Плюсеки

Содержание

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 сппишниках. 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
```

namepace (работает как "допиши пожалуйста") пишем в хедере (тут объявление) и спшнике (тут определение).

Еще бывает глобальный namespace.

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

```
static int x; // локальный для спшника
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 неквалифицированном lookupом-
```

А теперь стреляем себе в ногу

```
#include <vector>
namespace foo {

   std::vector<int> v;
} // скомпилится

#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      iostream      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 буффера
```

```
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 шестнадцетеричном
std::cout << 10 << std::ios::hex << 20 << std::ios::dec << 30; // так можно вывести
    20 в шестнадцетеричной и попасть в дурку

// Но можно перегрузить

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

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

`friend` не стоит пользоваться, так как будет неявная зависимость. Так же можно будет залезть в состояние шаблона, что плохо.

2 Долги + обсуждвем крестики-нолики

Долги

```
std::ostream& hex(std::ostream&);
cout << hex;
cout << write_le_int32(12345); // хотим чтобы работало

// минимальная рабочая версия

struct write_le_int32 {
   explicit write_le_int32(int val); //  так как от одного параметра
   friend ostream opeartor<<(ostream&, const write_le_int32&);
}

struct read_le_int32 {
   explicit read_le_int32(int& val); //  так как от одного параметра
   friend istream opeartor>>(istream&, const read_le_int32&); // здесь const нужен,
    иначе не скомпилируется
   int& val;
}

// Если не работает с fstream подключаем fstream или #include <iosfwd>
// ( в нем объявляются только имена классов)
//Если есть чтото- типа is.read(), то нужно #include <iostream>
```

Здесь все огребают: `employee.cpp`

```
#include "bin_manip.h"
#include <fstream> // нужно подключать именно в этот файл, а не в bin_manip.h

... (ifstream f, ){
   f >> read_le_int32();
}
```

Обсуждаем крестики-нолики

Не делаем циклические зависимости, так как мы не можем протестировать программу по частям. Есть классная книжка «Банда четырех» (жаргонное название).

Можем придкмать расслоение классов на слои. В одном слое классы могут ссылаться друг на друга, но на все нижние можно.

```
// одна из моделей проектирования
E,ployee::clone

Employee *a;

b = a->clone();
```

MVC

Отделим модель, которая хранит функции и поля, отвечающие за саму игру. Есть еще часть view, которая отвечает за интерфейс.

```
class GameModel {
   FieldState field[10][10];
   Player cur_player;
   GameState game_state();
   void make_turn(int x, int y, int pl);
   //  еще какоето- количество методов
}
```

Можно улучшить, добавив enum.

```
enum FieldState {E, X, O}; // так мы можем засорить пространство имен
enum class FieldState {E, X, O}; // а так не будем и можно вызывать FieldState::X
```

Есть еще класс view, который в диком мире еще можно разделить.

```
class View {
   CView(Model&);
   void RunGame(); // выводит текущее состояние игры и т д
}
```

Есть еще допзадание, где нужно сделать не консольку, а создать графическую версию крестиков ноликов и т д с помощью ncues.

Что происходит в диком мире по мнению Егора? К примеру, можно подключать плагины.

```
class View {
   virtual void RunGame();
}
```

```
class NCView{
   void RunGame();
}
```

Лучше использовать наследование, а не какие-то ифы, чтобы в будущем иметь возможность расширять программу.

Можно еще вынести логику контролера с помощью ModelViewController.

Автоесты

Нужно написать свой фреймворк для домашки. Магии нет, все просто (по мнению Егора).

Зачем же нужны автотесты и как их делать? Первое: у нас что-то работает неверно => хотим быстро понять и узнать, где что-то не работает. Второе: хотим тесты без рандома. Третье: хотим независимые тесты (лучше с котами). Четвертое: хотим еще уметь независимо запускать тесты для ускорения отладки. Пятое: еще удобно создавать группировки тестов. Шестое: писать тесты приятно (по мнению Егора).

Фреймворки предназначены для написания unit тестов. Они проверяют крайние случаи. Чаще всего пользуются чужими фреймворками: Google test, doctest.

А теперь делаем наброски (рисуем картину маслом):

```
assert(foo(2,2) == 4); // есть в языке, но както- пока чтото- не очень

#define DO_CHECK(expr) \
   TestCheck::check(expr, __FILE__, __LINE__)

class TestCase{
   virtual void RunAllTests();

   static void check(bool, const char *f, int l); // если ошибка вывели, что есть
    ошибка
   // еще передаем файл и строку, что не крута
   // можно работать с макросами написано( выше)
   static int failed, total;
   static void printResults();
};

class ModuleTestCase : TestCase{
   void RunAllTests(){
      DO_CHECK(foo(2,2) == 4)
   }
};

// гдето- в maine':

TestCase* cases[]= {
   new ModelTC;
```

```
    new ...
}
```