Наследование и полиморфизм — конспект темы

Введение в наследование

Отношения между классами:

- композиция;
- агрегация,
- зависимость.

В C++ для выражения отношения «является» используют **наследование**.

Механизм **наследования** создаёт новый класс на основе существующего. Новый класс сохраняет данные и поведение родительского класса.

Класс-родитель (класс-предок, родительский класс, предок, суперкласс, родитель) — класс, от которого наследуются другие классы.

Класс-потомок (класс-наследник, дочерний класс, потомок, подкласс) — класс, определённый через наследование.

Базовый класс — класс, находящийся на вершине иерархии наследования, то есть не определённый через наследование. Любой небазовый класс — это класс-потомок.

Иерархия наследования (иерархия классов) — структура, отражающая связи родителей и потомков.

Чтобы объявить класс-наследник в C++, нужно через двоеточие указать имя базового класса:

```
struct Point {
    double x = 0;
    double y = 0;
};

enum class Color { BLACK, RED, GREEN, BLUE };

class Shape {
public:
```

```
    Color GetColor() const;
    void SetColor(Color color);

private:
    Color color_ = Color::BLACK;
};

class Circle : public Shape {
public:
    Point GetCenter() const;
    void SetCenter(Point center);
    double GetRadius() const;
    void SetRadius(double radius);

private:
    Point center_;
    double radius_ = 1.0;
};
```

В коде объявлен класс `Circle`, который наследуется от класса `Shape`. Ключевое слово `public` означает **публичное наследование**, или **наследование интерфейса**. Такой тип наследования выражает отношение «является».

Полиморфизм

Полиморфизм — возможность работать через один интерфейс со множеством реализаций.

Наследование позволяет наследникам:

- использовать методы класса родителя,
- заменять их поведение на собственное.

`virtual` — ключевое слово, помогающее объявить метод класса-родителя виртуальным:

```
class Person {
public:
    ...
    // Реализация виртуального метода может быть переопределена в классах-наследниках
    virtual void Dance() const {
        cout << name_ << " is dancing"s << endl;
    }

private:
    string name_;
```

```
    int age_;
};
```

По умолчанию методы класса в C++ невиртуальные.

Полиморфизм времени выполнения — это когда реализация вызываемого метода выбирается во время работы программы.

Чтобы такой полиморфизм работал корректно, метод родительского класса, который можно переопределить в наследниках, надо объявить виртуальным. В дочерних классах используйте спецификатор `override`, когда метод переопределяет реализацию родителя.

Метод класса, который объявлен виртуальным в родительском классе, остаётся виртуальным и во всех его наследниках.

Наследование — детали

Данные и методы, объявленные со **спецификатором доступа** `protected`, становятся доступны не только самому классу, но и его наследникам.

Защищённый конструктор доступен только классам-наследникам. Поэтому использовать его для создания экземпляра класса не получится.

По умолчанию деструкторы, как и методы класса, невиртуальные. **Тривиальный деструктор** — деструктор, который не выполняет никаких действий.

Деструктор базового класса объявляйте **публичным виртуальным** или **защищённым невиртуальным** в зависимости от того, нужно ли полиморфное удаление классов-наследников через указатель на базовый класс.

Если класс не предназначен для наследования, не объявляйте в нём деструктор или объявите публичным невиртуальным. Сам класс объявите финальным.

Абстрактные классы. Разработка SVG-библиотеки

Абстрактный класс — класс, где объявлен хотя бы один чисто виртуальный метод. Абстрактные классы не предполагают создание своих экземпляров и предназначены для использования в качестве родительских или базовых классов.

Конкретный класс — класс без чисто виртуальных методов.

Абстрактные классы используются при реализации «Шаблонного метода»: базовый класс задаёт количество шагов и их порядок, а подклассы определяют детали реализации этих шагов.

Интерфейсы

Предельный случай абстрактного класса в C++ — это **интерфейс**, класс без членов-данных. Все его методы — чисто виртуальные. Интерфейс задаёт набор методов, которые должны быть у объекта, чтобы с ним взаимодействовать.

Интерфейсы помогают наладить взаимодействие между объектами, у которых минимум информации друг о друге. Это позволяет создавать гибкие архитектуры. Работая с ними, можно заменять одни части программы без модификации других.

Интерфейсы помогают написать модульные тесты. Класс, который зависит от передаваемого извне интерфейса или абстрактного класса, легче протестировать, чем класс, зависящий от конкретных классов:

```
// Интерфейс, от которого зависит тестируемый класс
class Service {
public:
    virtual int DoSomething(int param) = 0;
protected:
    ~Service() = default;
};

// Класс, который требуется протестировать
class MyClass {
public:
    MyClass(Service& service)
        : service_(service) {
    }
    // Метод Run использует service_
    void Run() {
        ...
        int value = service_.DoSomething(42);
        ...
    }
private:
    Service& service_;
};

// Реализация сервиса. Может зависеть от других объектов
class RealService : public Service { ... };
```

Множественное наследование

Множественное наследование — наследование от более чем одного базового класса. Оно позволяет классу реализовывать несколько интерфейсов или использовать функциональность сразу нескольких родительских классов.

Старайтесь избегать **ромбовидного наследования** — оно возникает, когда родители производного класса имеют общего предка.

Runtime-полиморфизм с std::variant

Так выглядит программа, решающая квадратные уравнения. В коде используется наследование:

```
struct Solution {
    virtual ~Solution() = default;
    virtual void Print() const = 0;
};

struct OneRoot : Solution {
    OneRoot(double v)
        : value(v) {
    }
    void Print() const override {
        cout << "One root: "sv << value << endl;
    }
    double value;
};

struct TwoRoots : Solution {
    TwoRoots(double v1, double v2)
        : value1(v1)
        , value2(v2) {
    }
    void Print() const override {
        cout << "Two roots: "sv << value1 << " and "sv << value2 << endl;
    }
    double value1, value2;
};

struct NoRoots : Solution {
    void Print() const override {
        cout << "No roots"sv << endl;
    }
};

// Возвращает корни квадратного уравнения вида ax^2+bx+c=0
unique_ptr<Solution> SolveQuadraticEquation(double a, double b, double c) {
    if (a == 0) {
```

```
        throw invalid_argument("Not a quadratic equation"s);
    }

    const double d = b * b - 4 * a * c;
    if (d > 0) {
        const double sqrt_d = sqrt(d);
        const double dbl_a = 2.0 * a;
        return make_unique<TwoRoots>((-b - sqrt_d) / dbl_a, (-b + sqrt_d) / dbl_a);
    } else if (d == 0) {
        return make_unique<OneRoot>(-b / (2 * a));
    } else {
        return make_unique<NoRoots>();
    }
}

int main() {
    const auto solution = SolveQuadraticEquation(1, 0, -4);
    solution->Print();
}
```

Код громоздкий, а динамическое выделение памяти снижает производительность. Чтобы избежать таких проблем, используют `std::variant`.

`std::variant` — стандартный шаблонный класс, который в один момент времени хранит значение одного из заданных типов либо, в случае ошибки, не содержит значения. Список типов, которые может хранить `variant`, задаётся в его шаблонных параметрах.

Так выглядит программа, решающая квадратные уравнения. В коде используется `std::variant`:

```
struct SolutionPrinter {
    void operator()(monostate) const {
        cout << "No roots"sv << endl;
    }
    void operator()(double root) const {
        cout << "One root: "sv << root << endl;
    }
    void operator()(pair<double, double> roots) const {
        cout << "Two roots: "sv << roots.first << " and "sv << roots.second << endl;
    }
};

int main() {
    const auto solution = SolveQuadraticEquation(1, 0, -4);
    // Функция visit вызовет оператор (), принимающий тот тип,
    // который хранится в variant
```

```
    visit(SolutionPrinter{}, solution);
}
```

Плюсы `variant`:

- Позволяет передавать объекты по значению без динамического выделения памяти.
- Облегчает добавление новых «методов» — вы просто создаёте структуру или класс с перегруженными операторами `()`.
- Не надо вводить базовый класс. `variant` позволяет объединять не связанные друг с другом классы и примитивные типы данных.
- При небольшом количестве типов внутри `variant` затраты на вызов нужной операции Посетителя функцией `visit` могут быть ниже, чем при использовании виртуальных функций.

Минусы `variant`:

- Типы, объединяемые в `variant`, должны быть известны на этапе компиляции. Поэтому использовать его для построения системы подключаемых модулей невозможно. С наследованием система, поддерживающая расширение, просто объявляет интерфейсы, которые модули должны реализовать.
- Трудно добавлять новые типов, так как это требует модификации всех посетителей.
- Неэффективно используется память, когда размеры типов сильно различаются.
- Для каждой операции надо создать отдельного Посетителя. Без должного контроля эти посетители будут разбросаны по различным файлам программы.
- Передача параметров Посетителю требует бóльших усилий по сравнению с обычными методами, так как функция `std::visit` не предоставляет для этого средств.
- При большом количестве типов внутри `variant` затраты на вызов нужной операции Посетителя могут быть выше, чем затраты на вызов виртуального метода.

Динамическое приведение типа

Оператор `static_cast` можно безопасно использовать только для приведения типа вверх по иерархии классов.

Для безопасного приведения типа в пределах иерархии классов служит оператор `dynamic_cast`. В отличие от `static_cast`, он проверяет возможность преобразования, используя информацию о типе объекта. Такое преобразование медленнее, чем `static_cast`, зато безопаснее.

`dynamic_cast` использует служебную информацию, хранящуюся в объекте, чтобы установить его тип и выполнить необходимое преобразование.

При приведении одного типа указателя к указателю другого типа возвращается ненулевой указатель, если преобразование типа возможно. В противном случае — нулевой.

`dynamic_cast` позволяет преобразовывать не только указатели, но и ссылки. Так как в C++ нет понятия нулевой ссылки, при невозможности преобразования выбрасывается исключение `std::bad_cast`:

```
void PlayWithAnimal2(Animal& animal) {
    try {
        // При невозможности приведения ссылки к нужному типу
        // оператор dynaic_cast выбросит исключение std::bad_cast
        Mouse& mouse = dynamic_cast<Mouse&>(animal);
        cout << "Mouse eats cheese"sv << endl;
        mouse.EatCheese();
    } catch (const std::bad_cast&) {
    }

    try {
        Hedgehog& hedgehog = dynamic_cast<Hedgehog&>(animal);
        cout << "Hedgehog sings songs"sv << endl;
        hedgehog.Sing("Jingle Bells"s);
        hedgehog.Sing("Yesterday"s);
    } catch (const std::bad_cast&) {
    }
}
```

`dynamic_cast` позволяет выполнить над объектом действия, зависящие от его типа. Но везде, где возможно, отдавайте предпочтение полиморфизму и виртуальным методам. Это делает код более гибким и не нарушает инкапсуляцию.

Плюсы `dynamic_cast`:

- Полезен, если нет возможности добавить виртуальный метод в базовый класс. Например, из-за того, что это класс сторонней библиотеки.
- `dynamic_cast` позволяет запросить у объекта нужный интерфейс и в случае успеха использовать его.

Другие виды наследования

Приватное наследование или **наследование реализации** происходит, когда класс-наследник унаследовал детали реализации своего родителя, но не его интерфейс. Наследник предоставляет свой набор публичных методов, с которым должны работать клиенты.

Чтобы обозначить приватное наследование, перед именем родительского класса пишут ключевое слово `private`. По умолчанию классы в C++ наследуются приватно, а структуры — публично:

```
class Parent {
public:
    void ParentMethod();
    ...
};
// Класс Child приватно унаследован от Parent
class Child : private Parent {
public:
    void ChildMethod() {
        ParentMethod(); // Классу Child доступны публичные и защищённые методы родителя
    }
};
// Тоже приватное наследование: классы по умолчанию наследуются приватно
class Child1 : Parent {...};

int main() {
    Child child;

    // Не скомпилируется: в классе Child метод ParentMethod стал приватным
    // child.ParentMethod();

    // Класс Child предоставляет свой набор публичных методов
    child.ChildMethod();
}
```

Используйте композицию везде, где возможно, а приватное наследование — там, где это необходимо.

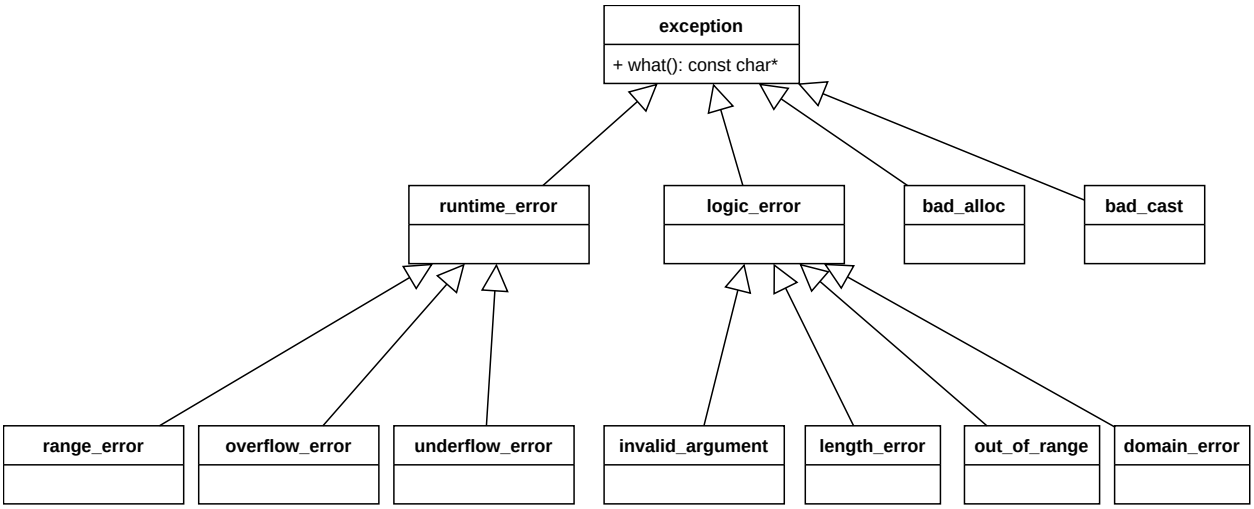
При **защищённом наследовании** публичные и защищённые данные и методы родителя становятся защищёнными в классе-наследнике. Поэтому наследники производного класса тоже смогут ими пользоваться:

```
class Parent {
public:
    void ParentMethod();
    virtual void ParentMethod2();
};

class Child : protected Parent {
public:
    void ChildMethod() {
        ParentMethod();
    }
};

// Внук
class GrandChild : public Child {
public:
    void GrandChildMethod() {
        ParentMethod(); // Внукам также доступны методы Parent
    }
};
```

Защищённое наследование используется редко.

Наследование и исключения

Классы и структуры можно использовать в качестве объектов исключений.



Иерархия классов стандартных исключений в C++.

Функция `stoi` преобразует строки в числа. Она может выбросить исключения двух типов:

- `out_of_range`, если в строке записано число, не вмещающееся в диапазон `int`;
- `invalid_argument`, если в начале строки не записано число.

Классы исключений ловите по ссылке — возможно, константной, — а не по значению. Так вы избежите срезки, когда исключение дочернего класса попадает в обработчик исключения родительского класса.

Обработчик, принимающий исключение родительского класса по значению, будет иметь дело не с оригинальным объектом исключения, а с копией части объекта, который относится к родительскому классу.