Лекция 08 — Инициализация и пространства имён

Содержание

”Универсальная инициализация в C++11 не является совсем универсальной, но это почти так.”
— FAQ ISO C++

Инициализация

Общие представления

Создание переменной — одно из самых фундаментальных и вместе с тем простых действий, которые только можно сделать в языке программирования. Тем не менее, C++ и здесь всех опередил: общеизвестно, что семантика инициализации — одна из наиболее сложных его частей. Соответствующие механики меняются от стандарта к стандарту. Значительная их часть была унаследована от C; какие-то концепции были привнесены вместе с C++11, перекрыты в C++17 и вновь изменены в C++20.

Далее будет предпринята попытка осознать и структурировать сколь-нибудь значительную долю информации, которую человечеству удалось выяснить об устройстве этого замечательного явления.

Методы инициализации

Инициализация нулём

Хотя инициализация нулём не является способом явно проинициализировать объект, она будет в дальнейшем возникать как один из этапов некоторых из рассматриваемых методов инициализации.

Она имеет следующий эффект:

- если объект имеет скалярный тип, производится неявное приведение `0` к этому типу;
- если объект имеет тип `union`:
  - все биты выравнивания становятся равны 0;
  - первый нестатический член объединения инициализируется нулём;
- если объект имеет классовый тип:
  - все биты выравнивания становятся равны 0;
  - каждый нестатический член инициализируется нулём;
  - каждый подобъект базового класса инициализируется нулём (с некоторой поправкой на виртуальность);
- если объект представляет собой C-style массив, каждый его элемент инициализируется нулём;
- если объект имеет ссылочный тип, ничего не происходит.

Инициализация по умолчанию

```
int x;
```

Инициализация по умолчанию (*default initialization*) — первый метод инициализации, который был унаследован от языка C. Она происходит, когда объект был объявлен, но не был определён; по сути дела, это не является инициализацией вообще — объект просто не инициализируется, и всё. Поведение, возникающее при обращении к переменной, инициализированной по умолчанию, не определено.

Копирующая инициализация

```
int x = 238;
```

Копирующая инициализация (*copy initialization*) также была унаследована от языка C.

Она происходит в следующих случаях:

- переменная объявлена, и через знак равенства указано её значение;
- переменная передана в функцию по значению;
- переменная возвращена из функции по значению.

Использование знака «=» может создать впечатление, что объекту присваивается некоторое значение. С точки зрения стандарта это не так, хотя он и допускает такие оптимизации для некоторых стандартных типов. Формально, при копирующей инициализации создаётся новый временный объект, его данные копируются по нужному адресу, после чего он удаляется.

Агрегатная инициализация

```
int arr[4] = {3, 1, 0, 4};
```

Агрегатная инициализация (*aggregate initialization*) — это третий унаследованный от языка C метод инициализации. Она происходит, когда *агрегат* инициализируется рядом значений, перечисленных в фигурных скобках, причём начиная с C++11 знак равенства при этом можно опускать.

Агрегатом считается:

1. C-style массив;
2. экземпляр класса, не имеющего: [1]
   - объявленных или унаследованных конструкторов
   - приватных или защищённых нестатических полей
   - базовых классов
   - виртуальных функций-членов

На самом деле, агрегатная инициализация — это всего-навсего поэлементная копирующая инициализация, поэтому, например, её нельзя использовать при работе с агрегатами, элементы которых имеют только `explicit` конструкторы:

```
struct S {
    explicit S(int x);
};

int main() {
    S arr[4] = {3, 1, 0, 4}; // CE!
}
```

Важное свойство агрегатной инициализации заключается в том, что если не описать какие-то из значений, соответствующие переменные автоматически инициализируются нулём:

```
int arr[5]{1, 2, 3}; // arr = {1, 2, 3, 0, 0}
```

При работе с подагрегатами, то есть вложенными агрегатными классами, можно использовать вложенные фигурные скобки:

```
struct S {
    int x;
    int y;
};

struct Soo {
    S x;
    int y;
};

int main() {
    Soo soo = {{1, 2}, 1};
}
```

При этом их также можно опускать — тогда компилятор будет рекурсивно обходить все подагрегаты и последовательно присваивать им соответствующие значения:

```
struct S {
    int x;
    int y;
};

struct Soo {
    S x;
    int y;
};
```

```
int main() {
    Soo soo = {1, 2}; // soo = {{1, 2}, 0}
}
```

Начиная с C++11, допускается инициализация членов по умолчанию (*default member initialization*, *DMI*):

```
struct Soo {
    int x = 5;
    int y = 2;
}
```

Настоятельно рекомендуется использовать её всегда, когда это допустимо, то есть не допускать объявления неинициализированных полей классов и структур.

Статическая инициализация

> ```
> static int x = 3;
> static int y; // y == 0
> ```

Последним способом инициализации, унаследованным от языка C, является статическая инициализация (*static initialization*). Она достаточно приятна тем, что статические переменные всегда инициализируются — либо константой, либо нулём.

Проблема может возникнуть, когда переменная инициализируется другим объектом:

```
static int x = 3;
static int y = x;
```

Это поведение не определено, поскольку нет чёткого порядка, в котором эти объекты будут инициализироваться.

Прямая инициализация

> ```
> int x(3);
> ```

Прямая инициализация (*direct initialization*) была введена в C++98 и практически являет собой не что иное, как вызов конструктора. Начиная с C++20 допускается также прямая инициализация агрегатных типов, хотя они и не имеют конструктора (поэтому по сути это всё та же агрегатная инициализация):

```
struct Soo {
    int x;
    int y;
}

int main() {
    Soo soo(1, 2); // soo = {1, 2}
}
```

Несмотря на то, что этот метод существует столько, сколько существует сам C++, у него всё ещё есть серьёзный недостаток. Рассмотрим следующий код:

```
struct Soo {
    Soo() {
        std::cout << "Soo()\n";
    }
}

int main() {
    Soo soo();
}
```

Кажется, что мы объявили экземпляр `soo` структуры `Soo` и вызвали конструктор без аргументов. Но — сюрприз! — компилятор интерпретировал запись `Soo soo()` как объявление функции `soo()`, возвращающей значение типа `Soo`. Такой код, разумеется, скомпилируется, однако вряд ли его поведение совпадёт с тем, которое мы ожидали — эта проблема также называется *most vexing parse*, и это больно. К счастью, большинство компиляторов предупреждают об этом, однако это неприятное обстоятельство всё равно необходимо иметь в виду.

Инициализация значением

> ```
> int x = int()
> ```

Инициализация значением (*value initialization*) была введена в C++03. Она возникает при указании имени типа с пустыми круглыми скобками. Если существует определённый пользователем конструктор по умолчанию, инициализация значением вызывает этот конструктор, в противном случае происходит инициализация нулём.

Универсальная инициализация

```
std::vector<int> vec_1{3, 1, 0, 4} // direct list-initialization
std::vector<int> vec_2 = {3, 1, 0, 4} // copy list-initialization
```

Универсальная инициализация (*uniform initialization*; она же *инициализация списком*, *list initialization*) — одно из ключевых нововведений C++11. Комитетом предполагалось, что новый синтаксис с фигурными скобками будет универсальным для всех классов, а также лишённым проблемы most vexing parse. Инициализация списком бывает двух типов: прямая и копирующая.

Используемый при инициализации список называется *bracded-init-list*. Важно, что он не является объектом и не имеет типа, хотя он может быть неявно преобразован в `std::initializer_list`, и если в классе определен конструктор, принимающий этот тип, то при инициализации списком будет вызван именно он:

```
struct Soo {
    Soo(std::initializer_list<int> init_list) {
        std::cout << "Soo(std::initializer_list<int> init_list)\n";
    }
};

int main() {
    Soo soo = {1, 2, 3, 4}; // Soo(std::initializer_list<int> init_list)
}
```

Начиная с C++20, при инициализации агрегатных классов в списках инициализации можно явно указывать поля, которым будут присваиваться значения:

```
struct Soo {
    int x;
    int y;
    int z;
};
```

```
int main() {
    Soo soo = {.x = 2, .y = 3, .z = 8};
}
```

Эта синтаксическая конструкция называется *designated initializer* (хорошего перевода найдено не было, поэтому будем называть это *назначенной* инициализацией).

> **Важно!**
>
> Порядок перечисления полей при инициализации списком должен совпадать с порядком их объявления в классе, поскольку именно в таком порядке они будут инициализироваться вне зависимости от того, как они были указаны в списке инициализации.

Глобально инициализация списком работает примерно следующим образом: [2]

- если фигурные скобки пусты, выполняется инициализация по значению;
- если фигурные скобки содержат единственное значение того же типа, выполняется прямая или копирующая инициализация в зависимости от синтаксиса;
- для встроенных типов выполняется прямая или копирующая инициализация в зависимости от синтаксиса;
- для агрегатных типов выполняется агрегатная инициализация;
- для классов:
  - если объявлен конструктор от `std::initializer_list`, он вызывается;
  - иначе начинается процесс разрешения перегрузки с множеством переданных в списке инициализации аргументов в качестве аргументов конструктора.

> **⚠ Важно!**
>
> Крайне важно осознавать, что при инициализации списком объектов пользовательских классов компилятор ВСЕГДА предпочтёт конструктор от `std::initializer_list`, если таковой найдётся. Пренебрежение таким поведением — наиболее частая причина проблем, связанных с этим методом.

Примечательное свойство инициализации списком заключается в том, что она не допускает сужающих преобразований:

```
struct Soo {
    int x;
};
```

```
int main() {
    Soo soo{2.0}; // CE!
}
```

Как и при агрегатной инициализации, при инициализации списком мы можем вкладывать фигурные скобки:

```
std::map<int, std::string> mapa = {{238, "tail"}, {30, "orange"}};
```

Увлечение этой механикой, впрочем, может приводить к совершенно абсурдным и практически неразрешимым ситуациям[3], поэтому вложенными скобками лучше не злоупотреблять.

> **Важно!**
>
> Универсальная инициализация, в отличие от агрегатной, не поддерживает пропуск скобок.

Передача и возврат braced-init-list также является копирующей инициализацией списка:

```
struct Soo {
    int x;
    int y;
};

Soo Foo(Soo soo) {
    return {3, 0};
}

int main() {
    Soo({0, 4});
}
```

Существование универсальной инициализации часто может привести к неожиданным последствиям. Рассмотрим следующий пример:

```
std::vector<int> vec_1(3, 0);
std::vector<int> vec_2{3, 0};
```

В первом случае выполнится прямая инициализация, то есть вызовется конструктор вектора, принимающий первым аргументом размер, а вторым — значение элемента. Как результат, `vec_1` будет хранить `{0, 0, 0}`.
Во втором случае выполнится инициализация списком, и вызовется конструктор от `std::initializer_list`, принимающий список элементов, и `vec_2` будет хранить `{3, 0}`.

> **Важно!**
>
> Необходимо различать, какой метод инициализации используется в конкретном определении и как именно она разрешается, поскольку это оказывает критическое влияние на логику работы программы.

Напутствие

Итак, было рассмотрено 7 основных способов проинициализировать переменную в языке C++ (при этом следует понимать, что выше было опущено множество тонкостей, связанных, например, с семантикой шаблонов, `auto`, конкретными деталями разрешения и проч.)

Конечно, теперь возникает естественное желание сформулировать некоторый свод принципов и правил, которые хоть немного облегчили бы нам жизнь при работе с этим безобразием. К сожалению, автор лишён достаточного опыта программирования на C++, чтобы разбрасываться советами самостоятельно или валидировать чужие, однако несколько самых простых идей, которых стоит придерживаться, всё же будет приведено.

1. Не следует использовать инициализацию по умолчанию, в том числе для полей классов и структур. Следует всегда использовать DMI.
2. Для простых типов следует использовать инициализацию копированием.
3. Для агрегатов следует использовать агрегатную инициализацию.
4. Для классов следует предпочитать использование прямой инициализации, если это допустимо. Следует избегать написания классов, в работе с которыми возникает неоднозначность при вызове `{}` и `()`.

Мнения здесь, впрочем, разнятся: среди разработчиков есть приверженцы идеи универсальной инициализации, при которой фигурные скобки используются как можно чаще; есть её противники, утверждающие (следует заметить, очень небезосновательно), что инициализация списком привнесла в язык только больше проблем и неоднозначностей.

Единственная рекомендация в этой ситуации — самостоятельно методом проб и ошибок выяснить, какой подход более удобен и приятен лично читателю.

Пространства имён

Общие представления

Из соображений организации кодовой базы и предотвращения конфликтов имён в больших проектах в C++ существует концепция **пространств имён** (*namespaces*).

Каждое пространство имён предоставляет изолированный блок кода, все объявления в котором помещаются в соответствующее поле видимости:

```
namespace noo {
    int x = 238;
}

int main() {
    std::cout << x; // CE: identifier x in undefined
}
```

Объявления, сделанные вне пользовательских пространств имён, относятся к *глобальному пространству имён*.

Объявление пространств имён

Имена пространств имён не обязаны быть уникальными. В случае, если они совпадают, все определения происходят в одном поле видимости:

```
namespace noo {
    int x = 238;
}

namespace noo {
    int y = 30;
}

// то же самое, что и

namespace noo {
    int x = 238;
```

```
    int y = 30;
}
```

Это свойство крайне важно при работе, например, с заголовочными файлами, поскольку позволяет программисту доопределять ранее объявленные функции и классовые методы. Первое объявление при этом считается оригинальным, а все последующие — *расширяющими* (*extension-namespace-definition*).

Более того, пространство имён может не иметь имени вообще:

```
namespace {
    int x = 238;
}
```

С точки зрения компилятора это объявление эквивалентно созданию пространства имён с уникальным именем и последующему вызову директивы `using`.

Пространству имён также можно объявить *псевдоним*, т.е. другое имя, по которому в дальнейшем к нему можно обращаться так же, как и по исходному:

```
namespace fs = std::filesystem;

int main() {
    fs::path path = "bin/archive.haf";
}
```

Доступ к элементам пространств имён

Чтобы получить доступ к элементам пространства имён за его пределами, необходимо использовать оператор `::`:

```
namespace noo {
    int x = 238;
}

int main() {
    std::cout << noo::x; // OK!
}
```

Процесс, происходящий при указании оператора `::`, называется *qualified name lookup*, и в общих чертах он заключается в просмотре всех деклараций в соответствующем пространстве имён.

Если при этом не указывать никакого имени слева от `::`, будет просмотрено только глобальное пространство имён, что делает доступ к нему возможным даже несмотря на то, что оно не имеет имени:

```
int x = 238;

namespace noo {
    int x = 239;

    void Foo() {
        std::cout << ::x;
    }
}

int main() {
    noo::Foo(); // output: 238
}
```

Так, помимо того, что пространства имён служат мощным инструментом организации, они ещё и несут информацию о том, откуда была вызвана функция, что положительно сказывается на читаемости кода.

Если нам часто приходится обращаться к объектам из одного неймспейса, мы можем предоставить себе возможность опускать спецификатор, использовав так называемую *using-декларацию*:

```
using std::cout;

int main() {
    cout << "itmo\n"; // то же, что и std::cout
}
```

Её также допустимо объявлять внутри функций:

```
int main() {
    using std::cout;
    cout << "Hello, World!\n";
}
```

Важно, что при использовании using-деклараций никакие дальнейшие расширения неймспейса, к которому произошло обращение, не будут учтены:

```cpp
namespace noo {
    void Foo() {
        std::cout << "Initial declaration\n";
    }
}

using noo::Foo;

namespace noo {
    void Foo(int x) {
        std::cout << "Extended declaration\n";
    }
}

int main() {
    Foo(); // OK!
    Foo(238); // CE: no matching function for call to 'Foo'
}
```

Когда в коде встречается using-декларация, в текущем пространстве имён начинает существовать объект, который мы привнесли. Таким образом, после неё мы не можем определять одноимённые объекты:

```cpp
namespace noo {
    int x = 238;
}

using noo::x;

float x = 239; // CE: declaration conflicts with target of using
               //     declaration already in scope
```

Мы также можем привнести в текущее пространство имён другое целиком — это будет называться *using-директивой*:

```cpp
using namespace std;

int main() {
    cout << "Hello, World!\n"; // то же, что и std::cout
}
```

В отличие от using-декларации, она поддерживает все дальнейшие расширения привнесённого пространства имён.

Хотя using-декларации и директивы могут показаться достаточно удобным инструментом, на практике они не рекомендуются к использованию, поскольку могут вновь спровоцировать проблему конфликта имён и усложнить чтение кода.

Вложенные и `inline` пространства имён

Пространства имён можно вкладывать друг в друга:

```
namespace noo {
    namespace nooo {
        int x = 238;
    }
}
```

Вложенные пространства имён вполне ожидаемо отделены друг от друга, то есть в этом примере `int x` не является членом пространства имён `noo`. Если мы захотим обратиться к `x`, это можно будет сделать достаточно естественным образом:

```
int main() {
    std::cout << noo::nooo::x;
}
```

Однако если при объявлении вложенного пространства имён указать квалификатор `inline`, ситуация изменится:

```
namespace noo {
    namespace nooo {
        int x = 238;
    }
}

int main() {
    std::cout << noo::x;
}
```

Объявления в `inline`-неймспейсах будут восприниматься так же, как если бы они были частью объемлющего пространства имён. Так, имя вложенных `inline`-неймспейсов можно опускать при обращении к их элементам, а директива `using`, включающая объемлющее пространство имён, включит и любое вложенное `inline` пространство имён.
Примечательно, что изменять элементы `inline`-неймспейсов непосредственно из объемлющих пространств имён нельзя:

```cpp
inline namespace noo {
    int x = 238;
}

x = 239; // CE!
```

Источники

1. А. П. Хвастунов — Лекции по основам программирования на C++, 1 семестр, 2025
2. Timur Doumler — «Инициализация в современном C++» (перевод) (https://habr.com/ru/companies/jugru/articles/469465/)
3. cppreference.com — Initialization (https://en.cppreference.com/w/cpp/language/initialization.html)
4. cppreference.com — Zero initialization (https://en.cppreference.com/w/cpp/language/zero_initialization.html)
5. cppreference.com — Aggregate initialization (https://en.cppreference.com/w/cpp/language/aggregate_initialization.html)
6. cppreference.com — List initialization (https://en.cppreference.com/w/cpp/language/initialization.html)
7. Barry Rezvin — Uniform initializiation isn't (https://medium.com/@barryrevzin/uniform-initialization-isnt-82533d3b9c11)
8. cppreference.com — Namespaces (https://en.cppreference.com/w/cpp/language/namespace.html)

---

1. Приведенные критерии намеренно упрощены. Для более формального определения см. Aggregate initialization — https://en.cppreference.com/w/cpp/language/aggregate_initialization.html ↩
2. Этот алгоритм достаточно точно отражает реальное положение вещей, хотя, конечно, в нём всё ещё опущены некоторые тонкости. Для более формального определения см. раздел Explanation — https://en.cppreference.com/w/cpp/language/list_initialization.html ↩
3. см. https://stackoverflow.com/questions/54504463/how-does-the-number-of-braces-affect-uniform-initialization ↩