Жизненный цикл объекта — конспект темы

Инициализация объектов и конструктор по умолчанию

Чтобы задать начальное состояние объекта, где все его поля согласованы, применяют **конструктор**. Это особый метод класса, он вызывается один раз — в момент создания объекта. Имя конструктора всегда совпадает с именем класса, а тип возвращаемого значения не указывается.

```
class Cat {
public:
    // Это конструктор
    Cat() {
        cout << "Meow"s << endl;
        name_ = "Tom"s;
    }

    string GetName() const {
        return name_;
    }

private:
    string name_;
};

int main() {
    cout << "Start"s << endl;
    Cat cat;
    cout << "The cat's name is "s << cat.GetName() << endl;
}
```

Сначала выполнится код, который предшествует объявлению переменной `cat`. Затем будет создана переменная `cat` и вызван её конструктор. Он проинициализирует поле `name_` и выведет “Meow”. Следом будет выведена кличка кота:

```
Start
Meow
The cat's name is Tom
```

Это конструктор без параметров — **конструктор по умолчанию**. Он вызывается, когда объявляют переменную класса, и инициализирует поля некоторыми значениями по умолчанию. В примере выше конструктор дал коту кличку Tom.

Более простой способ задать значения по умолчанию для полей структур и классов — присвоить полям класса желаемое значение. Сам конструктор по умолчанию в этом случае можно не писать:

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

// Компилятор неявно сгенерирует для класса Cat конструктор по умолчанию,
// инициализирующий поле name_ значением "Tom"s
class Cat {
public:
    const string& GetName() const {
        return name_;
    }

private:
    string name_ = "Tom"s;
};
```

Параметризованный конструктор

Если задать для полей объекта значение по умолчанию, все данные объекта после его создания будут согласованными. Но при использовании класса `Rational` возникают проблемы:

- Код, который выполняет инициализацию, слишком громоздкий.
- Ошибка компиляции: у константного объекта можно вызвать только константные методы.

Чтобы избежать проблем, используют **параметризованный конструктор**. Он принимает один или более параметров:

```
class Rational {
public:
    Rational() = default;

    // Параметризованный конструктор
```

```
    Rational(int numerator, int denominator) {
        numerator_ = numerator;
        denominator_ = denominator;
    }

    int Numerator() const {
        return numerator_;
    }

    int Denominator() const {
        return denominator_;
    }

private:
    int numerator_ = 0;
    int denominator_ = 1;
};
```

Когда в классе объявлен параметризованный конструктор, конструктор по умолчанию для этого класса сгенерирован не будет.

Если конструктор по умолчанию нужен, напишите его сами или попросите об этом компилятор: вместо тела конструктора укажите `=default` и задайте полям значения по умолчанию.

Параметризованный конструктор делает объявление объекта лаконичным и помогает задать начальное состояние константных объектов. Часто в классах делают несколько конструкторов, которые задают разные способы инициализации объектов.

Список инициализации конструктора

Имя класса перед фигурными скобками можно опустить при условии, что код останется понятным:

```
Rational AddRationals(Rational r1, Rational r2) {
    int numerator = r1.Numerator() * r2.Denominator() + r2.Numerator() * r1.Denominator();
    int denominator = r1.Denominator() * r2.Denominator();

    // Компилятор знает, что функция возвращает Rational, и неявно
    // вызывает соответствующий конструктор
    return {numerator, denominator};
    // Эта запись в данном контексте аналогична:
    // return Rational{numerator, denominator};
}

int main() {
```

```
    // Компилятор знает, что функция AddRationals принимает аргументы типа Rational
    // и конструирует дроби 1/6 и 1/3
    Rational sum = AddRationals({1, 6}, {1, 3});
}
```

Композиция — способ писать новые классы и структуры путём включения уже имеющихся.

Структура `RationalPoint` задаёт координаты точки на плоскости, используя композицию рациональных чисел:

```
struct RationalPoint {
    Rational x;
    Rational y;
};
```

Чтобы задать точку, можно передать в качестве координат уже имеющиеся переменные типа `Rational` или указать координаты напрямую:

```
 int main() {
    // Допустимые способы объявления переменной типа Rational
    const Rational x1{7, 8};
    const Rational y1(3, 4);

    // Инициализируем поля структуры при помощи имеющихся переменных
    RationalPoint p1 = {x1, y1};

    // Инициализируем поля структуры явным образом
    RationalPoint p0 = {
        {2, 3},
        {5, 6},
    };

    // Совмещаем разные способы инициализации полей структуры
    const RationalPoint p2{x2, {7, 8}};
}
```

Инициализировать классы сложнее, чем структуры, так как у классов поля приватные и защищены от доступа извне.

В C++ для неявной инициализации полей класса применяют конструктор по умолчанию. Поля, у которых его нет, инициализируют явно внутри списка инициализации конструктора.

Старайтесь всегда задавать значения полей класса в списке инициализации конструктора, а не в его теле. Так программа не будет выполнять двойную работу: вызывать у поля конструктор по умолчанию, потом перезаписывать его значение через присваивание.

Явный и неявный вызов конструктора

Конструктор, который принимает один аргумент, называется **конвертирующий конструктор**. Его можно вызвать неявно:

```
class Rational {
public:
    // Конвертирующий конструктор, создающий дробь из целого числа
    Rational(int numerator) { /* содержимое пропущено */ }
    // прочие конструкторы, методы и поля класса пропущены
};

Rational AddRationals(Rational r1, Rational r2) {
    int numerator = r1.Numerator() * r2.Denominator() + r2.Numerator() * r1.Denominator();
    int denominator = r1.Denominator() * r2.Denominator();

    return {numerator, denominator};
}

int main() {
    Rational sum = AddRationals(Rational{1, 6}, 5);
    // выведет 31/5
    cout << sum.Numerator() << "/"s << sum.GetDenominator() << endl;
}
```

Конвертирующий конструктор позволяет применять функцию `AddRationals` для сложения целых и дробных чисел в любых комбинациях. Функция `AddRationals` способна принимать в качестве аргументов и корректно складывать не только обыкновенные дроби, но и целые числа. Вместо ожидаемого типа `Rational` передано значение типа `int`.

Компилятор видит, что в классе `Rational` есть конструктор, который может сделать из целого числа дробь. Компилятор использует этот конструктор, чтобы создать объект типа `Rational`, а затем передаёт объект в качестве второго параметра функции.

Как правило, неявное преобразование типов нежелательно. Поэтому запрещайте неявный вызов: делайте явным конструктор с одним параметром, используя

ключевое слов `explicit`. Обычно так помечают конструктор с одним параметром, но иногда сто́ит помечать конструкторы с несколькими аргументами.

Деструкторы. Время жизни объектов

Деструктор — специальный метод класса. Он автоматически вызывается в конце жизни объекта, чтобы освободить ресурсы.

Деструктор называется так же, как класс, только перед именем деструктора ставится тильда `~`, а параметры не указываются:

```
class SomeClass {
public:
    // Это деструктор
    ~SomeClass() {
    }
};
```

Писать свой деструктор нужно довольно редко, особенно если пользуетесь классами стандартной библиотеки C++ и других популярных библиотек вроде boost. Он понадобится при разработке классов, использующих низкоуровневые функции операционной системы и компоненты, которые написаны на других языках.