Пространства имен в C++

Базилевич Григорий

Ноябрь 2021

При подготовке лекции и данного конспекта использовался урок с курса «Программирование на языке C++ (продолжение)» на платформе stepik. Автор — Александр Смаль.

Содержание

1 Что такое пространство имен?

Пространства имен (namespace) — механизм в C++, позволяющий разграничить области видимости имен.

Именами в C++ называются:

1. имена переменных и констант,
2. имена функций,
3. имена структур и классов,
4. имена шаблонов,
5. синонимы типов (typedef-ы),
6. enum-ы и union-ы,
7. имена пространств имён.

2 Зачем нужно разграничивать области видимости?

Например, вы создаете большой проект и у вас есть две большие части, одна из которых отвечает за графику, другая — за звук. Тогда вы можете поместить все функции и классы, отвечающие за графику, в один namespace, а то, что отвечает за звук, в другой namespace. Таким образом вы структурируете ваш код, вы синтаксически разделяете элементы, отвечающие за графику и звук, а также избегаете коллизий в именах.

Теперь предположим, что мы разрабатываем библиотеку. Тогда нам необходимо гарантировать уникальность имен, которые мы объявляем в заголовочных файлах. Посмотрим, как эта проблема решалась в языке C, в котором нет пространств имен. Там во избежание конфликта имен использовались префиксы — какая-то строчка, которая приписывалась ко всем именам из библиотеки. Например, вот часть кода из библиотеки Expat:

```
struct XML_Parser;
int XML_GetCurrentLineNumber(XML_Parser* parser);
```

В C++ это можно было бы записать так:

```
namespace XML {
    struct Parser;
    int GetCurrentLineNumber(Parser* parser);
}

XML::Parser parser;
int line = XML::GetCurrentLineNumber(parser);
```

Внутри namespace мы бы могли работать с именами структур и функций напрямую, без префикса. Если же нам понадобиться обратиться к именам из-вне, то тогда мы должны будем обратиться к элементу по полному имени, которое включает в себя имя namespace.

3 Как задаются пространства имен?

Как мы уже увидели из примера, namespace задается несложной конструкцией из одного ключевого слова, названия пространства имен и фигурных скобочек.

```
namespace NAME { /* ... */ }
```

Но есть и более сложные варианты их использования. Например, пространства имен могут быть вложенными:

```
namespace items {
    namespace food {
        struct Fruit { /* ... */ };
    }
}

items::food::Fruit apple;
```

Можно провести некую аналогию с файловыми системами: namespace будут папками, а файлами будут имена. Тогда полный путь к имени, по аналогии с файлами, будет содержать имена всех промежуточных элементов — пространств имен (или папок в файловых системах).

Определение пространств можно разбивать в коде:

```
namespace weapons {
    struct Bow { /* ... */ };
}

namespace items {
    struct Artefact { /* ... */ };
}

namespace weapons {
    struct Sword { /* ... */ };
}
```

Например, это означает, что пространства имен можно объявлять в разных файлах.

И последнее о чём стоит упомянуть — структуры и классы задают одноименные пространства имен:

```
struct Commit {
    struct Author { /* ... */ };
};

Commit::Author committed_by;
```

4 Доступ к именам

Есть несколько простых правил про то, как обращаться к именам в различных пространствах:

1. Внутри одного namespace все имена доступны напрямую.
2. NS:: позволяет обратиться внутрь пространства имен NS.

```
namespace NS {
    int foo() {
        return 0;
    }
};

int i = NS::foo();
```

3. Оператор :: позволяет обратиться к *глобальному пространству имен*.

```
struct Dictionary { /* ... */ };

namespace NS {
    struct Dictionary { /* ... */ };

    ::Dictionary globalDictionary;
};
```

5 Поиск имен

Если же мы не указали компилятору полный путь к имени, то компилятор запустит алгоритм поиска имен — алгоритм разрешения имени.

1. Если такое имя есть в текущем namespace
   - выдать **все** одноименные сущности в текущем namespace
   - завершить поиск
2. Если текущий namespace — глобальный
   - завершить поиск с ошибкой
3. Текущий namespace ← родительский namespace
4. goto 1

Разберем на примере (код далее):

Сначала поиск имен пройдет по namespace subns и, не найдя подходящих имен, перейдет к родительскому namespace. В пространстве имен test поиск найдет два подходящих имени foo — две функции, одна с аргументом float, другая с двумя double. В качестве функции для получения значения переменной global будет выбрана функция с аргументом float как наиболее подходящая. Выбрать функцию для переменной global2 не получится, так как ни одна из найденных функций не подойдет под сигнатуру. Это наглядно демонстрирует, что поиск останавливается при первом совпадении имени.

```
int foo(int a, int b, int c) {
    return 0;
}

int foo(int a) {
    return 1;
}

namespace test {
    int foo(float a) {
        return 2;
    }

    int foo(double a, double b) {
        return 3;
    }

    namespace subns {
        int global = foo(5); // foo(float a)
        int global2 = foo(5, 6, 7); // error
    }
}
```

5.1 using как метод повлиять на поиск имен

Но вдруг нам стало необходимо в каком-то пространстве имен использовать функцию, которая находится в другом пространстве имен. Конечно же, можно всегда указывать полное имя функции, но это может сильно перегружать код и делать его менее приятным для чтения. Фактически, мы просто хотим добавить нашу функцию из другого пространства в наше в качестве ссылки, по аналогии с typedef. Оказывается, что это умеет делать ключевое слово using.

```
namespace ru {
    namespace msk {
        int foo(int i) {
            return 1;
        }

        int bar(int i) {
            return 0;
        }
    }

    using namespace msk; // msk::foo & msk::bar
    // or
    using msk:foo; // msk::foo

    namespace spb {
        int global = foo(5);
    }
}
```

При использовании строчки using namespace msk будут подключены все имена из msk, то есть пройдет этап компиляции. Но у этого способа есть минус — мы подключили лишнее имя bar, которое может конфликтовать с уже имеющимися именами в ru. Тогда можно подключить foo с помощью команды using msk::foo, которое добавит в ru только имя foo, а значит конфликта имен не будет.

5.2 Поиск Кёнинга

```
namespace cg {
    struct RadiusVector { /* ... */ };
    RadiusVector operator+(RadiusVector a, RadiusVector b);
}

cg::RadiusVector a(1, 4);
cg::RadiusVector b(2, 3);

b = cg::operator+(a, b); // OK
b = a + b; // b = operator+(a, b)
```

В данном примере проиллюстрирована такая ситуация: в каком-то пространстве имен у нас есть структура, для которой переопределен оператор сложения. Если запись с явным указанием, в каком пространстве искать оператор, очевидна, то вторая строчка порождает вопросы. Например, такой: как данный оператор найдется, если он находится во вложенном namespace?

Эту проблему в компиляторе решает Argument-dependent name lookup (ADL) или просто поиск Кёнинга. При поиске имени функции на первой фазе рассматриваются имена из текущего пространства имен и *пространств имен, к которым принадлежат аргументы функции.*

То есть на первой фазе у нас будут рассмотрены все имена из namespace cg, а значит нужный нам оператор найдется.

Обратите внимание, что если на первом шаге у нас будет найдено более одного подходящего имени, то компилятор не будет выбирать и упадет с ошибкой. Например, если у нас аргументы из двух разных namespace и в каждом из них объявлен оператор с определенным порядком аргументов.

```
namespace cg {
    struct x__ {};
}
namespace cg2 {
    struct x__ {};
    int operator+(cg::x__ a, x__ b) {
        return 1;
    }
}
namespace cg {
    int operator+(x__ a, cg2::x__ b) {
        return 0;
    }
}

int global = cg:x__() + cg2::x__();
```

6 Безымянный namespace

Безымянный namespace — пространство имен с гарантированно уникальным именем.

```
namespace {
    struct Test { /* ... */ };
}
```

Будет эквивалентно:

```
namespace $GeneratedName$ {
    struct Test { /* ... */ };
}
using namespace $GeneratedName$;
```

где $GeneratedName$ — случайное имя, сгенерированное компилятором.

С помощью безымянного namespace можно подключить все имена из данного пространства в единственный файл. Такие пространства имен служат заменой для static.

7 Выводы

1. Используйте пространства имён для исключения конфликта имён.
2. Помните, что поиск имён прекращается после первого совпадения. Используйте using и полные имена.
3. Не используйте using namespace в заголовочных файлах.
4. Всегда определяйте операторы в том же пространстве имён, что и типы, для которых они определены.
5. Используйте безымянные пространства имён для маленьких локальных классов и как замену слова static.
6. Для длинных имён namespace-ов используйте синонимы:

```
namespace csccpp17 = ru::spb::csc::cpp17;
```