Программирование на языке C++

Вводный курс

Александр Морозов
gelu.speculum@gmail.com

ИТМО, весенний семестр 2020

Содержание

Конструктор

Конструктор – это специальная функция-член класса, служащая для инициализации объектов этого класса.

Имя функции совпадает с именем класса.

При создании объекта класса всегда вызывается один конструктор.

Конструктор нельзя вызвать явно.

У класса может быть произвольное число конструкторов.

Деструктор

Деструктор – это специальная функция-член класса, служащая для выполнения произвольных действий непосредственно перед удалением объекта класса.

Обычно используется для освобождения ресурсов, используемых классом, или иной финализации.

Имя функции – `~ClassName`.

У деструктора не может быть аргументов.

При любом удалении объекта класса, его деструктор вызывается.

У класса может быть только один деструктор.

Деструктор можно вызвать явно.

Особенности конструкторов и деструкторов

- нет возвращаемого значения
- нет cv-квалификаторов
- нет ref-квалификаторов

Пример конструкторов и деструктора

```
 1    struct S {
 2        S(); // 1
 3        S(int); // 2
 4        S(char, double); // 3
 5        S(const S &); // 4
 6        ~S();
 7    };
 8
 9    S get_s();
10
11    int main()
12    {
13        S s1; // 1st constructor is called
14        S s2(10); // 2nd constructor is called
15        {
16            const S & s_ref = get_s();
17            S s3('a', 0); // 3rd constructor is called
18            S s4 = get_s(); // probably, 4th constructor is called
19                            // then S::~S()
20        } // S::~S() for s3, s4 and tmp object is called
21    } // S::~S() for s1 and s2 is called
```

Список инициализации членов

```
 1    struct S {
 2        S(int x) : a(x), b(a*2) {}
 3
 4        int a;
 5        int b = 10;
 6    };
 7
 8    struct SS : S {
 9        SS() : S(20), c(b) {}
10
11        int c;
12    };
```

Варианты инициализации в списке

```
 1    struct A {
 2        A() {}
 3        int a = 10;
 4    };
 5    struct B {
 6        B(int x) : b(x) {} // direct initialization
 7        int b;
 8    };
 9    template <class... Ts>
10    struct C : A, B, Ts... {
11        C(int x, int y, const Ts &... ts)
12          : B{x} // list initialization
13          , Ts(ts)... // pack expansion
14          , c(y) // direct
15      {}
16      int c;
17    };
```

Порядок инициализации объекта класса

1. базовые классы инициализируются в том порядке, в каком они перечислены в списке наследования
2. нестатические поля класса инициализируются в том порядке, в каком они объявлены в определении класса
3. выполняется тело конструктора

Порядок уничтожения объекта класса

1. тело деструктора
2. нестатические поля класса разрушаются в порядке, обратном порядку их объявления в классе
3. для базовых классов деструкторы вызываются в порядке, обратном их порядку в списке наследования

Некорректное завершение выполнения конструктора

```
 1    struct S {
 2        std::string s;
 3        std::vector<int> v;
 4        S * next = nullptr;
 5
 6        S();
 7    };
 8
 9    S::S() : s("Hello, world")
10    {
11        v.resize(100);
12        next = new S(1);
13        throw 1;
14    }
15
16    S s;
```

Делегирование конструкторов

```
1    struct S {
2        S(int, double, char) {...}
3        S() : S(10, 0.5, 'a') {...}
4    };
```

У делегирующего конструктора список инициализации членов должен состоять из одного элемента – вызова целевого конструктора.

Наследование конструкторов

```
1   struct A {
2       A(int, double, char);
3   };
4
5   struct B : A {};
6
7   struct C : A {
8       using A::A;
9   };
10
11  int main() {
12      B b(10, 0.5, 'a'); // error
13      C c(10, 0.5, '\n'); // OK
14  }
```

explicit конструкторы

```
 1    struct A {
 2        A(int);
 3    };
 4
 5    struct B {
 6        explicit B(int);
 7    };
 8
 9    A f(const A &)
10    {
11        return 10; // OK, A::A(int) is called
12    }
13
14    B g(const B &)
15    {
16        return 10; // error
17        return B(10); // OK, B::B(int) is called
18    }
19
20    int main()
21    {
22        f(10); // OK, A::A(int) is called
23        g(10); // error
24        g(B(10)); // OK B::B(int) is called
25    }
```

Конструкторы приведения типа

Любые не-`explicit` конструкторы.

Могут участвовать в последовательности неявного приведения типа.

```
1   struct A {
2       A(int);
3   };
4
5   void f(A);
6
7   f(10);
```

Удалённые и автоматически создаваемые методы

Некоторые методы могут быть созданы автоматически – как неявно, так и явно.

Запретить неявную автоматическую генерацию можно “удалив” метод.

```
1   struct A {}; // A::A() is implicitly-defined
2
3   struct B {
4       B() = default; // B::B() implicit definition is
            ↪ forced
5   };
6
7   struct C {
8       C() = delete; // C::C() is not defined
9   };
```

Некоторые методы могут быть удалены неявно.

Методы, создаваемые автоматически

- конструктор по умолчанию
- конструктор копирования
- конструктор перемещения
- деструктор
- операторы присваивания

Конструктор по умолчанию

```
1   struct S {
2       S(); // default constructor
3   };
4
5   S s; // default constructor is called
6
7   struct A {
8       A();
9   };
10  struct B : A {
11      B();
12  };
13
14  B b; // A::A() and B::B() are called
```

Автоматическое создание конструктора по умолчанию

Если у класса нет явно определенных конструкторов, то конструктор по умолчанию создаётся автоматически и эквивалентен явно определенному конструктору без списка инициализации и с пустым телом:

```
1   struct A {
2   };
3
4   struct B {
5     B() {}
6   };
```

Удалённый конструктор по умолчанию

- наличие ссылочного поля без инициализатора
- наличие не статического константного поля без явно определённого конструктора по умолчанию или инициализатора
- наличие не статического поля без инициализатора, имеющего удалённый либо недоступный конструктор по умолчанию
- наличие прямого или виртуального предка с удалённым либо недоступным конструктором по умолчанию
- наличие прямого или виртуального предка с удалённым либо недоступным деструктором

Удалённый деструктор

```
1    struct S {
2        ~S() = delete;
3    };
4
5    static S ss; // error
6    S s; // error
7    S * ps = new S; // error
8    S * ps = new (ptr) S(); // OK
```

Содержание

Конструктор копирования

Это не шаблонный конструктор, принимающий первым аргументом lvalue ссылку на объект того же типа, что и класс конструктора, который может быть вызван с одним аргументом.

Конструктор копирования вызывается всегда, когда объект класса инициализируется из lvalue выражения того же типа (класса).

Пример конструкторов копирования

```
1    struct S {
2        S(int);
3        S(const S &); // copy constructor 1
4        S(S &); // copy constructor 2
5    };
6
7    S f(S s) {
8        return s; // copy constructor ? is called
9    }
10
11   const S s1(10);
12   S s2 = s1, s3(s1); // copy constructor ? is called
13   f(S(10)); // copy constructor ? is called
```

Автоматическое создание конструктора копирования

- `C(const C &)`, если
  - все прямые или виртуальные предки имеют конструктор копирования `B(const B &)`
  - все не статические поля имеют конструктор копирования `M(const M &)`
- `C(C &)` в противном случае

Такой конструктор выполняет копирование всех подобъектов создаваемого объекта, в обычном порядке инициализации.

Удалённый конструктор копирования

- наличие не копируемого не статического поля
- наличие не копируемого прямого или виртуального предка
- наличие прямого или виртуального предка с удалённым или недоступным деструктором
- наличие rvalue ссылочного не статического поля

Наличие пользовательского конструктора перемещения или перемещающего оператора присваивания $\equiv$ отсутствие автоматически созданного конструктора копирования.

Копирующий оператор присваивания

Это не шаблонный оператор присваивания, принимающий ровно один аргумент, имеющий тип `T`, `T&` или `const T&` (если `T` – это данный класс).

```
1   struct S {
2       S(const S &);
3       S & operator = (const S &);
4   };
5
6   S s1, s2 = s1; // copy constructor is called
7   s1 = s2; // copy assignment is called
```

Предполагаемое поведение такого оператора – копирование значения одного объекта данного класса в другой объект этого же класса.

Автоматическое создание копирующего оператора присваивания

- `C & operator = (const C &)`, если
  - все прямые или виртуальные предки имеют копирующий оператор присваивания `B & operator = (const B &)`
  - все не статические поля имеют копирующий оператор присваивания `M & operator = (const M &)`
- `C & operator = (C &)` в противном случае

Такой оператор выполняет копирующее присваивание всех подобъектов левого объекта из соответствующих подобъектов правого, в обычном порядке инициализации.

Удалённый копирующий оператор присваивания

- наличие не присваиваемого не статического поля
- наличие не присваиваемого прямого или виртуального предка
- наличие ссылочного не статического поля
- наличие константного не статического поля, чей тип не является классом

Наличие пользовательского конструктора перемещения или перемещающего оператора присваивания $\equiv$ отсутствие автоматически созданного копирующего оператора присваивания.

Конструктор перемещения

Это не шаблонный конструктор, принимающий первым аргументом rvalue ссылку на объект того же типа, что и класс конструктора, может быть вызван с одним аргументом.

Конструктор перемещения вызывается всегда, когда объект класса инициализируется из xvalue выражения того же типа (класса).

Предназначение перемещающего конструктора – забрать содержимое у аргумента - временного объекта и, по возможности, избежать затрат на копирование этого содержимого.
При этом аргумент должен остаться в валидном, но не обязательно определённом, состоянии.

Пример конструкторов копирования и перемещения

```
1    struct S {
2        S(const S &); // copy constructor
3        S(S &&); // move constructor
4    };
5
6    void f(S s);
7
8    int main() {
9      S s1;
10     f(s1); // copy constructor is called
11     f(S{}); // move constructor is called
12     f(std::move(s1)); // move constructor is called
13   }
```

Автоматическое создание конструктора перемещения

Создаётся, если

- нет ни одного пользовательского конструктора перемещения
- нет ни одного пользовательского конструктора копирования
- нет ни одного пользовательского оператора присваивания (копирующего или перемещающего)
- нет пользовательского деструктора

Такой конструктор выполняет инициализацию всех подобъектов создаваемого объекта, в обычном порядке инициализации, из xvalue выражения соответствующего подообъекта объекта-донора.

Удалённый конструктор перемещения

- наличие не статического поля, для которого нельзя вызвать конструктор перемещения
- наличие прямого или виртуального предка, для которого нельзя вызвать конструктор перемещения
- наличие прямого или виртуального предка, для которого нельзя вызвать деструктор

Удалённый конструктор перемещения не участвует в разрешении перегрузки.

```
1    struct S {
2        S(const S &);
3        S(S &&) = delete;
4    };
5
6    S get_s();
7
8    S s = get_s(); // OK
```

Перемещающий оператор присваивания

Это не шаблонный оператор присваивания, принимающий ровно один аргумент, имеющий тип `T&&` или `const T&&`.

```
1    struct S {
2        S & operator = (S &&);
3    };
4
5    S s1, s2;
6    s2 = S(); // move assignment is called
7    s1 = std::move(s2); // move assignment is called
```

Предполагаемое поведение перемещающего оператор присваивания – перемещение значения одного объекта данного класса в другой объект данного класса.

Автоматическое создание перемещающего оператора присваивания

Создаётся, если

- нет ни одного пользовательского конструктора перемещения
- нет ни одного пользовательского конструктора копирования
- нет ни одного пользовательского копирующего оператора присваивания
- нет пользовательского деструктора

Такой оператор выполняет перемещающее присваивание всех подобъектов левого объекта из соответствующих подобъектов правого, в обычном порядке инициализации.

Удалённый перемещающий оператор присваивания

- наличие не статического константного поля
- наличие не статического ссылочного поля
- наличие не статического поля, для которого нельзя вызвать перемещающее присваивание
- наличие прямого или виртуального предка, для которого нельзя вызвать перемещающее присваивание

Удалённый конструктор перемещения не участвует в разрешении перегрузки.

```
1    struct S {
2        S(const S &);
3        S(S &&) = delete;
4    };
5
6    S get_s();
7
8    S s = get_s(); // OK
```

Содержание

Материализация

prvalue $\rightarrow$xvalue

```
1    T f();
2
3    int a = 1 + 2;
4    int && b = 1 + 2;
5    T x = f();
6    const M & m = f().member;
```

Copy elision

Отсутствие материализации:

- инструкция `return`, когда её операнд – prvalue того же типа, что и тип возвращаемого значения функции
- инициализация переменной, когда выражение инициализации – prvalue того же типа

```
1    T f() {
2      return T();
3    }
4    T x = T(T(T(f()))); // only default constructor of T is
         ↪ called
```

NRVO

Named return value optimization: если в инструкции `return` операнд обозначает локальную переменную, которая:

- не является параметром функции
- объект которой не является `volatile`
- объект которой имеет автоматический тип размещения
- тип объекта которой совпадает с типом возвращаемого значения функции

То компилятор *может* не генерировать код вызова конструктора копирования/перемещения, как если бы это была copy elision.

```
1   T f() {
2     T t;
3     return t;
4   }
5   T x = f(); // only default constructor of T is called
```