Наследование и динамический полиморфизм в C++.

Необходимо выполнить 2 задания. Выложите решение на github в репозиторий InheritanceExample.

**Пример 1**. Построить иерархию классов согласно схеме наследования (рис. 1).

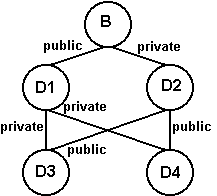


Рисунок 1. Схема классов.

**Решение.**

Описание классов

class B {

int a;

public:

B() { cout << "object of" << typeid(\*this).name() << " was created\n"; }

B(int x) :B() { a = x; }

~B() { cout << "object of" << typeid(\*this).name() << " was destroyed\n"; }

};

class D1 : public B {

int b;

public:

D1(int x, int y) : B(y)

{

b = x;

cout << "object of" << typeid(\*this).name() << " was created\n";

};

~D1() { cout << "object of" << typeid(\*this).name() << " was destroyed\n"; }

};

class D2 : private B {

int c;

public:

D2(int x, int y) : B(y)

{

c = x;

cout << "object of" << typeid(\*this).name() << " was created\n";

};

~D2() { cout << "object of" << typeid(\*this).name() << " was destroyed\n"; }

};

class D3 : private D1, public D2 {

int d;

public:

D3(int x, int y, int z, int i, int j) : D1(y, z), D2(i, j)

{

d = x;

cout << "object of" << typeid(\*this).name() << " was created\n";

}

~D3() { cout << "object of" << typeid(\*this).name() << " was destroyed\n"; }

};

class D4 : public D2, private D1 {

int e;

public:

D4(int x, int y, int z, int i, int j) : D1(y, z), D2(i, j)

{

e = x;

cout << "object of" << typeid(\*this).name() << " was created\n";

}

~D4() { cout << "object of" << typeid(\*this).name() << " was destroyed\n"; }

};

Проверьте последовательность вызовов конструкторов при создании объектов и вызовов деструкторов при уничтожении для объектов. Создайте объекты классов D3 и D4:

D3 temp(100, 200, 300, 400, 500);

D4 temp1(1, 2, 3, 4, 5);

Убедитесь в том, что конструктор класса B вызывается два раза: перед созданием класса D1 и класса В2. Чтобы исправить ситуацию, то есть выполнить вызов только один раз, необходимо сделать наследование виртуальным. Для этого перед модификатором базового класса при описании класса-наследника (для классов D1 и D2) нужно написать virtual. Деструктор так же будет вызван только один раз.

VS позволяет построить диаграмму классов. Диаграмма классов данного проекта (рис. 2).

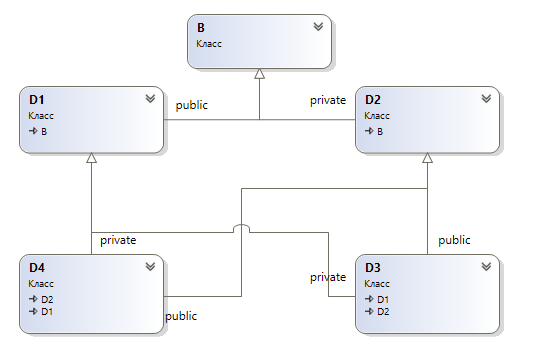


Рисунок 2. Диаграмма классов.

Отношение наследования меду кассами обозначается стрелкой, направленной от наследника к базовому классу. Чтобы добавить диаграмму классов к проекту, необходимо в контекстном меню выбрать: Добавить –> Создать элемент…

Далее в шаблонах Служебных программ выбрать Диаграмму классов. Созданные классы нужно перетащить на поле диаграммы. Связи наследования появляются автоматически.

Если диаграмма классов отсутствует, то нужно установить пакет для работы с ней <https://docs.microsoft.com/ru-ru/visualstudio/ide/class-designer/how-to-add-class-diagrams-to-projects?view=vs-2019>

|  |  |  |
| --- | --- | --- |
| **Задание 1 для самостоятельного выполнения.** | | |
| Постройте иерархию классов согласно схеме наследования, приведенной в варианте задания. Варианты можно найти по ссылке: <http://khpi-iip.mipk.kharkiv.edu/library/pgm/lab2/lab3.html>. Задание выполнить аналогично разобранному выше примеру 1. Построить диаграмму классов. | | |
| 1 вариант – рис. 6  2 вариант – рис. 10  3 вариант – рис. 11  4 вариант – рис. 12 | 5 вариант – рис. 13  6 вариант – рис. 15  7 вариант – рис. 16  8 вариант – рис. 18 | 9 вариант – рис. 19  10 вариант – рис. 20  11 вариант – рис. 21  12 вариант – рис. 24 |

**Пример 2.** Использование виртуальных функций.

// **класс**, описывающий **существо**

class creature

{

private:

string title;

double mass; //kg

protected:

creature() : mass(0.0) {}

creature(const creature& obj): title(obj.title), mass(obj.mass) {}

creature(const string& \_title, double \_mass): title(\_title), mass(\_mass) {}

// деструктор (объявлен как виртуальный)

virtual ~creature() { cout << "creature deleted" << endl << endl; }

// виртуальная функция для вывода информации об объекте

virtual void \_print() const { cout << "title: " << title << ", mass: " << mass; }

public:

// открытая функция для вывода информации об объекте

void print() const

{

cout << typeid(\*this).name() << ": (";

// вызов виртуальной функции

// т.к. функция \_print виртуальная, вызываться она будет

// не только из текущего класса, но и из производных,

// в зависимости от того, для какого объекта осуществляется вызов

\_print();

cout << ")" << endl;

}

};

// **класс animal** наследуется от класса creature

class animal : public creature

{

private:

double speed; //m/s

public:

animal() : creature() {}

animal(const animal& obj): creature(obj), speed(obj.speed) {}

animal(const string& \_title, double \_mass, double \_speed)

: creature(\_title, \_mass), speed(\_speed) {}

~animal() { cout << "animal deleted" << endl; }

protected:

// виртуальная функция \_print переопределяется в производном //классе

void \_print() const

{

creature::\_print();

cout << ", speed: " << speed;

}

};

// **класс bird** наследуется от класса animal

class bird : public animal

{

private:

double topfly; //km

public:

bird() : animal() {}

bird(const bird& obj): animal(obj), topfly(obj.topfly) {}

bird(const string& \_title, double \_mass, double \_speed, double \_topfly)

: animal(\_title, \_mass, \_speed), topfly(\_topfly) {}

~bird() { cout << "bird deleted" << endl; }

protected:

// виртуальная функция \_print переопределяется в классе bird

void \_print() const

{

animal::\_print();

cout << ", topfly: " << topfly;

}

};

// **класс fish** наследуется от класса animal

class fish : public animal

{

private:

double maxdeep; //km

public:

fish() : animal() {}

fish(const fish& obj): animal(obj), maxdeep(obj.maxdeep) {}

fish(const string& \_title, double \_mass, double \_speed, double \_maxdeep)

: animal(\_title, \_mass, \_speed), maxdeep(\_maxdeep) {}

~fish() { cout << "fish deleted" << endl; }

protected:

// виртуальная функция \_print переопределяется в классе fish

void \_print() const

{

animal::\_print();

cout << ", maxdeep: " << maxdeep;

}

};

Протестируйте работу классов на следующем примере

int main()

{

setlocale(0, "Rus");

// создание объектов и вывод информации о них

animal("Паук", 0.003, 0.05).print();

bird("Ворона", 0.3, 10, 0.1).print();

fish("Рыба Молот", 150, 5, 0.5).print();

cout << endl;

return 0;

}

Все животные могут перемещаться: рыбы плавают, звери бегают, змеи ползают, птицы летают. Но, как именно перемещается некое абстрактное животное, сказать нельзя.

Дополните пример следующими функциями.

В класс creature добавьте функцию получения массы существа

double get\_mass() const

{

return mass;

}

В класс animal добавьте функцию получения скорости существа

double get\_speed() const

{

return speed;

}

Добавьте описание нового класса для хищника.

// класс predator наследуется от animal

class predator : public animal

{

protected:

predator() {}

public:

~predator() {}

// чисто виртуальная функция hunt будет определять

// посредством производных классов,

// удастся ли хищнику поохотиться на жертву (obj)

virtual bool hunt(const animal& obj) = 0;

// т.к. hunt чисто виртуальная, класс predator является абстрактным

};

Опишите класс орла.

// используется множественное наследование, т.к. орел – и птица, и хищник

class eagle : public bird, public predator

{

public:

eagle() : bird() {}

eagle(const eagle& obj) : bird(obj) {}

eagle(double \_mass, double \_speed, double \_topfly)

: bird("Орел", \_mass, \_speed, \_topfly) {}

// определение тела функции hunt

// т.к. hunt определена, класс eagle – НЕабстракный

bool hunt(const animal& obj)

{

// функция get\_mass может наследоваться классом eagle из класса animal

// и через класс bird, и через класс predator

// нужно указать, что она наследуется через bird

return obj.get\_mass() < bird::get\_mass()

&& obj.get\_speed() < bird::get\_speed();

}

};

Проверка

int main()

{

setlocale(0, "Rus");

bird raven("Ворона", 0.3, 10, 0.1);

eagle eagle1(1, 100, 1);

fish hammerhead("Рыба-молот", 150, 5, 0.5);

raven.print();

hammerhead.print();

cout << "Eagle vs raven: " << eagle1.hunt(raven) << endl;

cout << "Eagle vs hammerhead: " << eagle1.hunt(hammerhead) << endl;

cout << endl;

return 0;

}

Функцию main можно переписать, вызывая hunt через ссылку на predator:

int main()

{

setlocale(0, "Rus");

bird raven("Ворона", 0.3, 10, 0.1);

fish hammerhead("Рыба-молот", 150, 5, 0.5);

raven.print();

hammerhead.print();

// динамическое создание объекта класса eagle,

// объект запоминается как ссылка на объект класса predator,

// подобное возможно, т.к. eagle является производным для predator

predator &eagle1 = \*new eagle(1, 100, 1);

cout << "Eagle vs raven: " << eagle1.hunt(raven) << endl;

cout << "Eagle vs hammerhead: " << eagle1.hunt(hammerhead) << endl;

cout << endl;

// очищение динамически выделенной памяти

delete &eagle1;

return 0;

}

Виртуальное наследование

В описанном выше примере при вызове функций get\_mass и get\_speed в функции hunt класса eagle можно не указывать область видимости класса bird, а вызвать get\_mass и get\_speed напрямую из animal, применив виртуальное наследование.

Виртуальное наследование является механизмом автоматического разрешения противоречий в случае возникновения неоднозначностей при вызове функций, в случаях, когда класс является наследником нескольких наследников общего базового класса. Для организации виртуального наследования нужно при наследовании перед спецификатором доступа базового класса (или после) указать слово virtual.

Добавьте virtual к базовому классу при описании класса bird. Таким образом, наследники класса bird будут являться виртуальными наследниками animal.

Аналогично, добавьте virtual к базовому классу при описании класса predator. Наследники класса predator будут являться виртуальными наследниками animal.

Класс eagle так же претерпит изменения:

class eagle : public bird, public predator

{

public:

eagle() : bird() {}

eagle(const eagle& obj)

: bird(obj),

// из-за виртуального наследования, для передачи параметров

// в конструктор класса animal, его необходимо вызвать явно

animal(obj) {}

eagle(double \_mass, double \_speed, double \_topfly)

: bird("", 0, 0, \_topfly),

// аналогично

animal("Орел", \_mass, \_speed) {}

bool hunt(const animal& obj)

{

return obj.get\_mass() < get\_mass()

&& obj.get\_speed() < get\_speed();

}

};

Создайте диаграмму классов для данного примера.

|  |
| --- |
| **Задание 2 для самостоятельного выполнения.** |
| Для данной предметной области составить ее программную модель посредством иерархии классов.  Предметная область – компьютерная игра. Предполагаемые классы: объект компьютерной игры, мифическое животное, человекоподобный персонаж, волшебный персонаж, волшебный предмет. Дополнение системы классов приветствуется. |
| Согласно предметной области написать программу, реализующую и демонстрирующую работу иерархии классов с применением механизмов динамического полиморфизма (виртуальных функций) и абстракции (чисто виртуальных функций), а также виртуального наследования.  Примечание: все неабстрактные классы должны содержать:  - конструктор без параметров;  - конструктор копирования;  - параметризированный(е) конструктор(ы);  - деструктор,  - переопределите в классах метод move(), draw() (добавление и переопределение своего метода приветствуется). |