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 сппшнике (тут определение).

Еще бывает глобальный namespace.

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

```
static int x; // локальный для сппшника
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::ifstram f("1.txt"); // автоматически вызовется конструктордеструктор/
// std::ifstram 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::ifstram 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 шестнаддетеричном
std::cout << 10 << std::ios::hex << 20 << std::ios::dec << 30; // так можно вывести
    20 в шестнаддетеричной 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 ...
}
```

---

3 Сборная соляночка

hiding

isocpp.org/wiki/faq Ссылочки на wiki

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

struct Bar : Foo {
    void foo(double); // 2
};

Foo f;
f.foo(1); // вызовется 1
f.foo(1.2); //вызовется 1

Bar b;
b.foo(1.2); // вызовется 2
b.foo(1); // вызовется 2 перегрузки( не включаются)
```

То что произошло выше называется `hiding`.

```
struct Foo {
    virtual void foo(int); // 1a
    virtual void foo(double); // 2
};

struct Bar : Foo {
    void foo(int) override; // 1b
};

Bar b;
b.foo(1.2); // вызовется 1b
```

```
b.foo(1); // вызовется 1b
// потерялись перегруз очки
// но можно обратиться к функциям

Foo& b2 = b;
b2.foo(1); // вызовется 1b
b2.foo(1.2); // вызовется 2
```

Можно сделать тоже самое, но специальным синтаксисом.

```
// первый вариант
b.Foo::foo(1.2); // 1
b.Foo::foo(1); // 1



// второй вариант

struct Bar : Foo {
   void foo(int) override;
   using Foo::foo;
};

//. Если убрать virtual and override, то:

Foo f;
f.foo(1); // вызовется 1b
f.foo(1.2); //вызовется 1b

Foo& b2 = b;
b2.foo(1); // вызовется 1a
b2.foo(1.2); // вызовется 2
```

---

using

```
struct Foo {
private :
   void foo(int) {};
};

struct Bar : Foo {
   using Foo::foo; // когда ну ооооочень хотим вытащить функции из привата
};

struct Foo {
private :
   void foo(int) {};
```

```
};

struct Bar : Foo {
   void foo(int); // ошибка линковки
};
```

Множественное наследование

```
struct PieceOfArt {
   std::chrono::time_point date;
   // ...
};

struct Music : PieceOfArt {

};

struct Lyrics: PieceOfArt { };

struct Song : Music, Lyrics { // инициализируются в том же порядке, в каком они
    написаны здесь
   std::string album;
   Song(...): Music(...), Lyrics(...), album(...) {];
   using Music::date;
};

void printLyrics(const Lyrics&);
print Lyrics(s);
s.date; // можно s.Music::date;
```

Лежал сначала базовый класс, затем наследник.

Не надо использовать `void*`.

```
// Размеры структур
struct A {}; // 1
struct B: A { char c; }; // 1
struct C : A {}; // 1
struct D : B, C {}; // 2
```

```
struct Person { ... };

struct Student : virtual Person {
   Student( ... ) : Person( ... ) {}
};

struct Employee : virtual Person {};
```

```
struct MagicStudent : Student, Employee {
   MagicStudent( ... ) : Person( ... ), Student( ... ), Employee( ... );
};
```

4 Обработка ошибок

Ошибки

- Ошибки программирования

```
void foo(char *s){
   printf("\% s", s); //printf(s);
}
```

- Ошибки окружения (формат файла, файл не найден)

```
void read_people() {
   FILE *f = fopen(filename, "r");
   if (!f){
      printf("Unable to open file");
      return;
   }
   // ...
   fclose(f);
}
```

После каждой функции писать `if`.

```
int sqlite3_open {
   const char *fname;
   sqlite3* *out;
};
```

```
struct invalid_vector_format {};
std::vector<int> ReadVector(){
   int n;
   if (!(cin >> c)){
      throw invalid_vector_format();
   }
   vector<int> Res(n);
   for (int i = 0; i < n; i++){
      if (!(cin >> Res[i])){
         throw invalid_vector_format();
      }
   }
```

```
   return Res(n);
}
```

```
void solve() {
   std::vector<int>
   writeAnswer();
}
int main() {
   try {
      char*x = malloc(k);
      solve();
      free();
   } catch (invalid_vector_format &err){
      printf("Oops");
   }
}
```

Контракты, гарантии, nothrow

5 Exceptions, шаблоны

```
strict Foo {
   std::vector<int> a, b;
   Foo(const std::vector<int> &_a, const std::vector<int> &_b):
      a = _a,
      b = _b {}
}; // все верно написано

try {
   Foo f(... , ...);
   f...
} catch (const bad_alloc&) {
}
Foo *f = new Foo(... , ...);
throw
```

```
strict Foo {
   std::vector<int> a, b;
   Foo(const std::vector<int> &_a, const std::vector<int> &_b) try : // можно так
    писать
      a(_a),
      b( _b) {} catch () {
         std::cerr << ...;
         throw;
      }
```

```
};
```

```
strict Foo {
   int *a, *b;
   Foo()
   try
      : a(new int[10])
      , b(new int[10]){
   } catch (const bad_alloc&) {
   } // UB
};
```

Управление памятью двух указателей в одной структуре – что-то плохо. Пользуемся uniqueptr.

```
new X(make_unique<Y>( ... ))
```

Зачем не использовать исключения?

```
-f noexcept
```

Почему так? Во-первых, исторически (пример Гугл). Во-вторых, скорость (исключения могут быть медленными) (зависят от фазы Луны). В-третьих, ABI (Application Binary Interface).

Шаблоны

Хотим чтобы структура работала с разными типами.

```
#define MY_ARRAY(TYPE) \
class Array##TYPE { \
private: \
   TYPE *data; \
   size_t len; \
public: \
} \

#include "my_array.h"
MY_ARRAY(int);

Array int a;
```

В C мы могли бы добавить макрос. Проблемы? Некрасиво... Странные ошибки... Не работает с namespace... А теперь делаем все православно:

```
template<typename T> // template<class T> абсолютно тоже самое
class Array { // это шаблон класса
private:
```

```
   T* data;
   size_t len;
public:
   T& operator[](size_t i) { return data[i]; }
}
```

Все проверяет синтаксически.

```
Array<int> // класс
Array<char> // класс
Array<Array<int> > // можно вкладывать
```

Это независимые классы (которые сверху).

```
// .h
template<typename T>
class Array { ... };

template<typename T>
T& Array<T>::operator[](size_t i){ ... };
```

6 Шаблоны наносят ответный удар

```
template<typename T, typename U>
void foo(U x);

foo<int>(10.0); // T = int, U = double
foo(10.0); // не скомпилиться, если не:

template<typename T, typename U>
```

```
typedef vector<int> v;
typedef vector v; // До C++11 не скомплилируется
#define v vector // :(

template<typename T>
using v = vector<T>;
```

```
template<typename T=int, std::size_t N = 10>
struct array {
   T arr[N];
};

template<typename T, std::size_t N>
T& array<T,N>::operator[](){};
```

Шаблоны плохо парсятся, если что добавьте скобочки. В качестве шаблона можно кидать сам шаблон.

```
template<typename T, template<typename>typename C>

struct priority_queue {
   C<T> data;
};

pr_q<int, pr_q> // ок, если укажем что устанавливаем
```

```
shared_ptr<Derived> d = ...
shared_ptr<Base> b = d; // не скомпилируется просто так
```

7 Я – черешня

Стек

```
#include <memory>

Stack(const stack&other) : data(std::aligned_alloc(alignof(T), sizeof(T).other.len)),
    len(other.len), cap(other.cap){
   // вопрос с bad_alloc
   // + дописать static_cast
   try {
   for (; len < other.len; len++)
      new(data+len) T(other.data[i]);
   } catch ( ... ) {
      // удалить первые len элементов в обратном порядке
      for (;len > 0; len--)
         data[len-1].~T();
      free(data);
      throw;
   }
}
```

```
void Reserve(size_t newcap){
   T newdata = static_cast<T*>(...);
   // копипаст вышенаписанного кода
   // идти в обратном порядке
   free(data);
   data = newdata;
   cap = newcap;
}
```

```
template<...>
Struct stack{
private:
   struct stack_holder {
      T* data;
      size_t len, cap;
      stack_holder(size_t _cap = 0):
         data( ... ),
         len(0),
         cap(_cap) {};
      ~stack_holder() {
         for (; len > 0; len--)
            data[len-1].~T();
         free(data);
      }
      stack_holder(stack_holder const&) = delete;
      stack_holder operator=(stack_holder const&) = delete;
   }
   stack_holder h;
   stack(const stack &other): h(other.len){
      for (; h.len<other.h.len; h.len++)
         new (h.data + h.len) T(other.h.data[h.len]);
   }
   void reserve( ... ){
      stack_holder newh( ... );
   }
}
```

move-семантика

```
struct Foo {
   string a, b;
   Foo(... _a, ... _b) : a(_a), b(_b){};
}
string read(){
   string s;
   cin >> s;
   return s;
}
Foo readFoo(){
   string a=read(), b =read();
   Foo f(std::move(a), std::move(b));
   return f;
}
Foo f=readFoo();
```

Много копирований, а давайте будем двигать.

```
string (string&&) noexcept;
string& operator=(string&&);
```

Сделаем умный swap

```
template<typename T>
void swap(T& a, T& b) {
   T t = std::move(a);
   a = std::move(b);
   b = std::move(t);
}
```