єкт повинен звільнити раніше виділену для нього пам'ять.

КОНСТРУКТОРИ ТА ДЕКТРУКТОРИ

```
ParticleDefinition.h  ParticleDefinition<T>

#pragma once
#include <string>

template <typename T>
class ParticleDefinition
{
private:
    std::string m_name;
    T m_mass;
    T m_Px;
    T m_Py;
    T m_Pz;
    T m_E;

public:
    inline T const& getEnergy() { return m_E; }
    std::string const& getName(){ return m_name; }
    T const& getMass() { return m_mass; }
    void setMass(T const& mass) { m_mass = mass; }
    void setName(std::string const& name) { m_name = name; }
};
```

```
template <typename T>
class ParticleDefinition
{
private:
    std::string m_name;
    T m_mass;
    std::vector<T> m_P;
    T m_E;

public:
    size_t const NUM_COMPONENT = 3;
    ParticleDefinition():
        m_name(""), m_mass(static_cast<T>(0.f)), m_E(static_cast<T>(0.f))
    {
        m_P.resize(NUM_COMPONENT, static_cast<T>(0.f));
    }
    ParticleDefinition(std::string name, T mass, T E) :
        m_name(name), m_mass(mass), m_E(E)
    {
        m_P.resize(NUM_COMPONENT, static_cast<T>(0.f));
    }
    ~ParticleDefinition()
    {
        m_P.clear();
        m_P.shrink_to_fit();
    }
};
```

ВИКОРИСТАННЯ ШАБЛОНІВ

```
ParticleDefinition.h  Source.cpp
ParticleDefinition<T>

#pragma once
#include <string>

template <typename T>
class ParticleDefinition
{
private:
    std::string m_name;
    T m_mass;
    T m_Px;
    T m_Py;
    T m_Pz;
    T m_E;

public:
    inline T const& getEnergy() { return m_E; }
    std::string const& getName(){ return m_name; }
    T const& getMass() { return m_mass; }
    void setMass(T const& mass) { m_mass = mass; }
    void setName(std::string const& name) { m_name = name; }
};
```

```
(Global Scope)
1  #include <string>
2  #include <iostream>
3  #include "ParticleDefinition.h"
4  #include <limits>
5
6  int main()
7  {
8      ParticleDefinition<double> p;
9      p.setMass(1.007276466879);
10     p.setName("proton");
11     std::cout.precision(std::numeric_limits< double >::max_digits10);
12     std::cout << "mass = " << p.getMass() << std::endl;
13     std::cout << "Name: " << p.getName() << std::endl;
14     return 0;
15 }
```

- **template <typename T>**- в шаблоні буде використовуватися вбудований тип даних, такий як: int, double, float, char ...
- **template <class T>** - в шаблоні функції в якості параметра будуть використовуватися класи.

КОНСТРУКТОРИ ТА ДЕКТРУКТОРИ

```
#include <string>
#include <iostream>
#include "ParticleDefinition.h"
#include <limits>

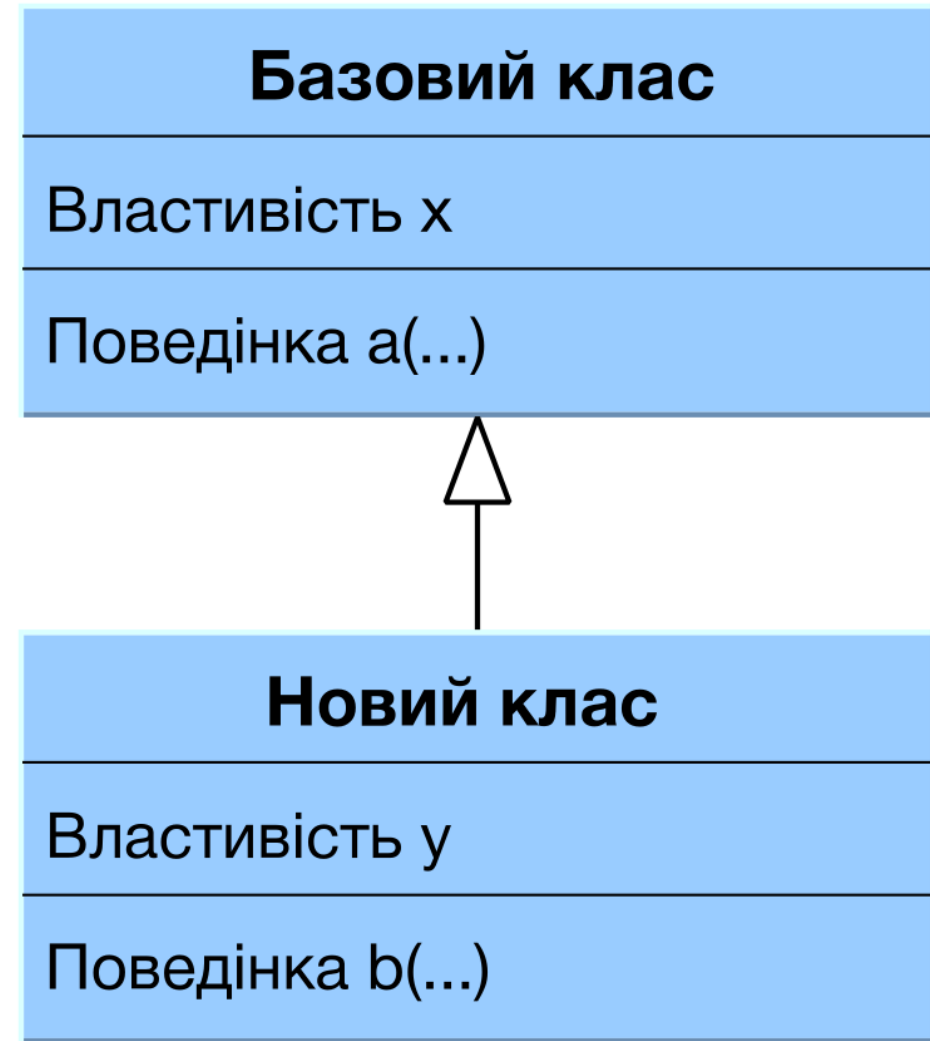
int main()
{
    ParticleDefinition<double> p("proton", 1.007276466879, 0.);
    // p.setMass(1.007276466879);
    // p.setName("proton");
    std::cout.precision(std::numeric_limits< double >::max_digits10);
    std::cout << "mass = " << p.getMass() << std::endl;
    std::cout << "Name: " << p.getName() << std::endl;
    p.~ParticleDefinition();
    return 0;
}
```

НАСЛІДУВАННЯ

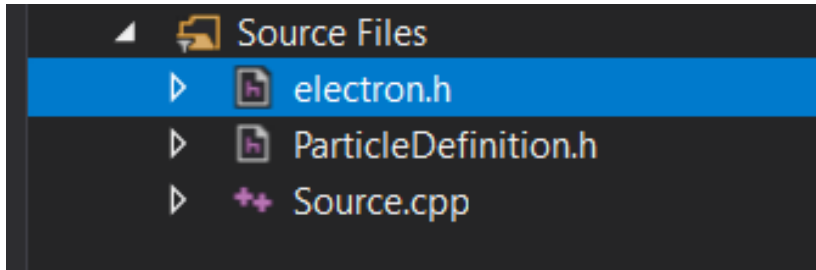
- Наслідування – один з трьох фундаментальних механізмів об'єктно-орієнтованого програмування, оскільки саме завдяки йому є можливість створювати ієрархічні побудови.
- Використовуючи механізми наслідування, можна розробити загальний клас, який визначає характеристики, що є властиві множині взаємопов'язаним між собою елементам.
- Цей клас потім може унаслідуватися іншими, вузькоспеціалізованими класами з додаванням у кожен з них своїх, властивих тільки їм унікальних особливостей.
- У стандартній термінології мови програмування C++ початковий клас називається базовим.
- Клас, який успадковує базовий клас, називається похідним.
- Похідний клас можна використовувати як базовий для іншого похідного класу. За таким механізмом і будується багаторівнева ієрархія класів.

НАСЛІДУВАННЯ

- Наслідування – надає можливість створювати ієрархічні побудови.
- Можна розробити базовий клас, який визначає характеристики, що є властиві множині взаємопов'язаним між собою елементам.
- Цей клас потім може унаслідуватися іншими, вузькоспеціалізованими класами з додаванням у кожен з них своїх, властивих тільки їм унікальних особливостей.
- Похідний клас можна використовувати як базовий для іншого похідного класу. За таким механізмом і будується багаторівнева ієрархія класів.



НАСЛІДУВАННЯ



```
template <typename T>
class ParticleDefinition
{
protected:
    std::string m_name;
    T m_mass;
    std::vector<T> m_P;
    T m_E;
public:
```

```
#pragma once
#include "ParticleDefinition.h"
template <typename T>
class Electron : public ParticleDefinition<T>
{
private:
    T m_electric_charge;
    T m_spin;
    std::string m_family;
public:
    void setFamily(std::string const& family) { m_family = "lepton"; }
    Electron() :
        m_electric_charge(-1.), m_spin (0.5), m_family("lepton")
    {
        this->m_name = "electron";
    }
    ~Electron() {}
    T getSpin() { return m_spin; }
    T getElectricCharge() { return m_electric_charge; }
    std::string getFamily() { return m_family; }
};
```


НАСЛІДУВАННЯ

Управління механізмом доступу до членів базового класу



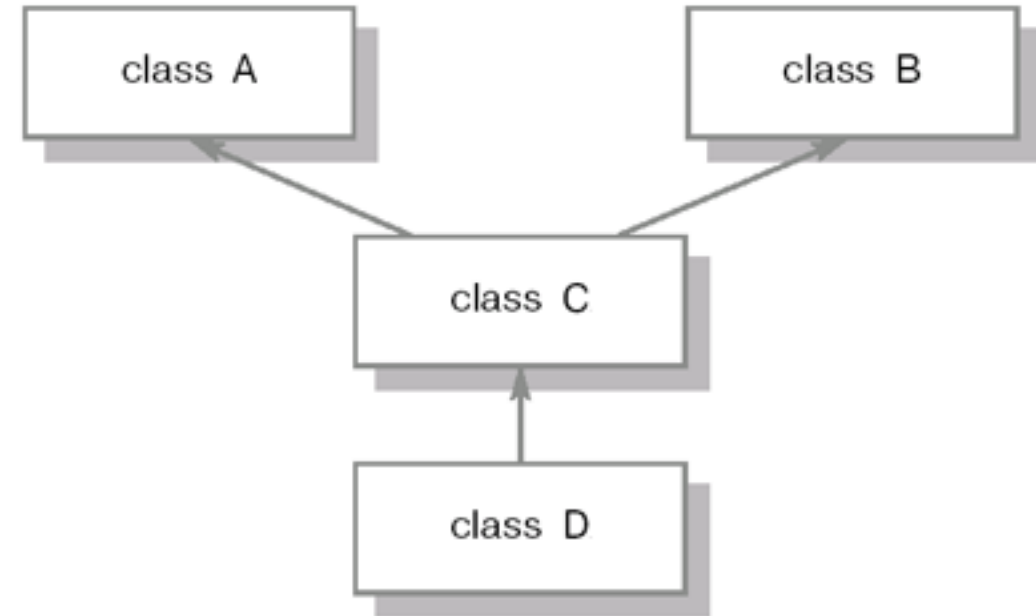
НАСЛІДУВАННЯ

```
#include <string>
#include <iostream>
#include "electron.h"
#include <limits>

int main()
{
    Electron<double> e;
    std::cout.precision(std::numeric_limits< double >::max_digits10);
    std::cout << "mass = " << e.getMass() << std::endl;
    std::cout << "Name: " << e.getName() << std::endl;
    std::cout << "Family: " << e.getFamily() << std::endl;
    std::cout << "Spin: " << e.getSpin() << std::endl;
    std::cout << "ElectricCharge: " << e.getElectricCharge() << std::endl;
    e.~Electron();
    return 0;
}
```

МНОЖИННЕ НАСЛІДУВАННЯ

- Множинна спадковість дозволяє класу успадковувати функціональність від декількох інших класів.
- Клас може бути похідним не лише від одного базового класу. А й від кількох.
- Створення класу на основі двох чи більше класів називається множинним наслідуванням.
- Синтаксис опису множинного наслідування схожий на синтаксис простого наслідування.



МНОЖИННЕ НАСЛІДУВАННЯ

```
1  #pragma once
2  #include "ParticleDefinition.h"
3  #include "constants.h"
4  template <typename T>
5  class Electron : public ParticleDefinition<T>, public Constants
6  {
7  private:
8      T m_electric_charge;
9      T m_spin;
10     std::string m_family;
11 public:
12     void setFamily(std::string const& family) { m_family = "lepton"; }
13     Electron() { ... }
14     ~Electron() {}
15     T getSpin() { return m_spin; }
16     T getElectricCharge() { return m_electric_charge; }
17     std::string getFamily() { return m_family; }
18 };
```

```
class Constants
{
public:
    double const c = 299792458;
    double const h = 6.62607004e-34;
    double const e = 1.6021766208e-19;
};
```

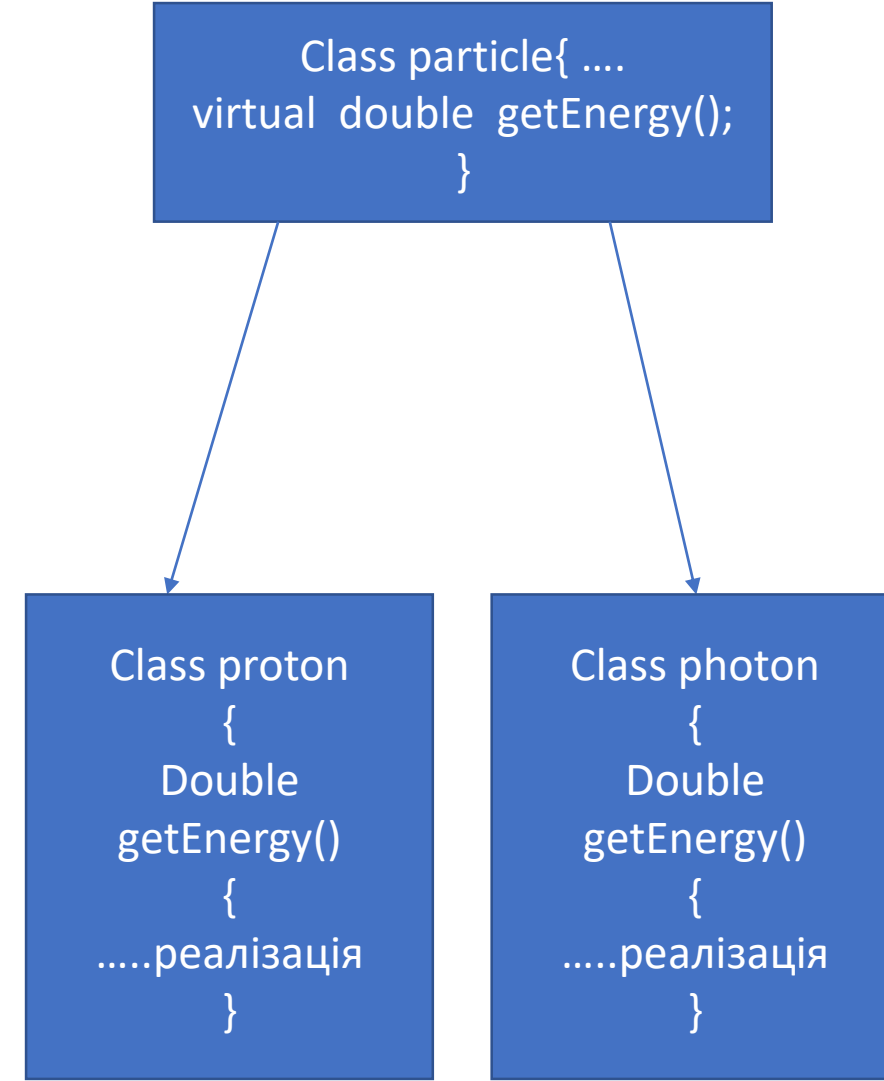
```
int main()
{
    Electron<double> e;
    std::cout.precision(std::numeric_limits< double >::max_digits10);
    std::cout << "mass = " << e.getMass() << std::endl;
    std::cout << "Name: " << e.getName() << std::endl;
    std::cout << "Family: " << e.getFamily() << std::endl;
    std::cout << "Spin: " << e.getSpin() << std::endl;
    std::cout << "ElectricCharge: " << e.getElectricCharge() << std::endl;
    std::cout << "h: " << e.h << std::endl;
    e.~Electron();
    return 0;
}
```

Віртуальні функції

Віртуальна функція – це така функція, яка оголошується в базовому класі з використанням ключового слова **virtual** і перевизначається в одному або декількох похідних класах.

Під час застосування віртуальних функцій часто трапляються ситуації, коли віртуальна функція викликається через покажчик (або посилання) на базовий клас.

У цьому випадку у процесі виконання програмний код визначає, яку саме версію віртуальної функції необхідно викликати за типом об'єкта, який адресується цим покажчиком.



АБСТРАКТНІ КЛАСИ ТА ІНТЕРФЕЙСИ

АБСТРАКТНИЙ КЛАС - ЦЕ КЛАС, У ЯКОГО НЕ РЕАЛІЗОВАНА ОДНА АБО БІЛЬШЕ ВІРТУАЛЬНА ФУНКЦІЯ.

ІНТЕРФЕЙС - ЦЕ АБСТРАКТНИЙ КЛАС, У ЯКОМУ УСІ ВІРТУАЛЬНІ ФУНКЦІЇ Є ЧИСТИМИ(ТОБТО НЕ МАЮТЬ РЕАЛІЗАЦІЇ В БАЗОВОМУ КЛАСІ).

АБСТРАКТНЫЙ КЛАС

```
1  #pragma once
2  #include <string>
3  #include <vector>
4
5  template <typename T>
6  class IParticleDefinition
7  {
8  protected:
9      std::string m_name;
10     T m_mass;
11     T m_P;
12     T m_E;
13
14 public:
15     virtual std::string getName() { return m_name; }
16     virtual ~IParticleDefinition(){};
17
18     virtual T getE() = 0;
19 };
```

ДОЧІРНІ КЛАСИ

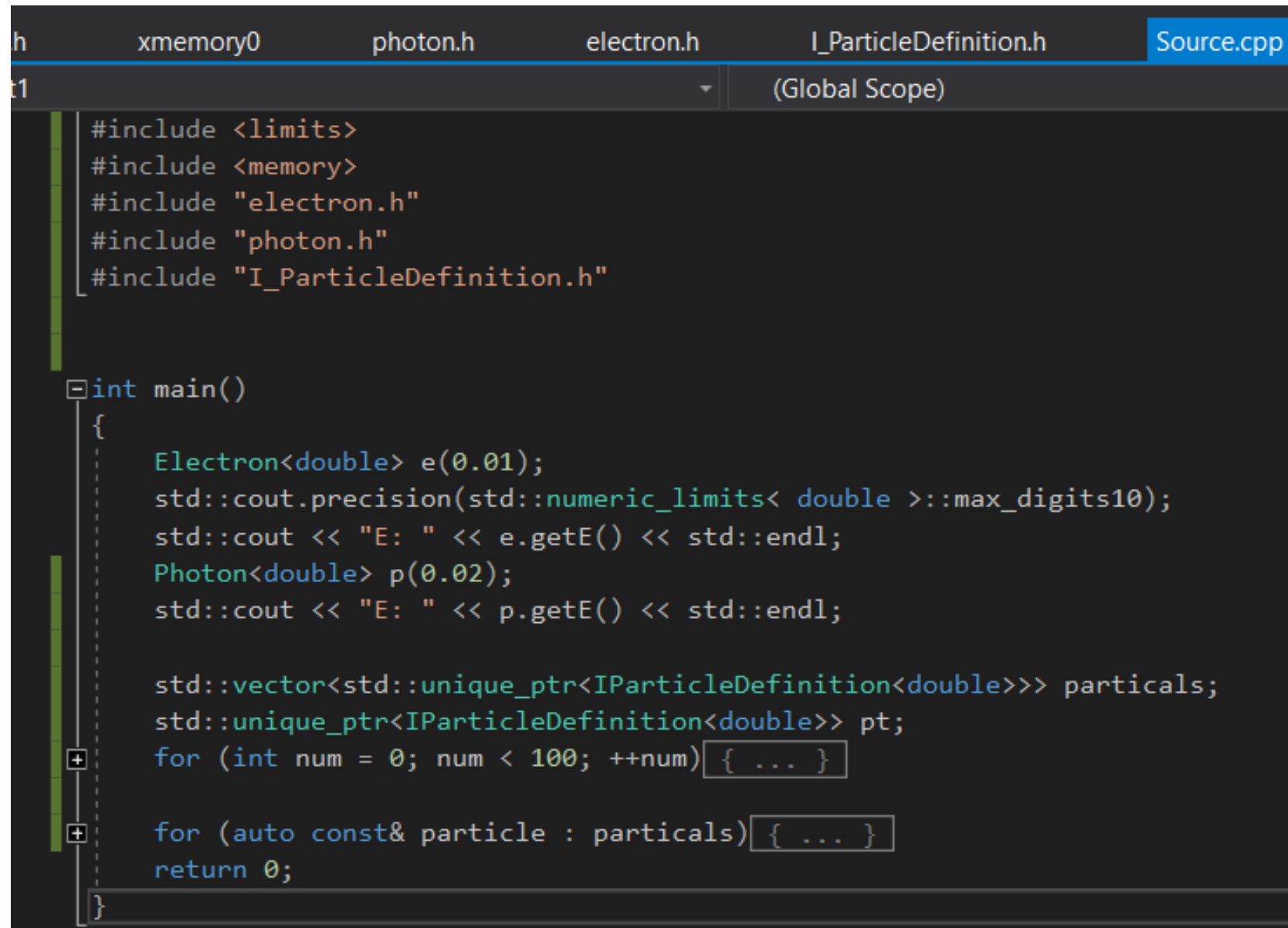
```
xmemory0  photon.h  electron.h  I_ParticleDefinition.h*  Source.cpp
Electron<T>

#pragma once
#include "I_ParticleDefinition.h"
#include "constants.h"
template <typename T>
class Electron : public IParticleDefinition<T>
{
private:
    T m_electric_charge;
    T m_spin;
    std::string m_family;
public:
    Electron(double P) :
        m_electric_charge(-1.), m_spin(0.5), m_family("lepton")
    {
        this->m_name = "electron";
        this->m_mass = 0.511;
        this->m_P = P;
    }
    ~Electron() {}
    T getE()
    {
        return sqrt(this->m_P * this->m_P * CONST::c2 + this->m_mass * this->m_mass * CONST::c2);
    }
};
```

```
xmemory0  photon.h  electron.h  I_ParticleDefinition.h*  Source.cpp
Photon

#pragma once
#include "I_ParticleDefinition.h"
#include "constants.h"
template <typename T>
class Photon : public IParticleDefinition<T>
{
private:
    T m_spin;
    std::string m_family;
public:
    Photon(double P) :
        m_spin(1.0), m_family("bozon")
    {
        this->m_name = "photon";
        this->m_mass = 0.;
        this->m_P = P;
    }
    ~Photon() {}
    T getE()
    {
        return this->m_P * CONST::c;
    }
};
```


ЗАСТОСУВАННЯ



```
h      xmemory0      photon.h      electron.h      I_ParticleDefinition.h      Source.cpp
t1      (Global Scope)

#include <limits>
#include <memory>
#include "electron.h"
#include "photon.h"
#include "I_ParticleDefinition.h"

int main()
{
    Electron<double> e(0.01);
    std::cout.precision(std::numeric_limits< double >::max_digits10);
    std::cout << "E: " << e.getE() << std::endl;
    Photon<double> p(0.02);
    std::cout << "E: " << p.getE() << std::endl;

    std::vector<std::unique_ptr<IParticleDefinition<double>>> particals;
    std::unique_ptr<IParticleDefinition<double>> pt;
    for (int num = 0; num < 100; ++num) { ... }

    for (auto const& particle : particals) { ... }
    return 0;
}
```

ПОЛІМОРФІЗМ

Успадкування надає можливість не тільки довизначити метод, залишений не визначеним у батьківському класі, але й замістити його іншим (*заміщення — overriding*).

Рішення про те, яку саме функцію буде викликано приймається на етапі виконання програми залежно від конкретного типу створеного об'єкту.

Це явище, при якому об'єкти одного типу (підкласи) заміщують об'єкти іншого типу (суперкласу) дістало назву **поліморфізму**.

ПОЛІМОРФІЗМ

```
int main()
{
    std::cout.precision(std::numeric_limits< double >::max_digits10);

    std::vector<std::unique_ptr<IParticleDefinition<double>>> particals;
    std::unique_ptr<IParticleDefinition<double>> pt;
    for (int num = 0; num < 100; ++num)
    {
        if (static_cast<float>(std::rand()) / RAND_MAX < 0.5)
        {
            pt = std::unique_ptr<IParticleDefinition<double>>(new Electron<double>(static_cast<float>(std::rand()) / RAND_MAX));
        }
        else
        {
            pt = std::unique_ptr<IParticleDefinition<double>>(new Photon<double>(static_cast<float>(std::rand()) / RAND_MAX));
        }
        particals.emplace_back(std::move(pt));
    }

    for (auto const& particle : particals)
    {
        double E = particle->getE();
        std::cout << "name: " << particle->getName() << "\tE:" << E << "\n";
    }
    return 0;
}
```

ЗАВДАННЯ

Змінити абстрактний клас `IParticleDefinition` таким чином, щоб від був інтерфейсом.