C++ Модули 3-4

Тарабонда Герман C++

Содержание

1 Пространство имен (namespace)

Пишем большую программу, у нас есть 2 класса User, но хотим использовать это слова и в бд и на сервере. Можно сделать так

```
class Database{
   class User{ ... };
};
class Application{
   class User{ ... };
};
```

Есть проблема. Первая: надо всегда писать слово `static` или постоянно проверять, что объект типа `Database` ровно один. Вторая: если файлов много, то надо писать 10 разных классов или 1 хедер на всех и тогда каждый раз нужно будет перекомпилировать этот большой хедер.

Придумали namespace.

Пример обращения:

```
namespace database{
class User{ ... };
} // namespace database
namespace application{
class User{ ... };
} // namespace application

// чтобы обратиться нужно
application::User;
```

Можно использовать так:

```
int main(){
   using application::User; // делаем User из application до }
```

```
}

void connectTo(){
   using namespace database; // в среднем не очень
}

std::vector // так все же лучше
```

Обычно using пишут в сппишниках. В хедерах ставим бан этим штукам.

Бывают вложенные namespace

```
namespace database{
   void connect(){internal::conectEx( ... );};
   namespace internal{
      void connectEx();
   } // namespace internal
} // namespace database
database::internal::connectEx( ... );

namespace database::internal {
}
namespace database {
} // можно переоткрывать ыnamespac
// это все работает как " допиши перед названием database::"
```

Еще бывает глобальный namespace.

```
int x;
::x = 10; // глобальный namespace
static int y; // влияет только на линковку
```

Еще существуют анонимные namespace

```
namespace {
   void foo();
} // анонимный namespace == все что внутри становится static
foo(); // можно писать так, а не A::foo() и foo() не виден в других единицах
    трансляции.
// В C++ лучше писать всегда анонимный namespace
```

---

2 Правила поиска имен

Unqualified lookup – когда пишем без двоеточий: `foo();`

```
// (4) потом ищется в глобальном namespace до строчки с foo()
namespace application{ // (3) потом здесь до строчки с foo()
class User{ // (2) затем везде в классе ищем foo()
```

```
    void x() { foo(); }; // (1) сначала ищется ссылка на foo в x до строчки с foo()
};
}
```

Qualified lookup – когда пишем с двоеточиями: `::foo();`

Тут ясно где искать нужную функцию/ переменные.

Например:

```
database::internal::foo();
// Порядок вызовов
// (1) qualified lookup foo() в internal
// (2) qualified lookup internal в database
// (3) unqualified lookup database
```

А теперь стреляем себе в ногу

```
#include <vector>
namespace foo {
    std::vector<int> v;
} // OK

#include <vector>
namespace foo {
    namespace std{
    }
    std::vector<int> v; // не надо пересекаться с глобальным
} // не OK
```

```
namespace mysql {
    void connect();
}

namespace application::database::mysql {
}
namespace application::database{
    void connect(){ mysql::connect(); } // не OK, так как будет искать в
     application::database
    // void connect(){ ::mysql::connect(); }  OK
}
```

3 Argument-dependent-lookup (ADL)

Там много сложных правил, которые зависят от аргументов (из названия следует), вот некоторые примеры с лекции:

Если видим неквалифицированный вызов функции, то смотрим на типы аргу-

ментов и ищем функции во всех связанных namespace'ах

©Егор Суворов

Пример:

```
namespace std {
    ostream& operator<<(ostream &os, string &s);
}

std::ostream &output = ... ;
std::string s;
output << s;  // У оператора нет квалифицированного вызова
/* Поэтому компилятор смотрит на аргументы, смотрит в каком namespace находятся и
    смотрит подходящую функцию */
```

Удобно для операторов.

Если мы вместе с классом дали пользователю какую-то функцию, то она должна иметь те же моральные права, что и методы.

©Егор Суворов

Пример:

```
std::filesystem::path from("a.txt");  // C++17
std::filesystem::path to("a-copy.txt");
copy_file(from, to);  // OK
copy_file("a.txt", "a-copy-2.txt");  // не OK
std::filesystem::copy_file("a.txt", "a-copy-2.txt");  // OK
```

Так же работает с range-based-for и structured binding

```
namespace ns {
    struct Foo { int x; };
    const int* begin(const Foo &f) { return &f.x; }
    const int* end(const Foo &f) { return &f.x + 1; }
};
int main() {
    ns::Foo f{20};
    for (int x : f) std::cout << x << '\n';
}
```

Нужны `begin` и `end` только связанные с `Foo`, поэтому можно пользоваться только ADL, иначе ошибка, глобально объявлять нельзя.

Теперь пример, когда он выключен:

```
namespace foo {
    namespace impl {
        struct Foo { int x; };
```

```
        int func(const Foo &f) { return f.x; }
        int foo(const Foo &f) { return f.x; }
    }
    using impl::Foo;
}
namespace bar::impl {
    struct Bar{ int x; };
    int func(const Bar &f) { return f.x; }
}

int main() {
    foo::Foo f;
    bar::impl::Bar b;
    func(f);  // OK Ищет сначала определение Foo, нашел, все ок
    func(b);  // OK Находит Bar, все ок
    foo::impl::foo(f);  // Qualified lookup, no ADL, OK. Просто ищет по qualified
    lookup
    foo::foo(f);  // Qualified lookup, no ADL, не OK. Не находит по qualified lookup
    foo(f);  //  не OK: namespace foo. Коллизия имен, считает, что это namespace
}
```

`std::swap` плохо работает для своих типов. Частичную специализацию для своих классов сделать нельзя, `a.swap(b)` не работает для `int`. Поэтому есть вариант с ADL:

```
using std::swap;
swap(a, b); // вызывается ADL, сначала ищется в файле, затем в std
```

Как работает ADL? А вот так:

```
void func(int); // (1)
namespace ns {
   int x(int); // (2)
   void func(const char* ); // (3)
   void finc(int, int); // (4)
   //using ::func;  (5)
   void foo() {
      int x; // (6)
      func(10); // Спускается в ns, видит (3) и (4), видит что не подходит, не OK
      // Если добавить (5), то будет OK, так как найдет (1)
      x(10);  // Видит (6), понимает, что какого фига и не OK
   }
}
```

Если что-то находим, то не включаем ADL, иначе включаем

```
namespace ns {
    struct Foo {};
    void foo(Foo) {}  // (1)
```

```
    void bar(Foo) {}  // (2)
    void baz(Foo) {}  // (3)
    void qwe(Foo) {}  // (4)
}
int baz;              // (5)
void qwe() {}         // (6)
struct S {
    ns::Foo f;
    void bar() {}     // (7)
    void method() {
        foo(f);  // Включился ADL, нашли (1).
        bar(f);  // Нашли (7), не стали включать ADL, ошибка компиляции.
        baz(f);  // Нашли (5), не стали включать ADL, ошибка компиляции.
        qwe(f);  // Нашли (6) и (4), разрешили перегрузку.
    }
};
```

А теперь, что происходит, если включен ADL:

```
namespace root { // (1)
    namespace ns1 { struct Base1 {}; void func1(...) {} } // (2)
    namespace ns2 { struct Base2 {}; void func2(...) {} }  // ADL (3)
    namespace ns3 {  // ADL (4)
        void func3(...) {} // (5)
        struct Container : ns1::Base1 { // (6)
            static void func4(...) {} // (7)
            struct Derived : ns2::Base2 {}; // (8)
        };
    }
    void func5(...) {} // (9)
    using Derived = ns3::Container::Derived; // (10)
}
void (*data)(std::tuple<root::Derived*>) = nullptr; // (11)
sort(data);  // std::sort found, не OK (11) -> видим tuple -> std
func1(data);  // не OK, (11) -> указатель на root::Derived -> (10) -> (8) -> нашли
    (5) и (3)
func2(data);  // OK
func3(data);  // OK
func4(data);  // не OK
func5(data);  // не OK
```

Теперь выбор перегрузок при ADL:

```
namespace ns1 {
    struct Foo {};
    namespace Foo_adl { void func(Foo) { cout << "(1)\n"; } }
    using Foo_adl::func; // Это имеет значение
}
```

```
namespace ns2 {
    struct Bar {};
    namespace Bar_adl { void func(Bar) { cout << "(2)\n"; } }
    using namespace Bar_adl; // Это имеет значение
}
void func(ns1::Foo) { cout << "(3)\n"; }
void func(ns2::Bar) { cout << "(4)\n"; }
// ....
ns1::Foo f; ns2::Bar b;
ns1::func(f);  // (1), qualified lookup
ns2::func(b);  // (2), qualified lookup
::func(f);     // (3), qualified lookup
::func(b);     // (4), qualified lookup
// func(f);    // ambiguous: (1) or (3), global and ns1
func(b);       // (4), global and ns2 (using namespace)
```

Теперь с помощью ADL можно делать скрытых друзей!

```
#include <iostream>
namespace ns {
    struct Foo {
        friend void foo(Foo) {}  // implicitly inline.
    };
    // void foo(Foo);  // (1)
}

ns::Foo f;
foo(f);      // OK, ADL
ns::foo(f);  // не OK, так как не находит
// Расскоментим (1), и заработает прошлая строчка
```

Важно знать! Unqualified lookup смотрит на конструкторы, а ADL хрен клал на все это дело.

```
namespace ns {
    struct Foo {};
    struct Bar { Bar(Foo) {} };
    void foo() {
        Foo f;
        auto x = Bar(f);  // OK, смотрит в конструктор
    }
}
// ....
ns::Foo f;
auto x = Bar(f);  // не OK, а эта скотина не смотрит в конструктор, просто видит
    класс с таким же названием
```