

Содержание

1 Namespace, cin, cout	1
-------------------------------	----------

1 Namespace, cin, cout

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**. Вторая: если файлов много, то будет больно.

Придумали namespace.

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

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

Мы не будем мешать другим, используя это.

```
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 спшишниках. В хедерах ставим бан этим штукам.

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

```
namespace database::internal{  
}  
namespace database {  
} // можно так
```

namespace (работает как "допиши пожалуйста") пишем в хедере (тут объявление) и спшишнике (тут определение).

Еще бывает глобальный namespace.

```
int x;  
::x = 10; // глобальный namespace
```

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

Как происходит разрешение имен.

Первое: неквалифицированный lookup

```
namespace application{ //потом до foo здесь (3)  
class User{ //затем здесь везде (2)  
    void x() { foo(); }; //сначала ищется ссылка на foo в x до foo (1)  
};  
}
```

Второе: квалифицированный lookup

```
database::internal::foo(); // тут ставятся двоеточия, ищется в какомто- namespace  
// но database будет запущен неквалифицированным lookupом-
```

А теперь стреляем себе в ногу

```
#include <vector>
namespace foo {

    std::vector<int> v;
} // скомпилился
```

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

```
namespace mysql{
}
//-----
namespace application::database::mysql{
}
namespace application::database{
    void connect(){ mysql::connect(); } // не OK
    // void connect(){ ::mysql::connect(); } OK
}
```

::std писать или нет лучше спрашивать у практиков/тимплейтов

стандартная библиотека C++11

Иногда ошибочно называют STL, но это не совсем так, так как это всего лишь часть библиотеки.

```
#include <iostream> // в std
#include <stdio.h> // ::fopen
#include <cstdio> // ::std::fopen
```

Что у нас есть:

```
std::ios // везде std::
|
|-----+
istream      ostream
|           \       /
ifstream    istream  ofstream

basic_istream<char> // тоже самое что и istream

istream cin;
ostream cout, cerr;
```

Лучше пользоваться более общими классами (до istream, ostream, iostream)

```
#include <iostream>
int x;
std::string y, z;
std::cin >> x >> y; // читается до ближайшего ' '
std::cin.getline(z);
std::cout << (x + 5) << "10" << z; // скобочки, иначе проблема с порядком операторов
std::cout << "1" << std::endl; // еще и сброс буфера
```

```
std::ifstream f("1.txt"); // автоматически вызовется конструктордеструктор/
// std::ifstream f("1.txt", ios_base::in | ios_base::binary); бинарный ввод
f >> x;
f.close() // если очень хочется, то можно

std::stringstream s;
s << 10 << " " << 20;
std::string str10;
s >> str10; // здесь будет 10
```

```
std::ifstream f("1.txt"); // автоматически вызовется конструктордеструктор/
f >> x;
f.close() // если очень хочется, то можно

std::stringstream s;
ostream& operator<<(ostream &os, int &v){
    os << ('0' + (v*10));
    return os;
}

(((cout << 1) << 2) << 3); // так парсится

istream& operator>>(istream &is, int &v){
    v = 10;
    return is;
}
```

```
(s << 10) >> str10; // не работает так как типы разные
```

```
std::cout.setf(std::ios::hex); // все вводим в шестнадцатеричном
std::cout << 10 << std::ios::hex << 20 << std::ios::dec << 30; // так можно вывести
    20 в шестнадцатеричной и попасть в дурку
```

```
// Но можно перегрузить
```

```
struct hex_impl { ... } hex;
// и перегружаем <<
```

```
// или  
ostream& hex(ostream &os){  
}  
std::cout << change_base(13);
```
