Лекция 7 – Иерархии классов в C++

Кафедра прикладной математики и информатики

12 мая 2020 г.

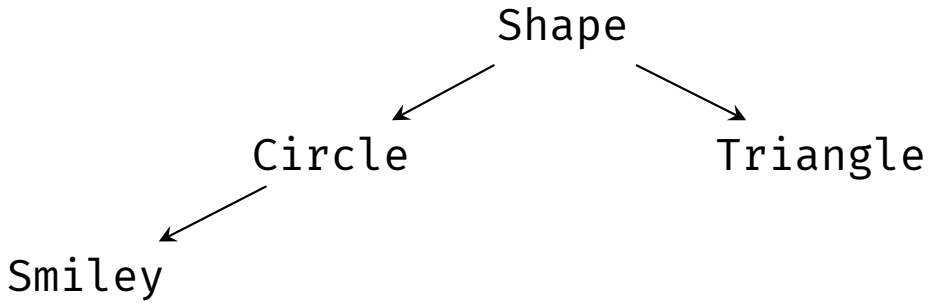
Иерархии классов I

Определение

Иерархия классов – это набор классов, упорядоченных в ориентированном графе, созданной путем наследования (например, public).

Ранее рассмотренный абстрактный класс `Container` был примером простой иерархии.

Мы используем иерархии классов для представления концепций, которые имеют иерархические отношения, такие как «Пожарная машина - это своего рода грузовик, который является своего рода транспортным средством» и «Смайлик - это своего рода круг, который является своего рода форма. В проектах распространены огромные иерархии с сотнями классов, как в глубину так и в ширину.



Иерархии классов II

Стрелки представляют отношения наследования. Например, класс `Circle` является производным от класса `Shape`
Чтобы представить диаграмму в коде, мы должны сначала указать класс, который определяет общие свойства всех фигур:

```
class Shape {
    public:
    virtual Point center() const = 0; // чистая виртуальная
    ↪  функция
    virtual void move(Point to) = 0;
    virtual void draw() const = 0;    // виртуальная функция
    ↪  отрисовки
    virtual void rotate(int angle) = 0;
    virtual ~Shape() {}    // деструктор
    // ...
};
```

Естественно, этот интерфейс является абстрактным классом: что касается представления данных, то ничего (кроме расположения указателя на `vtbl`) не является общим для каждого `Shape`.
Приведя это определение, мы можем написать общие функции, управляющие векторами указателей на фигуры:

Иерархии классов III

```
void rotate_all(vector<Shape*>& v, int angle) // поворот всех
↪   элементов вектора на угол angle
{
    for (auto p : v)
    p->rotate(angle);
}
```

Произвольные классы I

Чтобы определить конкретную форму, мы должны указать, какая это форма и определить ее конкретные свойства (включая ее виртуальные функции):

```
class Circle : public Shape {
    public:
    Circle(Point p, int rad); // конструктор
    Point center() const override
    {
        return x;
    }
    void move(Point to) override
    {
        x = to;
    }
    void draw() const override;
    void rotate(int) override {} // отличный простой алгоритм
    private:
    Point x; // центр
    int r; // радиус
};
```

Произвольные классы II

Пока что в примере Shape and Circle нет ничего нового по сравнению с примером Container и Vector_container, но мы можем построить дальше:

```
class Smiley : public Circle { // используем Circle как родительский
↪   класс
    public:
    Smiley(Point p, int rad) : Circle{p,rad}, mouth{nullptr} { }
    ~Smiley()
    {
        delete mouth;
        for (auto p : eyes)
        delete p;
    }
    void move(Point to) override;
    void draw() const override;
    void rotate(int) override;
    void add_eye(Shape* s)
    {
        eyes.push_back(s);
    }
    void set_mouth(Shape* s);
    virtual void wink(int i); // моргнуть глазом № i
    // ...
    private:
    vector<Shape*> eyes; // обычно 2 глаза
    Shape* mouth;
};
```

Произвольные классы III

Теперь мы можем определить `Smiley::draw()`, используя вызовы родительского класса `Circle` и других членов класса:

```
void Smiley::draw() const
{
    Circle::draw();
    for (auto p : eyes)
    p->draw();
    mouth->draw();
}
```

Обратите внимание на то, как `Smiley` следит за вектором стандартной библиотеки и удаляет их в своем деструкторе. Деструктор `Shape` виртуален, а деструктор `Smiley` переопределяет его. Виртуальный деструктор необходим для абстрактного класса, поскольку объектом производного класса обычно манипулируют через интерфейс, предоставляемый его абстрактным базовым классом. В частности, он может быть удален через указатель на базовый класс. Затем механизм вызова виртуальной функции обеспечивает вызов надлежащего деструктора. Затем этот деструктор неявно вызывает деструкторы его родительского класса и членов.

В этом упрощенном примере задачей программиста является размещение глаз и рта соответствующим образом внутри круга, представляющего лицо. Мы можем добавить поля данных, методы или и то и другое, поскольку мы определяем новый класс путем наследования. Это дает большую гибкость с соответствующими возможностями для путаницы и плохого дизайна.

Польза

Иерархия классов предлагает два вида преимуществ:

- *Наследование интерфейса*: объект производного класса можно использовать везде, где требуется объект базового класса. Таким образом, базовый класс действует как интерфейс для производного класса. Классы `Container` и `Shape` являются примерами. Такие классы часто являются абстрактными классами.
- *Наследование реализации*: базовый класс предоставляет функции или данные, которые упрощают реализацию производных классов. Примеры использования в `Smiley` конструктора `Circle` и `Circle::draw()`. Такие базовые классы часто имеют поля-члены и конструкторы.

Concrete vs base class

Конкретные классы – особенно классы с небольшими представлениями – очень похожи на встроенные типы: мы определяем их как локальные переменные, обращаемся к ним по их именам, копируем их и т.д. Классы в иерархиях классов различны: мы склонны размещать их в куче (динамической памяти) с использованием ключевого слова **new**, и мы получаем к ним доступ через указатели или ссылки.

note

оба типа классов в русском языке тяготеет к переводу ”базовый класс”.

Пример чтения I

Рассмотрим функцию, которая считывает данные, описывающие фигуры, из входного потока и создает соответствующие объекты Shape:

```
enum class Kind { circle, triangle , smiley };
Shape* read_shape(istream& is) // читает описание Shape из istream is
{
    // ... чтение заголовка, чтобы определить Kind k ...
    switch (k) {
        case Kind::circle:
        // read circle data {Point,int} into p and r
        return new Circle{p,r};
        case Kind::triangle:
        // read triangle data {Point,Point,Point} into p1, p2, and p3
        return new Triangle{p1,p2,p3};
        case Kind::smiley:
        // read smiley data {Point,int,Shape,Shape,Shape} into p, r,
        ↪  e1, e2, and m
        Smiley* ps = new Smiley{p,r};
        ps->add_eye(e1);
        ps->add_eye(e2);
        ps->set_mouth(m);
        return ps;
    }
}
```

Пример чтения II

Программа может использовать такое чтение формы следующим образом:

```
void user()
{
    std::vector<Shape*> v;
    while (cin)
    v.push_back(read_shape(cin));
    draw_all(v); //вызов draw() для каждого элемента
    rotate_all(v,45); //вызов rotate(45) для каждого элемента
    for (auto p : v) // не забываем удалять данные из вектора
        delete p;
}
```

Очевидно, что пример упрощен – особенно в отношении обработки ошибок - но он наглядно демонстрирует, что `user()` абсолютно не представляет, с какими формами он манипулирует. Код `user()` может быть скомпилирован один раз, а затем использован для новых фигур, добавленных в программу. Обратите внимание, что нет указателей на фигуры вне `user()`, поэтому `user()` отвечает за их освобождение. Это делается с помощью оператора удаления и критически зависит от виртуального деструктора `Shape`. Поскольку этот деструктор является виртуальным, **`delete`** вызывает деструктор для самого производного класса. Это очень важно, поскольку производный класс может получить все виды ресурсов (например, файловые дескрипторы, блокировки и выходные потоки), которые должны быть освобождены. В этом случае Смайлик удаляет свои глаза и предметы рта. После этого он вызывает деструктор Круга. Объекты конструируются «снизу вверх» конструкторами, а «разрушаются сверху» деструкторами.

Определение класса наследника

Функция `read_shape()` возвращает `Shape*`, чтобы мы могли обрабатывать все объекта типа `Shape` одинаково. Однако что мы можем сделать, если мы хотим использовать функцию-член, которая предоставляется только определенным производным классом, таким как `Smiley::wink()`? Мы можем спросить: «Эта форма – своего рода смайлик?», Используя оператор **`dynamic_cast`**:

```
Shape* ps {read_shape(cin)};
if (Smiley* p = dynamic_cast<Smiley*>(ps)) { // ... указатель
↪  на Smiley? ...
    // ... Smiley; используем
}
else {
    // ... не Smiley, пробуем что-то еще ...
}
```

Если во время выполнения объект, на который указывает аргумент **`dynamic_cast`** (`ps`), не относится к ожидаемому типу (здесь `Smiley`) или к классу, производному от ожидаемого типа, **`dynamic_cast`** возвращает **`nullptr`**.

Исключение при **dynamic_cast**

Мы используем **dynamic_cast** для типа указателя, когда указатель на объект другого производного класса является допустимым аргументом. Затем мы проверяем, является ли результат нулевым. Этот тест часто удобно помещать в инициализацию переменной в условии.

Когда другой тип неприемлем, мы можем просто провести **dynamic_cast** на ссылочный тип. Если объект не относится к ожидаемому типу, **dynamic_cast** генерирует исключение `bad_cast`:

```
Shape* ps {read_shape(cin)};
Smiley& r {dynamic_cast<Smiley&>(*ps)}; // где-то позже
↪  обработаем исключение std::bad_cast
```

`dynamic_cast`

Код чище, когда **`dynamic_cast`** используется со сдержанностью. Если мы можем избежать использования информации о типе, мы можем написать более простой и эффективный код, но иногда информация о типе теряется и должна быть восстановлена. Обычно это происходит, когда мы передаем объект в какую-либо систему, которая принимает интерфейс, заданный базовым классом. Когда эта система позже передаст объект нам. Нам, возможно, придется восстановить исходный тип. Операции, аналогичные **`dynamic_cast`**, известны как ”является своего рода”, ”является экземпляром”.

Ошибки в реализации

Опытные программисты заметят, что мы оставили открытыми три возможности для ошибок:

- Разработчик `Smiley` может не справиться с удалением указателя `mouth`.
- Пользователь `read_shape()` может не справиться с удалением возвращаемого указателя.
- Владелец контейнера содержащего набор `Shape*` может не справиться с удалением объектов, на которые указывает указатель.

В этом смысле указатели на объекты, размещенные в динамической памяти, опасны: «простой старый указатель» не должен использоваться для представления владения ресурсом. Например:

```
void user(int x)
{
    Shape* p = new Circle{Point{0,0},10};
    // ...
    if (x<0) throw Bad_x{}; // возможная утечка
    if (x==0) return; // возможная утечка
    // ...
    delete p;
}
```

Это утечкой памяти, если x не является положительным. Присваивание результата оператора **new** «голому указателю» вызывает проблемы.

Избегаем проблем с памятью I

Одним простым решением таких проблем является использование `unique_ptr` из стандартной библиотеки, а не «голый указатель», когда требуется удаление:

```
class Smiley : public Circle {
    // ...
    private:
    vector<unique_ptr<Shape>> eyes; // обычно два глаза
    unique_ptr<Shape> mouth;
};
```

Это пример простого, общего и эффективного метода управления ресурсами.

Как приятный побочный эффект этого изменения, нам больше не нужно определять деструктор для `Smiley`. Компилятор неявно сгенерирует тот, который выполняет требуемое уничтожение `unique_ptr` в векторе. Код, использующий `unique_ptr`, будет так же эффективен, как и код, использующий голые указатели.

Избегаем проблем с памятью II

Теперь рассмотрим использование `read_shape()`:

```
unique_ptr<Shape> read_shape(istream& is) // читает описание Shape из
↪  istream is
{
    // ... чтение заголовка, чтобы определить Kind k ...
    switch (k) {
        case Kind::circle:
        // read circle data {Point,int} into p and r
        return unique_ptr<Shape>{new Circle{p,r}};
        // ...
    }
}
```

```
void user()
{
    vector<unique_ptr<Shape>> v;
    while (cin)
    v.push_back(read_shape(cin));
    draw_all(v); //вызываем draw() для каждого элемента
    rotate_all(v,45); //вызываем rotate(45) для каждого элемента
} // все Shape неявно разрушены
```

Теперь каждый объект принадлежит `unique_ptr`, который будет удалять объект, когда он больше не нужен, то есть когда его `unique_ptr` выходит из области видимости.

Чтобы работала версия `unique_ptr user()`, нам нужны версии `draw_all()` и `rotate_all()`, которые принимаютvector<unique_ptr<Shape>>.

Рекомедации I

1. Выражай идеи прямо в коде.
2. Конкретный тип (concrete type) – самый простой вид класса. Где это применимо, предпочитайте конкретный тип более сложным классам и простым структурам данных.
3. Используйте конкретные классы для представления простых концепций.
4. Предпочитайте конкретные классы иерархиям классов для компонентов, критичных к производительности.
5. Определяйте конструкторы для инициализации объектов.
6. Объявляйте функцию членом, только если ей нужен прямой доступ к представлению данных класса.
7. Определяйте операторы в первую очередь для имитации обычного использования.
8. Используйте свободные функции для симметричных операторов.
9. Объявляйте функцию-член, которая не изменяет состояние своего объекта **`const`**.
10. Если конструктор получает ресурс, его классу нужен деструктор для освобождения ресурса.
11. Избегайте «голых» операций **`new`** и **`delete`**.
12. Используйте дескрипторы ресурсов и RAII для управления ресурсами.

Рекомедации II

13. Если класс является контейнером, создайте ему конструктор списка инициализаторов.
14. Используйте абстрактные классы в качестве интерфейсов, когда необходимо полное разделение интерфейса и реализации.
15. Получайте доступ к полиморфным объектам через указатели и ссылки.
16. Абстрактный класс обычно не нуждается в конструкторе.
17. Используйте иерархии классов для представления концепций с внутренней иерархической структурой.
18. Класс с виртуальной функцией должен иметь виртуальный деструктор.
19. Используйте **`override`**, чтобы указать переопределение явным в больших иерархиях классов.
20. При разработке иерархии классов следует различать наследование реализации и наследование интерфейса.
21. Используйте **`dynamic_cast`**, где навигация по иерархии классов неизбежна.
22. Используйте **`dynamic_cast`** для ссылочного типа, когда неудача получения требуемого класса является ошибкой.
23. Используйте **`dynamic_cast`** для типа указателя, когда неудача получения требуемого класса является допустимой альтернативой.
24. Используйте `unique_ptr` или `shared_ptr`, чтобы не забыть удалить объекты, созданные с помощью **`new`**.