

(5) `list<T>::size()`

`size O(1)`      `splice O(N)`

`size O(1)`      `splice O(1)`

Более глубокий взгляд на концепцию

▷ фундаментальные концепции C++ (1 ГЛАВА)

- 1) `f(T x)`
- 2) `f(T& x)`
- 3) `f(const T& x)`

`const int & g = 0`

- 1) `f(g)` →  $T = \text{int}$
- 2) `f(g)` →  $T = \text{const int}$
- 3) `f(g)` →  $T = \text{int}$

(60) const T&    const T&&

`const int * const f(const int& y) {`

`const int *ptr = &y; // f(x) - OK`

`y return ptr;`

↑

↑ *перегрузка ф. с аргументом, чтобы различать оба*

↑ *один и тот же ф., в зависимости от ограничения*

`const int & f(const int&& x) {`

`return nullptr;`

↑ *функция для получения*

func (value, evale)

Композитные концепции

`struct A {`

`virtual A* f() { return this; }`

`};`

`struct B : public A {`

~~`B*`~~ `f() override { return this; }`

`};`

Следующий раздел дает более детальное описание того, как композитные функции работают.

↑ *кооперативные*

↑ *функции*

↑ *кооперативные*

↑ *функции*

157  
class TypeAction : public IActions {

char symbol;

Cursor cursor;

public:

TypeAction (char, Cursor);

void Do (TextEditor \* edit) const override;

edit → text\_[cursor\_.line].insert(

cursor\_.col, symbol);

edit → cursor\_ = {cursor\_.line,

cursor\_.col + 1};

}

void Undo (Text \* edit) const override;

edit → text\_.erase(cursor\_.col);

edit → cursor\_ = cursor\_;

}

158  
Operator overloading example / copy constructor

① class A {

public:

(inline) void f () { cout << 0; }

}

void A::f () { cout << 0; }

Iterator traits < It >

② iterator\_traits < It > :: value\_type

:: reference

:: pointer

:: iterator\_category

(input iterator tag,  
random access iterator, ...)

class ListIterator {

public:

typedef int value\_type

typedef std::bidirectional\_iterator\_tag

(using)

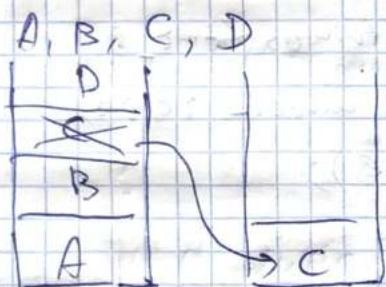
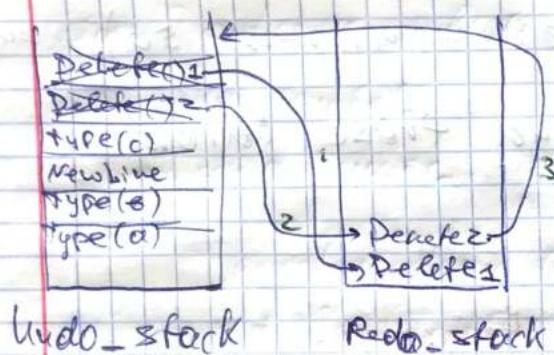
iterator\_category

Adapters

③

④ std::next, advance, prev

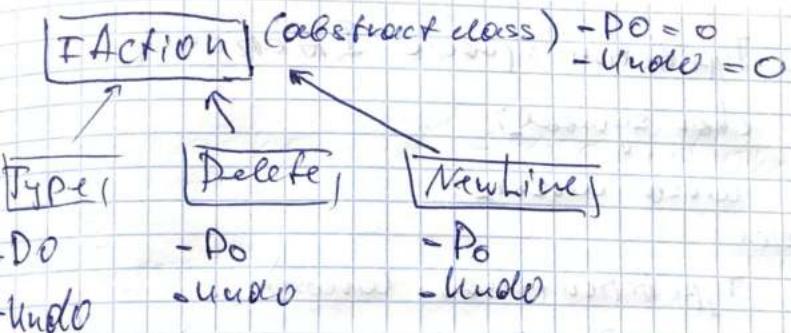
155



Even occur undo more often, a norm & P  $\Rightarrow$  happens occurs Redo

types DATASTRUCT Command (Action)

Text Editor



void Do (TextEditor \* edit) const {

}

156

TextEditor {

Type (char X) {

auto new\_type = make\_unique<Type>(X, cursor);  
new\_type  $\rightarrow$  Do (this)  
undo.push(..)

Delete() {

auto new\_delete = ... ;

new\_delete  $\rightarrow$  Do

undo.push(..)

}

153

```

vector<int> v(1000, 50);
generate(v.begin(), v.end(), [generator, &distr]{
    () {
        return distr(generator);
    };
});

```

```

cout << count(v.begin(), v.end(), 50)
auto it = find(v.begin(), v.end(), 50)
while (it != v.end()){
    v.erase(it);
    it = find(v.begin(), v.end(), 50);
}

```

← Xorun yopats bee „50“  $\rightarrow$   $x \oplus x = 0$   
 копицо)

~~auto it = std::remove(v.begin(), v.end(), 50)~~

// removes elements bee an-ya ≠ 50 & returns  
 copy-  
 iterator  
 new begin 50 6 7 50 1 ) remove  
 up to.  
 new beg. 6 7 1 50 1 ↑  
 v.erase(it, v.end())

154

v.erase(v.begin(), v.end(), 50), v.end())

remove and erase idiom

size = 3  
 + разоб спрятано

### Задача "Редактор"

TEXT Editor

std::vector<string> text\_;

Type Cursor cursor {0, 0};

Type (symbol);

Delete();

PasteNewLine()

new | new // текст  
 new 2

new  
 1 new

Undo() (ctrl+z)

Redo() (ctrl+y)

[Type('a'), Type('b'), NewLine(), Delete()]

(передать копицо one point)  
 Undo: Type('b') [a, b]

Type('b') [c, d]

## #include <algorithm>

- ~~Функции и алгоритмы~~ properties

Многие гарантии для алгоритмов

✓ all\_of ✓ find //адресует константные

✓ any\_of (хорошо ✓ find\_if  
для <5 → true)

✓ none\_of (хорошо ✓ for\_each //запоминает  
ан-0бс <5 → true) ✓ for\_each //запоминает  
for\_each(v.begin(), v.end(), [](int x){cout << x;});

vector<int> v{1, 2, 3, 4}; predicate

all\_of(v.begin(), v.end(), [ ](int x){  
return x < 5;});

// предикат не будет проверять

каждую итерацию с 5

стока & <5 → true, иначе → false

unordered\_map<int, int> m;

m.find(6); // O(log n) или O(1)

find(v.begin(), v.end(), 8); // O(n)

## Алгоритмы ~~второго~~ класса

✓ find\_if (v.begin(), v.end(); [ ](int x){vector<x> % 2 == 1);  
результат не содержит нечетных элементов  
аналог v.end())

vector<int> v(10);

std::mt19937 generator (std::random\_device{}());

std::uniform\_int\_distribution<int> distr(1, 10)

for\_each (v.begin(), v.end(), [ ](int x){cout <<

[&generator, &distr](int x))

x = distr(generator);});

## • Модифицирующие алгоритмы

✓ copy (begin, end, target) or go <sup>до</sup> <sub>когда</sub>

✓ copy\_if //копирует только те элементы, где предикат true

✓ move (begin, end, target) //перемещение в target

✓ move\_if

✓ remove

✓ replace (b, e, s, rc, dest)

✓ reverse //инвертирует

✓ rotate //сдвигает

### • СОРТИРОВКА:

sort, partition, stable\_sort, stable\_partition, is\_sorted, partial\_sort, nth\_element //n-элемент.

replace min parab [if phi, then replace sort +]

```

int * ptr = new int[10];
for (int i=0; i<10; ++i) {
    cout << ptr[i];
}
delete [] ptr;

```

Bage  
normal

~~На~~ ~~Ондреевская~~ ~~всегда есть наимен~~ ~~нон 10 итат~~ ~~(когда~~ ~~наимен~~)  
~~годов~~ ~~наимен~~ ~~день~~ ~~прекрас~~  
~~данных (БЕЗОПАСНОСТЬ)~~

## λ-Функции

```
vector<string> v{"a", "ab", "bba", "cabc"};  
sort(v.begin(), v.end()); // Compare length
```

```
struct CompareLength {
    bool operator() (const string &lhs, const string &rhs)
        return lhs.size() < rhs.size();
```

```
sort(v.begin(), v.end(), []  
      (const string &lhs,  
       const string &rhs)  
    {  
        return lhs.size() < rhs.size();  
    }  
);
```

```

size_t x = 9;
sort(v.begin(), v.end(), [x](const...)) {
    x = 10; //ICE
    return lhs.size() + x - rhs.size() - x
}

y
sort(v.begin(), v.end(), [x](..) mutable {
    x = 3; //OK
    return ...
})

```

~~[x]~~  $[ \wedge x, v ] ( \dots ) \{ \dots \}$

auto f1 = []() { y; } [ ] - see no closure  
 auto f2 = []() { y; } [ ] postpone run y f1 u f2  
 std::function<int(int)> f4 = [](int x) { return x+y; }

143

$[ ](\text{int } x)$  if (...) return x;  
 else (...) return 1.0 } // CE

$[ ](\text{int } x) \rightarrow \text{int } f(x)$

Бозбажылардың 3 таралықтарында күрт

Иллюстрация  $\lambda$ -ф.

auto  $f = [ ](\text{auto } x) \{ \text{return } x; \}$

$f(5)$

$f(5.0)$

$\text{decltype}(f)$  // Бозбажылар  
текн. реализм.

$\text{std}::\text{set} < T,$   ~~$\&f$~~   $\&f$  : function<cool(T,T)>

$\text{cool } f(T,T)\{\}$  // озакчылар

$= \text{Comp}()$  cool, uparam:  $T, T$

$\text{std}::\text{function}<\text{bool}(T,T)> \text{func} = f;$

$= [ ](\dots) \&g$

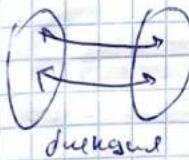
$= \text{Comp}();$

ss

$\&f$  : Any

144  
CEMUMAP

19.05.20



unordered\_map<int, int> first;

unordered\_map<int, int> second;

first[x] = y; // right side  
second[y] = x; } ?  
first[y] = x;

n/a/n:

const int n = 100;

vector<vector<int>> v;

v.reserve(100); v.resize(n) // Бозбажылар  
оңа n бөлшектелді

for (int i=0; i < n; ++i) {

v[i].reserve(n); // UB

for (int j=0; j < n; ++j) {

v[i].push\_back(j);

}

random or v.begin()..v.end()

for (const auto &row : v) {

for (int val : row) {

cout << val;

}

145

```

struct A {
    int z;
    operator()(x, y) {
        return |x-z| < |y-z|
    }
};

std::sort(b, e, A{5});

```

$[x, y, z]$  ( $\text{int } t$ ) ( $\text{return } x - y + z - t; y$ )  
 $x, y, z - \text{const}$   
 корум  
~~func & Cmputable~~  
~~return, const,~~  $\Rightarrow$   $x, y, z$  mixed results  
~~operator() const~~

$[&x, &y, &z]()$   $\downarrow \dots y$   
 ~~$\star y - \text{модификатор} \Rightarrow$  модификатор~~  
 $z - \text{зарезервировано}$

$[=]$  - захватывает все иконы зарезерв.

$[*]$  - no обработка

$[\&x, x]$  - все иконы, но  $x$  - не захвач.

$[=, \&x]$

146

Захватывает только одинаковые  
символы разрешенные

```

int f() {
    static int y;
    int x, z;
    int x;
}

```

~~$[\&x, z]$~~  ()  $\downarrow y = 1; y()$

• Захват  $\circ$  не разрешено обратно

$\text{int } x;$

$[\&\text{pivot} = x]$  () {} ;

~~$[\&\text{pivot} = x]$~~  ()  $\downarrow y$   $\times = \text{pivot} - \text{символ разрешен}$

$[\&\text{ptr} = \&x]$  () {} ;  $\text{ptr} = \text{адрес } x$

~~$[\&\text{ptr} = \&x]$~~  () {} ;

символ на адрес  $x$  (запрещено одеск)

## Функции

- Функция - обьект, который может хранить кода на функции (operator())

auto f = func;

f(5); Функция

struct Sqr {

int operator()(int x) const noexcept {

return x \* x;

y

main Sqr sqr;

sqr(5); sqr(x)

std::set<T, Compare> std::sort(v.begin(), v.end());

Compare  
std::less<T>

Compare  
std::less

<

#include <functional>

std::greater<T>

std::equal

struct Compare {

bool operator()(string x, string y) const {

return x.size() < y.size();

y

объект

меню std::sort(v.begin(), v.end(), Compare());

std::set<string, Compare> t;

## Lambda expressions

int main() { [ ](); }

std::sort(v.begin(), v.end(), [ ](const string& x, const string& y) { return x.size() < y.size(); })

написанные  
забывчивости

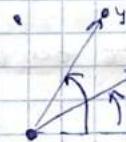
составные  
аргументы  
closure  
и оператор()

У каждого lambda-expression свой тип:

type([ ]() { return s; }) ≠ type([ ]() { return S; })

[...]

lambda  
выражение



sort(&v, [ ](Point x, Point y) {  
return angle(x, P0) < angle(y, P0);});

Struct Closure {

Point P0;

-operator()(x, y){...}

May  
14 05

14 0570

Unordered - set < T, Hash, Equal >  
map < key, value >

$\text{set} = O(\log n)$  requires more

unordered - O(1) & ~~expensive~~  
    \ see replace & propagate

Xem - radanya	0 1 2 3	99
{0, 1, ..., 99}	<u>1 0 0 0 0</u>	0 0

↑, em ecto reas odd 3  
0, test reas

Diagram illustrating the memory representation of floating-point numbers:

Diagram illustrating the memory representation of floating-point numbers:

Memory layout:

- Address 0: 1
- Address 1: 1
- Address 2: 1
- Address 3: 2
- Address  $10^6 - 1$

запись  $\approx 10^6$

$x = m \cdot 10^e$

$$h("aba") = \text{id}_X$$

$$\text{max\_load\_factor}(k) = \max\left(\frac{n_{\text{elem}}}{n_{\text{sockets}}}\right) = k$$

$$\text{load-factor} = \frac{n_{\text{elem}}}{n_{\text{buckets}}}$$

reserve - браков из Н гроб на съвет непреклонен

rehash(k) - ycreate buckets na buckets = k

Bucket\_count - n-buckets

`Bucket_size(i) = k0-k0_fn-06` & `Bucket[i]`

если  $\frac{n_{-elem}}{n_{-Баркет}} = k \Rightarrow$  ПЕРЕВЫДЕЛЕНИЕ

## • Structured Bindings (C++17)

for (const std::pair<const int> &x : m) {

count < x, first < x, second

for key, value in m.items():

8

for (const auto & [key, value] : m) { } konsiderera

размер массива  $\rightarrow$  array  $\text{int } x[3]$   
допускается использовать  
точку вместо запятой

42) Класс, о котором  
все НЕСТАРИ находятся  
и НЕГРАДЕЙКАЮТСЯ

↳) Even greater efficiency  
by reusing the std::tuple::size  
tuple::element

$\Rightarrow$  std::get<T>(x), modo uno o  
x.get<T>()

28.04.20

## СЕМИНАР

Point {

double x\_;  
int<sup>и</sup> y\_;

double y\_;  
int<sup>и</sup> x\_;

};

Все корректно, коечко не хотят синтаксис  
в своих глазах, языке синтаксиса разных  
языков

Unique ptr      p → █

T\* release(); // создает указатель на один  
при этом предыдущий указатель указывает  
на него показывает на nullptr.

reset(\$ptr); // теперь указатель y-го create.  
одинаковый указатель, указывает ptr

get();

operator bool // приведение к bool

Пример C'

class A {

public:  
A() = default

explicit A(int) {}

void f() {}

~A() {}

std::cout << "A destructor call";

};

};

std::unique\_ptr<A> ptr = std::make\_unique<A>();

ptr

auto ptr = std::make\_unique<A>();

make\_unique<A>()

g(std::unique\_ptr<A>(new A), std::unique\_ptr<A>(new A))

new

неизвестно, то работает багом  
последнее => если при выделении  
памяти багом обработано илен., то  
указатель

reference

=> регистрируется make\_unique

int main()

std::unique\_ptr<A> ptr1(new A)

// a destructor call

std::unique\_ptr<A> ptr2(ptr1)

// CE

std::unique\_ptr<A> ptr3(

std::move(ptr1))

// OK

s

2

std::unique\_ptr<A> ptr = std::make\_unique<A>();

ptr

3

auto ptr = std::make\_unique<A>();

make\_unique<A>()

make\_unique<A>()

g(std::unique\_ptr<A>(new A), std::unique\_ptr<A>(new A))

4

new

неизвестно, то работает багом  
последнее => если при выделении  
памяти багом обработано илен., то  
указатель

reference

=> регистрируется make\_unique

Voo

template <class T>  
class UniquePtr {  
 T\* ptr\_;

public:

UniquePtr(): ptr\_(nulptr){}  
explicit UniquePtr(T\* pte): ptr\_=pte{}

~UniquePtr();

delete ptr\_;

UniquePtr(const UniquePtr&) = delete;

UniquePtr& operator=(const UniquePtr&)=  
delete;

UniquePtr(UniquePtr&& other);

ptr\_=other.ptr\_;

other.ptr\_=nulptr;

y

UniquePtr& operator=(UniquePtr&& other){  
if (this == &other) return \*this;  
}

delete ptr\_;

ptr\_=other.ptr\_;

other.ptr\_=nulptr;

return \*this;

T\* Get() const { return ptr\_; } 98

T& operator\*() const {  
return \*ptr\_;}  
y

T\* operator->() const {  
return ptr\_;}  
y

Базовый класс реагирует, когда  
ее определяют "членом"  
указателя

struct S {  
 int value;  
}; class Y;  
main:  
UniquePtr<S> a  
(new S)

\*a = h & y;  
a->value;

// a.operator->().operator->(operation),  
value

### #include <memory>

std::unique\_ptr<T> a(new T);

std::unique\_ptr<S> b = std::make\_unique<S>(5, 6.0)

new S(5, 6.0)

- std::shared\_ptr;  
Без указателя с фиксированным  
объектом (безик)
- std::weak\_ptr

set(CMAKE\_CXX\_FLAGS \${CMAKE\_CXX\_FLAGS}  
-fno-exceptions") // отключение  
контрактов C++ 11

✓ forwarding references (граверинг  
рэфэрэнсес)

23.04.20

## Smart pointers

- Проблемы

1) int main(){

    int \*ptr = new int;

    ...  
    delete ptr;

}

Ручное управление

2) void f(){

    int \*ptr = new int;

    g(); h(); f(); // → exception

    delete ptr;

}

Единственный указ.



if

try

h();

catch()

R A I I template class T>

class SmartPointer {

    T \*ptr\_;

public:

    ~SmartPointer();

    delete ptr\_;

    ...

};

main:

{  
    SmartPointer<int> a(new int);  
}

? SmartPtr<int> a(new int); ?

? Smart Ptr<int> b(a); ?

- auto\_ptr

→ Нужен ли классический auto\_ptr

A(A& other)

(использование, когда  
есть более строгие  
семантики)

• std::unique\_ptr (единственный владелец)

a →

⊗ ~~автоматическое уничтожение  
(delete)~~

Можно использовать:

a = std::move(a)

a → → a →   
a → → a →

```

32
Пример class Info {
    string first_name_;
    string second_name_;
    std::vector<string> goods_;
```

public:

Info (string first\_name, string second\_name,

~~std::vector<string> goods\_;~~:

first\_name\_(move(first\_name)),

second\_name\_(move(second\_name)),

goods\_(move(goods\_));

}

Info info1(first\_name, string("alpha"), good);

один раз  
использовать

в качестве  
first\_name

0 раз конструктор,  
т.к. в String  
есть проблема с  
(они есть копии строк  
без открытия)

Если объект register не имеет геттеров  
(нет открытых методов) → const T &

Если объект register имеет открытые методы  
или открытые методы есть копии, → T &

Создание копий

- ✓ Copy elision - оптимизация компилятора
- A a = A(5); //Parametric constructor ~~copy elision~~
- A a(A(5)); //Parametric constructor copy elision
- A b(a); // Copy constructor
- A b = move(0); // move constructor
- d = A(5); //Parametric constructor, move ~~assigment~~

A e = f(); //Parametric constructor RVO

A f = g(); //Parametric constructor NRVO generated object

A f() h returns 5; g

RVO - это оптимизация

//RVO(return value optimization)

A g() h A o(5);  
return o; g

//NRVO(normal RVO)

5) template < class T >

```
void OldSwap(T& x, T& y){  
    T temp = x; x = y; // конвертация O(n)  
    x = temp; // конвертация O(n)  
    y = temp; // конвертация O(n)}
```

template < class T >

```
void NewSwap(T& x, T& y){  
    O(1) T temp = std::move(x); // передача move  
    O(1) x = std::move(y); // передача move  
    O(1) y = std::move(temp); // передача move  
    // один раз  
    // время = O(1)
```

Репозитории специализации:

Stack operator = (Stack & other)

if (this == &other) { referenc \*this; }

```
std::swap(buffer, other.buffer_);  
std::swap(capacity_, other.capacity_);  
std::swap(size_, other.size_)
```

Stack & operator = (Stack & other) { [конвертация]  
[перенос в один раз] [конвертация]

Swap()

referenc \*this;

}

В данном случае

конвертация

использование

одноразового move

оператора assignment,

move не operator.

Изменение of Push(): Stack < vector<int> >

При использовании vector вagine move, or  
или конвертацию.

```
void Push(T& value){ // перегрузка Push  
    if (не константа) {  
        Expand();  
    }
```

buffer\_[size\_ + 1] = std::move(value);

9

21.04.20

## CEMUIHAP

int x; ~ ~ ~

int & y = x; // lvalue ссылка, memory changes

c lvalue одесканси, то

НЕ memory changes c rvalue

const int & cy = 5; // OK

cy = 9; // CE

template <class T>

void g(const T& x); ~ ~

g(x); // OK

g(5); // OK

int && rv = x; // CE, r.r. rvalue ссылка

также ссылка с  
rvalue ссылкой

int & t z = 8;

Напишите

void f (int& x) {  
std::cout << "lvalue\n";

}

void f (int& x) {  
std::cout << "rvalue\n";

}

replace with acceptation top:

Stack (Stack & other)

: Buffer\_(other.Buffer\_),  
capacity\_(other.capacity\_),  
size\_(other.size\_);

other.Buffer\_ = nullptr;

other.size\_ = 0;

other.capacity\_ = 0;

~

f(x);  
f(5);

f(std::move(x))

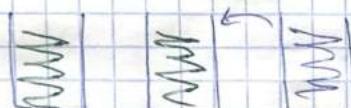
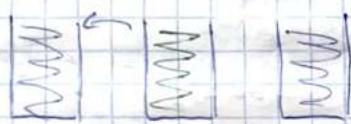
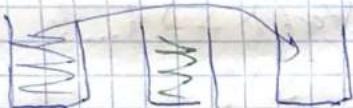
уникальные  
специфичные одесканси  
или все специфичные

other не изменяется  
Была ли ошибка

this => other // CE, T.

то g & this g \* e запрещено  
использовать

Stack<int> z(z); // OK



`std::move` - ~~copying~~, ~~no copy overhead~~  
object  $\&$   $\neq$  `rvalue` ( $\approx$  `static_cast<T&>`)

`std::move(x)` - ~~copying~~, ~~no copy overhead~~

`void Swap(T& x, T& y){`

$T \text{ temp} = \text{std::move}(x); \quad \text{ba } O(1)$

$x = \text{std::move}(y); \quad \text{ba } O(1)$

$y = \text{std::move}(\text{temp}); \quad \text{ba } O(1)$

}

`const Stack x = ...;`

`Stack y = std::move(x);`

`std::move(const) \Leftrightarrow const T& (kontraposition)`

`int & x = temp;`

`int & y = s;`

`int & z = x / / CE`

~~`z = y / / CE,`~~  $\star$   $y$  - referenced

`int & z = x / / OK`

`z = y / / OK`

`rvalue context -> free space  
object`

✓ copy/move and swap idiom

`Stack& operator=(Stack s){`

`std::swap(header_, s.header_);`

`return *this;`

}

} kontraposition  
referenzieren

85

✓ Move constructors and assignment

`Stack(const Stack& other){}`

`size_ = other.size_;`

`...`

`other → [ ] → [ ] → [ ]`

`O(n)`

`y`

`Stack x(y);`

`Stack z(f());`

`f() - copy-const operator object`

`object`

Xorreecc ob:

`Stack (Stack& other)`

`size_ = other.size_;`

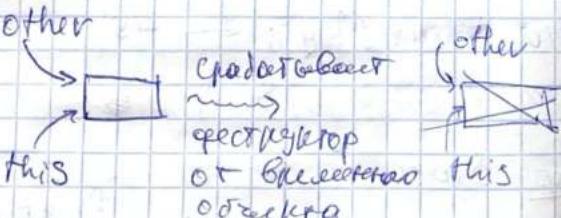
`head_ = other.head_;`

`other.size_ = 0`

`Boxed!`

`other.head_ = nullptr; O(1)`

`y`



`stack & operator=(const Stack& s) { ... }`

`Stack & operator=(Stack& s) {`

`std::swap(size_, s.size_)`

`std::swap(head_, s.head_); O(1)`

`y`

✓ Rule of 5

Если есть конструктор копирования (хотя бы один)

то p конструируется, опер-т копируется (конструируется),

копирование, move k-p, move =,

то можно реализовать все 5.

`y`

Если бы отсутствовал хотя бы один из них, то move бы стал единственным для копирования

= default

= delete

✓ std::move

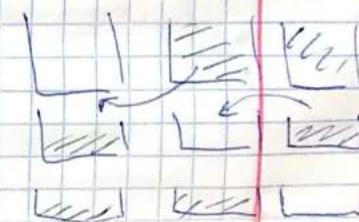
`void swap(T& x, T& y){`

`T tmp = x;`

`x = y;`

`y = tmp;`

`y`



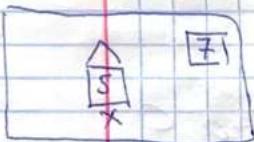
83)

## Move semantics

Использование временного объекта

• **Lvalue / Rvalue** (`Xvalue, prvalue, qvalue`)

- Lvalue - именное выражение, памятью которым которого хранится в постоянной ячейке
- наименование



`int x = 5; - lvalue`

`7; // не lvalue`

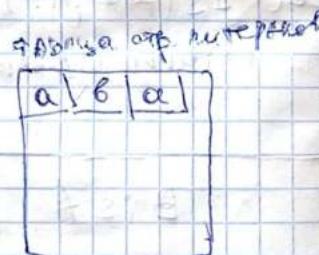
- перемещение (операторы)
- ссылки (в частности возвращаемые функции)

`int& f(); f() = 5; ++x`

`z = 4; //OK`

- стековый контейнер

`"aba"` - Rvalue



- Rvalue - все остальное  
(операторы для материалов)

## ✓ Lvalue / Rvalue ссылки

`type & x = ...;` - Lvalue ссылка может ссылаться на значение с rvalue

`int & z = g(); //OK int & z = 5 //CE int & g()`

`const int& cref = 5; //OK`

Rvalue ссылка может ссылаться только на Rvalue

`int && x = g(); //CE`



`int && z = 5; //OK`

`int && t = f(); //OK int f()`

`int && a = g(); //CE int & g()`

`z = 4; //OK`

`void f(int& x); ← f(x) непрерывное копирование`

`void f(int& y); ← f(s)`

`void f(const int& x); //OK`

Пример: Stack:

`Print(const Stack& s);`

`Stack copy = s; // копия`

`while (!s.empty());`

`z --`

`Print(Stack(5)); - копия`

`объекта`

`Print(Stack& s);`

`на временный объект`

39)   
Dynamic cast <sup>недоработано</sup> <sub>использование (virtual)</sub>  
struct A {  
 virtual ~A() = default;

A a;  
B b;

struct C : public A {

B\* B\_ptr = &a; //CE

B\* B\_ptr = ~~reinterpret~~ cast<B\*>(&a); //OK

агрессивное управление памятью, т.к. не используется деструктор  
а методом конвертации

B\* B\_ptr = static\_cast<B\*>(&a); //OK

агрессивное управление памятью, т.к. не используется  
конвертация

выводится то же значение

B\* B\_ptr = dynamic\_cast<B\*>(a\_ptr); //CE,

т.к. A - не polymorf

зато все же virtual ~A() => A - полиморфист

A: → AAAA  
B: → AAAABBBB

if (dynamic\_cast<C\*>a\_ptr) B

cout << ...;

else

cout << ...;

надо ~~созерцать~~ <sup>использовать</sup> глядя на исходник C,  
надо ~~созерцать~~ <sup>использовать</sup> new\_ptr.

template <class T>  
bool IS(A\* ptr){  
 cout << IS<B>(a\_ptr);  
 return dynamic\_cast<T\*>(ptr);  
}

B is A

C is A

A is not B, ~~C~~ A is not C

try {  
 C\* c\_ref = dynamic\_cast<C\*>(\*a\_ptr);  
} catch (std::bad\_cast &){  
}

29)

Виртуальные члены класса

Обычное наследование.

struct A {

void f() { cout << "A\n"; }

}

struct B : private A {

void f() { cout << "B\n"; }

void g() { cout << "g\n"; }

main: A aa = B; // корректно через g,  
но private  $\Rightarrow$  CE

struct B : public A { }

main: A aa = B //OK

aa.f(); // A::f

aa.g(); //CE

A a;

B b;

A\* a\_ptr = &B; //OK

B\* B\_ptr = &a; //CE

a\_ptr  $\rightarrow$  f(); // A::f

a\_ptr  $\rightarrow$  g(); //CE

Виртуальное наследование: (virtual)

A: virtual void f() { ... }

B: void f() override { ... }

void g() { ... }

A  
main: A aa = B;

aa.f() // A::f

aa.g() //CE

A\* a\_ptr = &B;

a\_ptr  $\rightarrow$  f() // B::f

a\_ptr  $\rightarrow$  g() //CE

f - переопределение  
 ↓  
 дубл. наследование  
 f() в B

struct A {

explicit A(int) { cout << "A\n"; }

};

struct B : public A { }

main: B b;

Koress

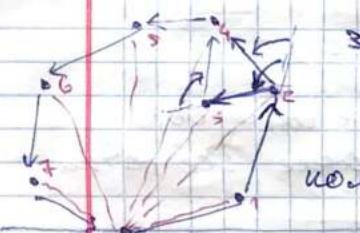
Конструирование производится A  $\rightarrow$  B  $\rightarrow$  C...

если создать B, создается A. То есть создание  
 A в default  $\Rightarrow$  конструирование (если не указать явно),  
 а потом создать производное B.

Алгоритмика:  $O(nk)$ ,  $k$  - число точек в  
многограннике

$O(n^2)$  - в худшем случае

- Graham scan | stack:  $\text{Top}(-1)$
- 1) Выбираем  $p_0(\min y)$
- 2) Сортируем все точки по величине  
коэффициента угла относительно  $p_0$ .



3) Пытаемся добавить  $p_i$ :

Если  $\vec{p}_{i-1}\vec{p}_i$  и  $\vec{p}_i\vec{p}_{i+1}$  однозначно

изменяют направление, то добавляем  $p_{i+1}$   
иначе: удаляем точку из списка

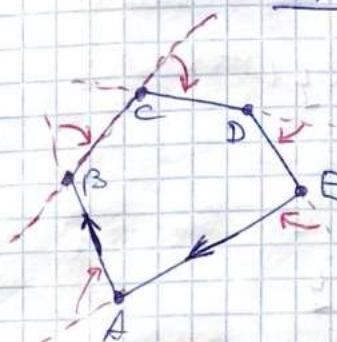
- 4) до 3) пока есть точки - max 2n операций  
(Все добавили + Все удалили)

Алгоритмика:  $O(N \log N)$

$$O(N) + O(N \log N) + O(N) = O(N \log N)$$

Доказательство низкого:

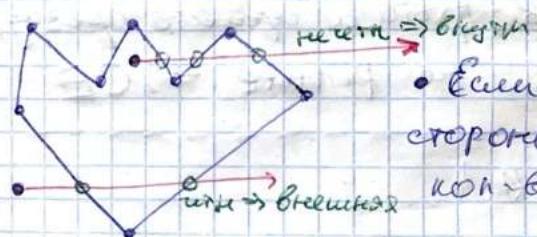
## СЕМИНАР



Быстроизбыточность многоугольника:

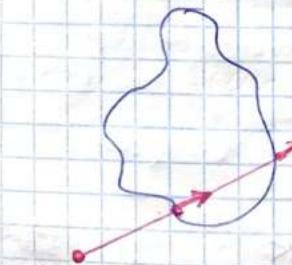
Если  $[\vec{AB}, \vec{AC}], [\vec{BC}, \vec{CD}], \dots$  одного знака, то многоугольник (перекрывается сама себя в одну сторону)

Точка принадлежит многоугольнику



- Если луг из точек пересекает стороны многоугл.  $\Rightarrow$  нечетное кол-во раз  $\Rightarrow$  Точка Внутри

- Если четное кол-во  
Точка снаружи



(see th Tanya)

75)

точка принадлежит отрезку  $[AB]$



$$[AP, RB] = 0$$

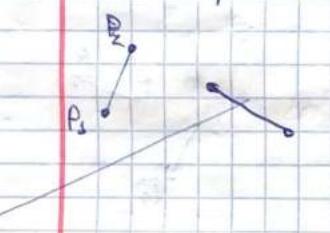
$$[AP, AB] \geq 0$$

точка принадлежит отрезку?



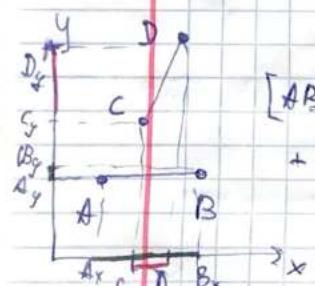
$$P \in [AB] \Leftrightarrow P \in [AB] \cup P \in [BA]$$

Пересечение прямой и отрезка  
(глобик отрезков)



$$\text{Если } [AB; AP_1], [AB, AP_2] \text{ отрезки}$$

то  $P_1 = P_2$   
тогда,  
то  $P_1 \neq P_2$  и отрезок не содержит



$$[AB] \cap [CD] \Rightarrow [AB] \cap (CD) \cup [CD] \cap (AB)$$

+ краевые условия

либо самое промежуточное  
норманди  $\in AP_1 \cup AP_2$

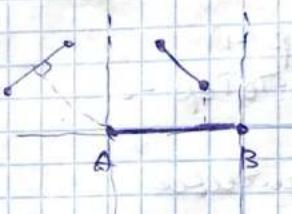
Последовательные сдвиги отрезков



1) Если  $A \neq B$ , то  $P=0$

2) Если  $A = B$ ,

$$P = \min(P(A, [CD]), P(B, [CD]), P(C, [AB]), P(D, [AB]))$$



Помощь в вычислении отрезков

Минимальное значение  
самоотрицательное значение

• Jarvis scan (Gift wrapping algorithm)



1)  $P_0$  - левая вершина ( $\min y$ ).  
Если не одна, то  
если и самую правую.

2)  $P_i \rightarrow P_{i+1}$ : минимум по  
относительно  $P_i$

3) Пока  $P_{i+1} \neq P_0$

Д-ко суперпозиции не верн.:  $P_0 \in S$ . Все точки  
для них синий

33  
`sfd::tuple<int, double> Func()
 return {5, 5.0};`

4

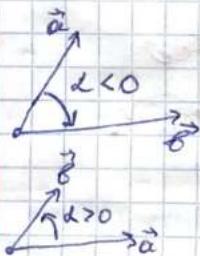
`int main()`

```
auto x = Func();
std::get<0>(x);
```

Векторное произведение

на плоскости

$A(x_1, y_1)$   $B(x_2, y_2)$   $\vec{AB} (x_2 - x_1, y_2 - y_1)$



Ортогональный угол

(векторное произведение

$$(\vec{a}; \vec{b}) = a_x b_x + a_y b_y = |\vec{a}| |\vec{b}| \cos \varphi \quad \cos \varphi = \cos(-\varphi)$$

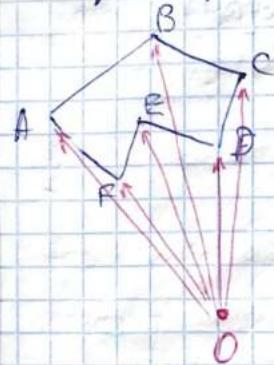
5

74  
 векторное произведение

$$[\vec{a}; \vec{b}] = \begin{vmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = 0 = (a_x b_y - a_y b_x) \vec{k}$$

$$|\vec{a}| |\vec{b}| \sin \varphi$$

Ортогональный угол



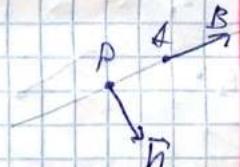
$$S_{ABCDEF} = \sum \text{огранич. подынеги}$$

Прическа на плоскости

$$ax + by + c = 0 \rightarrow n = (a, b)$$

$$\vec{OP} + t \cdot \vec{AB} \quad (n, \vec{AB}) = 0$$

$$t \in [0, 1]$$



Тогда при什么样的 условиях?

$$[\vec{AP}; \vec{AB}] = 0$$

поставить  $P_x, P_y$  &  $y_P$  в правиль

template <class T, class U, class... Args>

struct IsSame {

static const bool value =

= IsSame<T, U>::value &&

IsSame<U, Args...>::value,

}

template <class T, class U>

struct IsSame<T, U> {

static const bool value = false;

}

template <class T>

struct IsSame<T, T> {

static const bool value = true;

}

std::cout << IsSame(int, int, int, int)

Компилятор угадал типы "демонстрируя  
сопроводитель" (если не ошибся в определении  
Более того T)

template <int X, int...>

template <int... Values>

struct Sum;

// Обработка

template <int X, int... Values> *искусственный*  
struct Sum<X, Values...> {

static const int value = X + Sum<Values...>::  
value;

}

template <> *искусственный* от  
struct Sum<> {

static const int value = 0;

}

Для каждого объекта есть

исключительно один изображение, но

могет изображение повторяться.

std::cout << Sum<3, 4, 5>::value;

Обработка & трансляция  $\rightarrow$  class Sum<int,int,int>

исключительно время  $\leftarrow$   $\Rightarrow$  <int>

исключительно, но не время  
исключительно радионет сопоставление

63)

```

class A {
    int x_;
    std::string y_;
    int z_;
public:
    explicit A(int x=0, const std::string& y="",
               int z=0) :
        x_(x),
        y_(y),
        z_(z) {
        std::cout << x_ << y_ << z_;
    }
};

class B {
    A a_;
public:
    template <class... Args>
    explicit B(const Args... args): a_(args...) {
    }
};

B b(1, "aba", 2)

```

Применяется `emplace_back` в `std::vector`

### Метаупрограммирование

```

template <class T>
IsConst<T>::value
void printConst() {
    if (T is const) {
        std::cout << "const\n";
    } else {
        std::cout << "not const\n";
    }
}

```

неблагодар

Унарная операция,

```

template <class T>
struct IsConst {
    static const bool value=false;
}

```

```

template <class T>
struct IsConst<const T> {
    static const bool value=true;
}

```

`std::cout << IsConst<int>()`  
`std::cout << IsConst<const int>()`

07.04.20

## CEMURAP

template < class ... Args >  
const  
type Min(Args&... args){  
 odptia  
 trueq  
 reiam  
 no konkratetem  
 obecne

Typical min quiz & non-RA object members  
~~template < class T, class ... Args >~~  
~~T Min (const T & value, const Args&... args){~~  
 ~~first~~  
 ~~const T& second,~~  
 ~~T min\_value = (first < second)? first : second;~~  
 ~~class T~~

template < class ... Args >  
void Print(const Args&... args)  
 const T& value  
 std::cout << value << ' ';  
 Print(args...);

}

void Print(){  
 std::cout << "hi";

}

Args uvozeti sa prekaze, a da ne pozivaju  
bez naziva funkcije

Print(true, 4, "abc")

template < class T >  
T Min (const T & value){  
 return value;

}

template < class T, class ... Args >  
T Min (const T & first, const T & second,  
 const Args&... args){

T min\_value = std::min(first, second);  
return Min(min\_value, args...);

}

// args...  $\Leftrightarrow$  args[0], args[1], ...  
// pattern(args)...  $\Leftrightarrow$  pattern(args[0]), pattern(args[1])

65

### ✓ Синхронизация исключений

```
void f() { / void g() noexcept {
    g();           throw x;   } // "глобальная функция
    y;           // не дробит исключения"
    warning
    terminate - если все-таки будет
```

### ✓ Гарантии безопасности

try  
v. Push Back(1)

{catch(...){

y

- Гарантия очищивания исключений (nothrow)

- Справедливая гарантия безопасности  
(одна из функций не используется именем ...)

- Базовая гарантия безопасности  
(одна из функций содержит, но не использует)

66

### Гарантии безопасности исключений

```
struct A {
    int x;
    double y;
};

int main() {
    A a{1, 3};
    y
}
```

```
class A {
    std::string s(100, 'a');
public:
    A() {}
```

OK

explicit A(int) {}

A(int, double) {}

y

A Func(A a) {

Func(A(1, 2)) => Func({1, 2})

Задача решена  
использованием

not main()

A a(); // так нечестно, можно: A a{};

A b();

A c(1, 3);

A d(1, 3); // A d(1, 3) - нечестно

A e(1, 3); double -> int

A f(1, 0) // OK

Rational r(2, 3);

return {2, 3};

26.03.20

## Чеканка исключений

При работе с исключениями все происходит  
искусственное туннелирование (норма)

```
void f() {  
    throw int;
```

```
}
```

```
main() {
```

```
    try {
```

```
        f();
```

```
    } catch (double) { ← некорректно
```

```
    } catch (int*) { ← не ожидается  
        номинально
```

```
}
```

```
OK
```

Чеканка из проблема - исключение  
class A;

```
class B : public A {
```

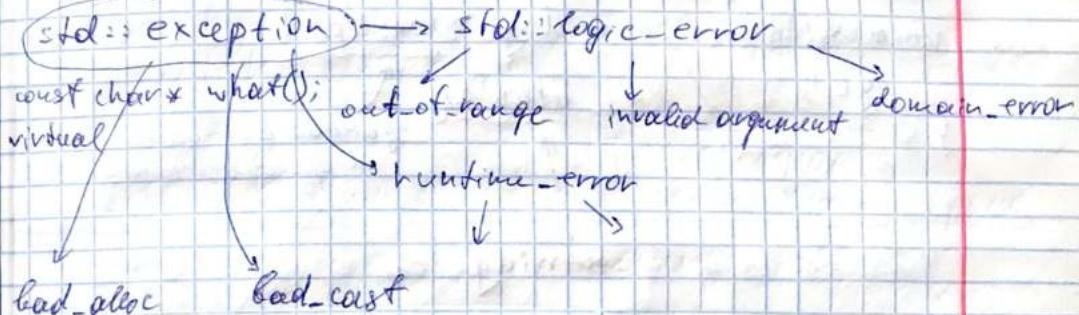
```
    void f() {
```

```
        throw B();
```

```
}
```

```
main() { try {  
    f();  
} catch (A a) { ← OK, B наследует A  
} catch (B b) { ← то что ожидаем
```

## Некорректные исключения



```
void f() {
```

```
}
```

```
main()
```

```
try {  
    f();
```

```
} catch (exception ex) {  
    cout << ex.what();
```

Быстро!

✓ `bad_alloc` проблемы, even new T[...]

`int * ptr = new T[100];`

закрывает  
исключение

`int * ptr = new (std::nothrow) T[100]`

✓ `bad_cast`



`B b;`

`A * a = &b`

`B * B_ptr = a; // CE`

`B * B_ptr = dynamic_cast<B*>(a);`  
even a user man B\*, то OK, user will ptr

24.03.20

## CEMIIHAD

### #include limits

- ✓ Численческое значение на 0 - UB
  - ✓ floatовое значение на 0 - negative inf
  - ✓ 0/0 = nan
- class A {  
 ...  
};

Template < class T >

T Divide(T x, T y) {

if (y == 0) {

throw A();

...  
};

return x/y;

}

double sumAndDivide(double x, double y, double z)

double result;

try {

result = Divide(x+y, z)

} catch (A a) {

return std::numeric\_limits<double>::infinity();

}

try {

y catch (double) {

y catch (int) {

...  
};

y

Если оно int, то  
тое же приведено к  
double

**Также торое  
содержание!**, кроме

наследников

A

↑

catch(A)

= наследие A и B

Если это нечто иное  
⇒ circumstances, то обеих HF создает  
⇒ регистрирует все обеих Ошибки, оставленные

Использование в деструкторах запрещено, т.к.  
они уже не могут отработать.

Неправильное использование

std::exception

excepting what()

class Exception

#include exception

public

class A : public exception {

std::string error;

public:

A(const std::string& error){

error = error.c\_str();

Быстро!  
редко подразумевается

try { ... } excepting what()  
catch (std::exception& ex){

}

Пример: void f() {

```
    int *arr = new int[100];
```

```
    try {
```

```
        g();
```

```
    } catch (...) {
```

```
        delete [] arr;
```

```
    } throw; ← предполагается, что оно не
```

```
catch (std::string & x) {  
    throw x; // исключение  
    throw; // исключение
```

✓ Исполнения в конструкторах

```
class A {  
public:  
    string x;  
    B b;
```

```
    main() {
```

```
        A a; // A()
```

```
        return 0;
```

```
    A() {  
        ptr = new int();  
        throw i; ←
```

```
    } try { ptr = new int();
```

```
        throw i;
```

```
    } catch (...) {
```

Но где создается конст., если в конст-те  
ничего не написано, исключение возникает!

✓ Исполнения в деструкторах

Проблема: Никогда не происходит исполнения в деструкторах

```
class A {  
public:  
    ~A() {  
        throw l;  
    } }
```

```
void f() {
```

```
    A a;
```

```
    throw "error";
```

```
↓ [1] ["error"] ↓
```

2 исполнения std::exit() → abort()

аварийное завершение  
программы

## Механизмы

✓ void f() {

    if (Error)

        g();

    else

        h();

    }

    void g() {

        if (Error)

            return 10;

        else

            kog\_outska

    void g() {

        if (error)

            throw 1/True;

    void f() {

        g(); ← Exc.

    } ← exc. в main

    nextree rask:

    void f() {

        try {

            g();

        } catch (int x) {

            std:: cerr << "Caught int" <<

        } catch (string) {

        } catch (...) {

Если ошибка не обработана

то terminate();

    ↑  
    упорядочен

    if (Error)

        g();

    else

        h();

    }

    }

## ✓ РАСПЫТКА СТЕКА (обратная вызов)

void h() {

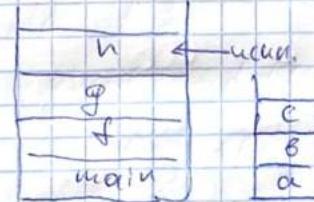
    A a;

    B b;

    C c;

    throw d;

    }



    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

    }

19.03.20

## Häufungsbauweise



Vector<Furniture\*> CreateCart (User & u) {

    Vector<Furniture\*> cart;

    if (...) {

Furniture f = Sofa();  
 cart.PushBack(Sofa(1))  
Sofa  
 cart[0] → Order() // Furniture :: Order

    }

zusammen

Void Order::f (vector<F\* > cart) {

    for (int i=0; i < cart.size(); ++i) {

        cart[i] → Order();

        ...

        if (i == 0) Sofa

        if (i == 1) Bed

y

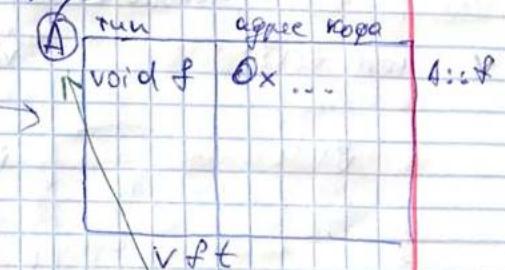
## ▼ Traditionelle Implementierung von Methoden

class A:

    virtual void f();

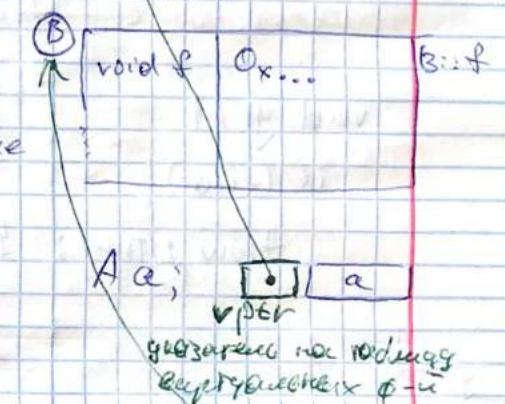
class B: public A {

    void f() override;



## ▼ Multiklassenhäufung Häufungsbauweise

...



B b; [ ] | A | [ ] B/f

A \* a; aptr = & b;

a->ptr → f(); // B::f()

53  
17.03.20

## СЕМИНАР

10<sup>8</sup> операций - 1 sec

Компилятор = стандарт + доп. функционально

```
class A {  
public:  
    void f();  
    virtual void g();  
};
```

```
class B : public A {  
public:  
    void f();  
    virtual void g();  
};
```

```
B b;  
b.f() // B::f  
b.g() // B::g  
  
A a = B; // OK - срезка  
a.f(); // A::f  
a.g(); // B::g
```

```
A* a_ptr = &B;  
a_ptr->f(); A::f  
a_ptr->g(); B::g
```

```
A& a_ref = B;  
a_ref.f(); // A::f  
a_ref.g(); // B::g
```

```
#include <iostream>  
#include <exception>  
#include <string>
```

```
class DivisionByZero : public exception  
public:  
    const char* what() const noexcept override  
    return "In function...";
```

int Divide(int x, int y){  
 if (y==0){  
 throw DivisionByZero();

y  
return x/y;

y  
int main(){  
 Divide(1, 0)

~~DivisionByZero~~

- Почему же не сработало ~~throw~~ с конструктором исключительного объекта?
- Где же все-таки были ошибки!  
(ссылка на ~~инспектор~~)

5)

`dynamic_cast` — (), но (с) проверкой корректности

`dynamic_cast<B*>(a_ptrB)) // OK.`

`dynamic_cast<B*>(a_ptrA); // вызывает nullptr  
т.к. некорректно!`

`static_cast<B*>(a_ptrA); CE`

`dynamic_cast<B*>(* a_ptrB) // OK`

`dynamic_cast<B*>(* a_ptrA) // ИСПОЛЬЗУЕТ  
(инвертирует)`

Пример



```

void Print(const Stack& S) {
    StackMin * Sm = dynamic_cast<const StackMin*>(S);
    StackMax * Smx = dynamic_cast<const StackMax*>(S);
    StackNone * Sn = dynamic_cast<const StackNone*>(S);

    if (Sm) {
        std::cout << "StackMin";
    } else if (Smx) {
        std::cout << "StackMax";
    } else {
        std::cout << "Stack";
    }
}
  
```

✓ АБСТРАКТНЫЙ КЛАСС

`class Furniture {`

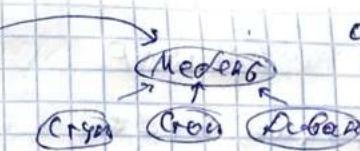
`public:`

`virtual int price();`

`virtual int Year() = 0 // Чисто виртуальный метод`

`virtual std::string Publisher();`

`}`



Общая логика  
создает новый

52

(pure virtual method)

Это метод, где которого  
не определено никак  
использовать

Этот класс содержит хотя бы  
один чисто виртуальный метод,  
поэтому является  
абстрактным.

Этот класс содержит НЕ виртуальный  
чисто виртуальный метод, то  
есть здесь СТАНОВИТСЯ  
АБСТРАКТНЫМ

## ✓ Конкретные возвращающие функции

struct A {

virtual int f() { ... }  
};

ТАК НЕЛЬЗЯ, потому что не может быть конкретных возвращающих функций.

struct B : A {

    double f() override { ... };

};

(A)  $A^* \rightarrow B^* \rightarrow C^*$

(B)  $A^* \rightarrow B^* \rightarrow C^*$

(C) A:  
    virtual A\* f() { ... }  
B:  
    B\* f() { ... }

## ✓ Виртуальный деструктор

struct A {

    int \*a = new int;

    virtual ~A() {  
        delete a;  
    }

};

struct B : A {

    int \*b = new int;

    ~B() {  
        delete b;  
    }

};

~~для~~ virtual

мак4: A \* a\_ptr = new B; // вынужден вызвать ~A() =>  
                                  // => деструктор наследника B...

virtual ~A() => Ok. Компилятор заменяет ~B(), т.к.  
a\_ptr указывает на A.

## ✓ ПРАВИЛО: В конструкторах запрещается использовать

struct A {

    virtual void f() { ... }  
};

A() {  
    f()  
};

struct B {

    void f() override;

    B() { ... };

    f(); // запрещено вызвать B::f()

};

## ✓ ~~dynamic\_cast~~ dynamic\_cast

использование динамического - с возвращением

B B;

A \* a\_ptr = &B; // Ok

B \* b\_ptr = a\_ptr; // Err

(A)

(B)

A \* a\_ptr = &a; // Ok

B \* b\_ptr = static\_cast<B\*>(a\_ptr);

Быть приблизительно гипотезой, что можно доделать класса к употреблению  
и особенно на уровне инфраструктуры, то есть ~~здесь~~ пределом корректности

12.03.20

## 1) Encapsulation

public  
protected  
private  
  
"cooperativ"

```
struct A {
    int a;
    void g();
};

public
struct B: A {
    int b;
    void f();
};
```

```
struct B {
    private:
        A a;
        public:
            void g();
};

struct B: private A {
    private:
        A a;
        public:
            void g();
};
```

## Наследование II

```
class Stack {
    int *data;
    size_t size;
public:
    virtual void push(int val) {
        data[size] = val;
        ++size;
    }
};
```

```
class StackMin : public Stack {
    int *min_data;
public:
    void push(int val) {
        min_data[size] = size == 0 ? val : min_data[size - 1];
        Stack.push(val);
    }
};
```

```
int main() {
    StackMin s;
    s.push(1);
    s.push(2);
    f(s); // OK, OK
```

```
void f(Stack s) {
    s.push(3);
}

void f(Stack &s) { // upper min_data
    s.push(3);
}
```

- Виртуальность несет, что наследник наследует базовую
- Определение в базовом классе - базовый подразумевается в производственном базовом
  - Виртуальность несет - несет, несущийся, которое соответствует наименование базового.
- Virtual*  $\Rightarrow$  "s.push(5) // s.StackMin::push(5)

✓ Краткое изображение override и final

```
struct A {
    virtual void f(int);
};

struct B: A {
    void f(double) override; // final
};
```

A.f(5) // f(1.5), т.к. f(int) более  
присущее для  
с финальной

номера в наследнике,  
тогда подразумевается наследник  
базового класса.

Если такого нету  
то подразумевается базовый

Базовый  
присущий  
наследник  
struct B: final  
||  
or B должна  
иметь наследника

45

class SquareMatrix : public Matrix {

public: det();

4

BigInt

template <s, ze + N>

class BigInt {

using DigitType = int;

static const DigitType kBase = 10 000;

DigitType digit[N];

bool isNegative;

10<sup>4(n-1)</sup>

One  $\leftarrow$  скобка

число 1234567  $\xrightarrow{+ *}$  представление [7|6|5|4|3|2|1]

(+) for (size\_t i=0; i< N; ++i){

~~res[i] =~~  = a[i] + b[i] + res; ~~decrese~~  $\rightarrow$  operator

res[i] = sum % kBase;

res; ~~decrese~~ = sum/kBase;

5

1 2 3 4 5 6 2  $\xrightarrow{1}$

[1 2 3 4 5 6 7] 7 6 5 4 3 2 1 0 0 0

while (x < divisor) {

x \*= kBase

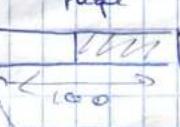
x += a[i]

Peque

Page

[1000]

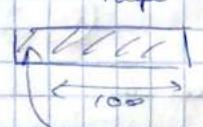
100



Page

[1000]

100



Page

[1000]

100



Page

[1000]

100



d[i]

if (i < front.size())

newear front[100 - i - 1]

else

page = ds(i - front.size()) / 100;

pages[page\_idx][(i - front.size()) % 100]

создание новых  $\rightarrow$   
заполнение новых pages

1) вр

2) текущий вр page

cycle - diff[i]

i - real = (k + i) % size

13)

Как определить это значение?

```
A (const AD) {
    ...
}
```

// no умножение

```
A (const AD*)
```

10.03.20

(EMUHAP)

```
void Func (1) T value
          (2) & value
          (3) const & value
```

```
Func ("abca") 1 OK
      2 CE
      3 OK!
```

Event e;

Date  
↑  
Event

Date d=e; // ошибка

Date &amp; d\_ref=e;

d\_ref.GetDescription(); CE

Matrix (size\_t + n, size\_t + m)

$$a_{ij} \quad i \in [0, \dots, n-1] \\ j \in [0, \dots, m-1]$$

ЗАГАДКА ПРО МАТРИЦУ

13)

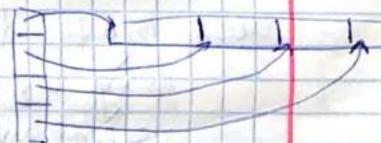
I]  $T^{**} \text{matrix} = \text{new } T[n]$

for (...)

matrix[i] = new T[m];



II]  $T^{**} \text{matrix} = \text{new int}[n]$

 $T^* \text{buffer} = \text{new int}[n * m]$ 

Matrix &lt;int&gt; m(10, 11);

m[5][5] = 5;

operator[];

Генератор вызывает buffer

OK

m(5, 5) = 10;

 $T^* \text{operator}[](\text{size\_t } i)$   
 $\text{return matrix}[i];$ 
~~operator[](T\*)~~
 $T^* \text{operator}() (\text{size\_t } i, \text{size\_t } j)$   
 $+ (i > u - 1 \text{ || } j > v - 1) \{$ 

cross();

 $\text{return matrix}[i][j];$ 
 $\text{const } T^* \text{operator}() (i, j) \text{ const};$

4)

### ✓ КОНСТРУКТОРЫ...

```
class A; public A(int) { }
```

```
class B: public A {
```

```
    int x;
```

```
public:
```

```
B() { }
```

```
x=0
```

```
}
```

Even if A has constructor

default constructor.

no parameter  $\Rightarrow$  CE

```
B(): A() { }
```

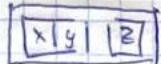
default constructor

```
...
```

```
B(int a, int x): A(a), X(x)
```

### ✓ ДЕСТРУКТОРЫ...

```
B
```



Memory usage is &, then

y < x.

```
~B()
```

delete p;

```
y
```

last bytes freed by A

### ✓ СРЕДСТВА (Slicing)

```
struct A {
```

```
int x;
```

```
void f();
```

```
Y
```

```
struct B: A {
```

```
int y;
```

```
void g();
```

```
Y
```

A a; A aa = b; //ok, сконструированно то же

B b; B bb = aa//CE //ok & B or A

struct B: private A { A aa = b//CE  
protected

B bb = a//CE

### Изменение указателя

aa.aa = b;

A\* aptr = &B;

```
struct A {
```

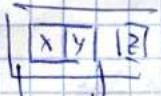
```
int x;
```

```
void f();
```

```
Y
```

```
struct B: public A {
```

aa.x = 0 //изменение & B



b

aa.f()

aa.y = 0//CE

aa.g()//CE

5)

39) struct A {

private:

int x;

protected:

int y;

public:

int z;

}

B::B;

B.x // CE ok, ok

B.y // CE ok, ok

B.z // CE, ok, ok

struct B : A {

1) private

2) protected

3) private

void g() {

x=0 // CE ok, ok

y=0 // CE ok, ok

z=0 // CE ok, ok

}

struct C : B { }

}

✓ ЗАПЕЧЕННІ ЗАМЕСТУВАННІ ЧЕГОДІГ (shadowing...)

struct B : A {

void f(int);

A::f(); // ok

B::B;

B.f(); // ok

B.A::f(); // ok

B.f() // CE - використання членів з іншої класу

B.A::f() // ok

B.g() // ok

struct A {

void f(int);

void f();

void g();

}

struct A {

void f();

void f(int);

void f(double);

void f(double) = delete;

B.f(); // Ok

B.f() // ok, т.к. "using A::f"

B.A::f(); // ok

B.f(); // ok

B.f(5.5) // ok, overload A::f(int)

B.f(5.5) // even though using..., no able → int in f(int)  
in B

struct B : A {

using A::f

void f();

void f(int);

void f(double) = delete;

B.f(); // Ok

B.f() // ok

B.A::f(); // ok

B.f(); // ok

B.f(5.5) // ok, overload A::f(int)

B.f(5.5) // even though using..., no able → int in f(int)  
in B

✓ СОЗНАЧЕННІ В УКАЗОВАННІХ ОДНОРОДНИХ ПРОВІДОВИХ КЛАССІВ

struct A {

int x;

int y;

}

A a;

[x][y]

B b;

[x][y][z]

, не очікує

координат початку, а кінця  
написки

struct B : A {

int z;

}

37) struct A {

int x;

private!  
int z;

y;

struct B: A {

int y;

void f() {

z = 0

y

B: B,

B.x = 0

B.y = 0

B.z = 0 // CE

некорректное обращение  
к недоступной переменной.

функция, before фиксирована

38) ✓ Модификатор доступа protected

struct A {

protected:

int t;

y;

struct B: A {

void f() {

t = 0 } ] ok

y

Классы, которые могут использовать  
protected, то

B.t ... //CE

✓ Модификатор доступа public

struct B: A;  $\Leftrightarrow$  struct B: public A;

Бесконечный цикл зацикливание

struct B: private A;

также не имеет  
право зацикливания

исследование от A, кроме "зарезано"

A: int x;  
int f();

B: int y;  
void g() {  
f();  
y = 3 } ] ok

B. f() //CE

B. g() //OK

Бесконечный цикл зацикливания в  
private члене B.

struct B: protected A; тут тоже неизвестно

struct C: ? B

35

26.03.20

return  
us  
для всех

## Вариантные функции

```
printf("%d.%d.%lf", x, y, z);
```

```
int Min (int x, int y); { } - MAX 0!
int Min (int x, int y, int z) { }
int Min (int, int, int, int)
```

template <class ... Args> // производное  
int Min (Args... args) { args... организовано  
// const Args\* - все параметры наследуются  
return sizeof...(args); } // возвращает количество параметров

template <class T, class ... Args> int Min (int\*x) {  
int Min (T value, Args... args) { }  
int min\_value = Min (args...);  
return (value < min\_value) ? value : min\_value;

template <class ... Args>  
int MinSquared (Args... args) {  
return Min ((args \* args)...);  
} Func(args...) Func(args)...

5.03.20

36

## Компьютерные I

- Наследование - механизм языка, позволяющий создавать классы на основе уже существующих, с наследием /наследованием/ заложенных ограничений

```
class Stack; // хранит один итератор на текущий элемент
```

- 1) Переименовать stack на stackMin
- 2) Реализовать новый метод StackMin
- 3) Использовать наследование

```
struct A {  
    int x;  
    void f();  
};
```

(A) Родитель, B наследует от A  
A - базовый класс  
(наглядно...)

```
struct B : A {  
    int y;  
};
```

B - производный класс  
(наглядно, потомок...)

A a;  
a.x=5;  
B b;  
b.y=4;  
C c;  
c.x=6;  
a.f(); b.f();

template <class ... Args>  
class A : public Args... {  
public:  
 void f();  
};

→

B b;

B, C - это организовано  
A < B < C > D > 0;

```

template <class T>
void f(const stack<T> &)

```

↑  
нпопу3 Бомббикс ареа

```

int a[100];
type(a) = int[100]
int[100] ↔ int * const

```

```

void f(int (&a)[100]) {
}

```

```
std::array<int, 30> arr;
```

```

new String[10];
new String;

```

→ Выгружение памяти

→ Быстро копируется  
(копируется)

3%

```

String * p = malloc(sizeof(String))
*p = S //RE

```

new(p) String(10);

Быстро копируется  
или последнее значение

void \*

```

operator new(size_t, size);
void *
operator new[](size_t, size);

```

Конечно же неизвестно

Аналогично предыдущему delete

delete [] arr;

for (int i = 0; i < 100; ++i) {
 arr[i] → ~String();
}

operator delete(arr);

8)

### Контроле синтаксиса

✓ since синтаксис - обработка синтаксиса;

auto x = 5; // type(x) = int;

auto y = f(); // type(y) = int

int f();

auto &z = x // ссылка

const auto &t = z;

(Проблема синтаксиса в том что в нем есть ненужные  
за исключением std::min и std::max. ~~лишние~~)

### register

Температура является стеком синтаксиса

(абстрактный класс синтаксиса)

### Параметры по умолчанию

template <class T> void f(T);

class Queue {

..

Queue<int> qr;

}

std::vector<int>

### CEMULHAP

30.03.20

32

mutable Stack<int>; // может изменяться  
non mutable Stack<int>; // неизменяется

fixable queue

stack.h

struct Node;

Stack {

Node\*

y

stack.cpp

struct Node {

T value;

y

..

T TOP() const {

Приложение которое проверяет корректность  
создания -> макросы этого могут быть в L.h,  
или же CE

Template <class T>

class Stack {

..

Stack<int> S;

void f(Stack<int> S) // CE

29

## Conventional Methods

Designers usually create their cool  
desired functionality or characteristics now

Template < >

```
class Stack < Cool > {
    :
}
```

## Functional approach:

Designers make more generic no dependency  
functionality & guarantees

Template < class T >

```
class Stack < T * > {
    :
}
```

Stack < int \* > z; // T is template argument (T)  
exp. like T \*

void f(T\*)

if (f(12)) {  
if (T - pointer) {

T is pointer < T >; value

else {

:

Template < class T >

class Is pointer {

public:

specific const cool;

value false;

y

Template < class T >

class Is pointer < T \* > {

value true;

q

## nontype template parameters

int arr [10];

↑  
compile time  
cost

~~parameterized~~

Average int, 10 > arr;

Template < class T, size + N >

class Array {

T[N] left;

T last - [N];

compile time  
cost  
only function

27

```
template <class T>
void f(T x, T y);
{
    int x;
    float y;
    f(y, x); // CE
    f<float>(x, y); — сюда не попадет
}
```

Частичное предложение - это фрагмент кода

Частичное выражение - не выражение.

При определении частичного выражения оно не используется

$f(T x)$   
const int y;

Многорядное предложение  
 $c++$   
(Partial)

28

Частичное выражение

```
template <class T> // <class T, int ...>
class Stack {
    T * = 0; // ...
    size_t size = 0;
    size_t capacity;
}
```

int main()
{
 <sup>int</sup> Stack stack;

void f(int, int)

template <class T>
void f(T, T)

template <class T>
void f(T, T)

$f(x, c) = f(10, 10)$

$f(1.0, 0) = f(T, T)$

Если выражение не является выражением, то оно не поддается частичному

Частичное выражение можно так, чтобы это поддавалось

### • reinterpret\_cast<Type\*>(expr)

expr - указатель/ссылка/член-члн.

expr Expr - ссылка, но type - объект

указ. указатель/ссылка

Пример: reinterpret\_cast<size\_t>(8x)

float x;

reinterpret\_cast<int>(8x)

reinterpret\_cast<size\_t>(x) = 8

### • dynamic\_cast (автомат.)

## 27.02.20 Модули

• Полиморфизм - C++, позволяющее одному и тому же коду работать в различных типах (один интерфейс, много реализаций)

• Системический полиморфизм - шаблонный, в котором реализации разбросаны по строкам классификации

void F(int x);

void F(double x);

F(a); // x+y

void swap(int& x, int& y){

int temp = x;

x = y;

y = temp;

y

void swap(double &x, double &y){

y

void swap(Stack &x, Stack &y);

### Модуль функций

template <class T>  
void swap(T &x, T &y)

T temp = x;

x = y;

y = temp;

int x,y // swap(x,y)

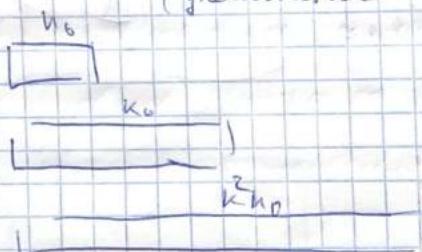
int a,b // swap(a,b)

swap(Stack &x, Stack &y) = delete

Система  
им-готика

Недоступно  
использование.

2) Многоразмножебое (хана  
(увеличение в  $K$  раз)

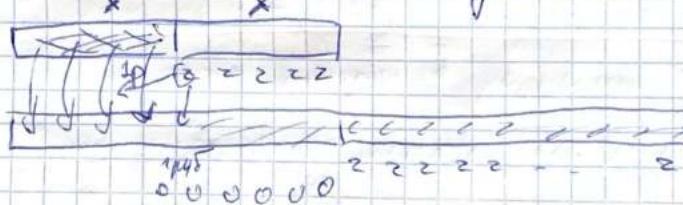


Формул:

$$h_0 + \overbrace{(h_0 + (K-1)h_0)}^{K^1} + K^2 h_0 + \dots + h_0 K^{\log_{K} \frac{h}{h_0}} = \\ = \sum_{i=0}^{\log_K \frac{h}{h_0}} h_0 K^i = h_0 \frac{K^{\log_K \frac{h}{h_0}} - 1}{K - 1} \approx h_0 \frac{h/h_0}{K-1} = O(h)$$

Банковский метод ( $K=2$ )

Тип iterable имеет для  $\text{for}$  3  $\text{P}$  на шаги -  $\text{for}(x)$



Б) программа (автоматизированная) строка  
1 из 21-го  $O(1)$

Г) выделение места в стеке C++.

в) в стеке C: (type) expr      (int)x  
                  type(expr)      int(x)

точно противоположно к совместимости

в) в стеке C++:

- static\_cast<type>(expr)

Возвращает то же значение что и выражение  
void  $\leftarrow$  type

Если это то, что требуется, то CE

Пример void f(int)

void f(double)  
int x = 5;

f(\*)      f(static\_cast<double>(x))

- const\_cast<type>(expr)

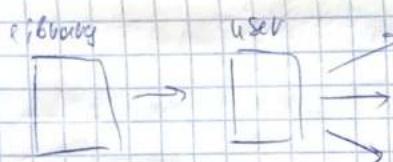
ни expr - указатель / ссылка

type - тот же тип, что и у expr с той же областью

то const и volatile

нельзя  
int \*  $\rightarrow$  const int \*  
const int \*  $\rightarrow$  int \*  
operator

2)



```

A a;
A* a_ptr;
A& a_ref;

```

ReturnType operator+(...){

}

- 1) Echte Wiederaufrufe (wie hierog)
- 2) Keine Wiederaufrufe

$x+y$ ; //  $x$ .operator+( $y$ );

$y+s$ ; //  $y$ .operator+( $s$ );

$s+y$ ; //  $s$ .operator+( $y$ ); // CE

const char\* s1 = "aBa";

s2 = "aBa";

if (s1 == s2) {

std::cout << "OK";

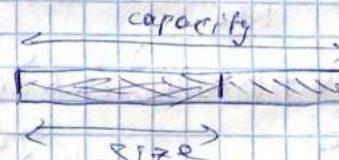
}

12.2 TABL esp. alle neueren

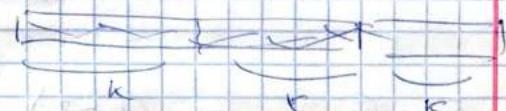
$\boxed{a \ b \ a}$

<cstring>

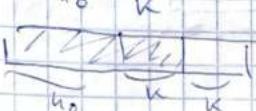
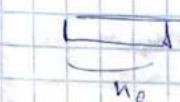
std::string



→ Appenzeller's exercise (gleiches wie für K-Dimension)



Atomics:



$$n_0 + (n_0 + k) + (n_0 + k + k) + \dots =$$

$$= \sum_{i=0}^{n-k} (n_0 + i \cdot k) =$$

$$= \frac{n-n_0}{k} n_0 + k \sum_{i=0}^{\frac{n-n_0}{k}} i = \frac{n-n_0}{k} n_0 + O(n^2)$$

O(n^2) genf

## 19) НЕПРЯМЫЕ ON-B ИСПОЛЬЗУЕМЫЕ МЕТОДЫ

void f(double x);

f(x), где x - Complex

Complex:

(explicit) operator double ()(const){}

refers to -

}

void f(Complex x);

также можно написать f((double)x), где x - Complex

explicit operator bool(){}

return re\_ != 0;

}

указывается

if (x) -

for(..; x; --)

x? j: 2

operator <sup>(int)</sup> (auto) const {

return (int) re\_;

}

## СЕМИНАР

25.02.20

void SetDay (int day) {

if (IsCorrect (day, month\_, year\_)) {

day\_ = day

else {

... = ...

}

void SetMonth (int month) {

if (IsCorrect ..)

}

void SetYear ( ) {

IsCorrect

}

• NULL // C #define NULL 0

• nullptr // C++ & gp = std::nullptr\_t

void\*  
char\*  
int\*

**++ , --**

```
class Complex {
```

public:

```
Complex & operator++() { // ++x  
    ++re_;  
    return *this;
```

}

*out*

```
Complex operator++(int) { // x++  
    Complex old = *this;  
    re_++;  
    return old;
```

}

**>>, <<**

```
Complex x,  
std::cin >> x; // stream  
std::cout << x; // stream
```

*Stream buffer..*

```
friend std::istream & operator>> (std::istream & i, Complex & x) {  
    i >> x.re_ >> x.im_;  
    return i;
```

}

**friend void f(Complex x)**

**x.re\_ ... //OK, OK, friend**

}

**friend class A**

:

}

**friend std::istream & operator>>(..)**

}

**friend int A::f(Complex x);**

**friend** - unclear concept Complex

**void operator()(...);** // rough how work unclear

**[ ]();** - poor usage of operators

## ПРИВИДА ПЕРЕГРУЗКИ

1) Класс переопределяет все базовые методы  
( $\Rightarrow$  виртуальные методы этого класса. т.к.)

2) Члены определяют свои операторы

~~operator~~, ~~operator~~

3) Члены переопределяют функции и приватные

4) При переопределении операторов &&, ||, >

переопределяются все эти

$\hookrightarrow$  (сторона симметрично)

5) Члены переопределяют ::, ?:, ., .\*, sizeof

6) Операторы (), [], ->, = можно переопределить  
максимально широким диапазоном

Class Complex {

public:

Complex operator+(Complex other);

Complex reset(Re\_+other.Re\_,

Im\_+other.Im\_)

return result;

release3. delete work memory

Complex(x)

Complex(y)

x+y

if

x.operator+(y)

Complex& operator+=(Complex other);

Re\_+=other.Re\_;

Im\_+=other.Im\_;

return \*this;

[ this (A const) ]

Complex x;

5+x; // CE

конструктор

Complex (double x);

}

переопределение конструктора

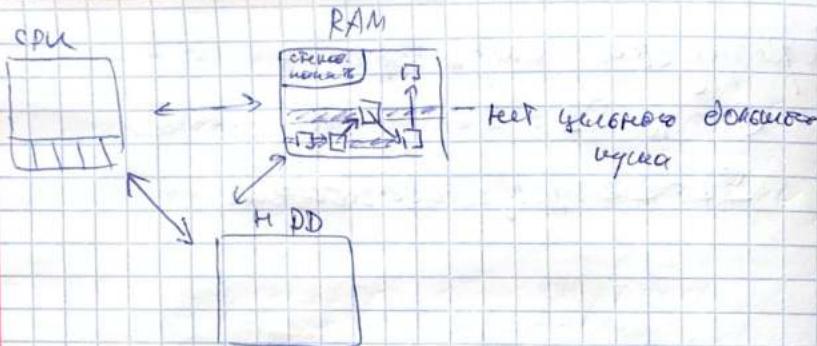
Complex operator+(Complex x, Complex y){

}

Complex operator+=(Complex x, Complex y){

y = return x;

13)



CPU заливает в cache ОЗУ. Использует RAM.  
 ⇒ Even stack array можно

Stack x=y; // к-р коррект.

x=y; // неправильное

Stack & operator = (const Stack & other) {

this - y - no new code

Проблема конструктора.

Copy And Swap idiom

14)

PIMPL = pointer to implementation

Лекция

20.02.2020

RAT I - Resource Acquisition

IS Initialization

(вызывающие функции-это инициализаторы)

③ ПЕРЕГРУЗКА ОПЕРАТОРОВ

class Complex {

double re\_;

double img\_;

public:

double Re();

double Img();

void SetRe(double Re);

void SetImg(double Img);

Complex Add(Complex x, complex);

Add (a, b) // a+b

Add (Add(a, b), c) // a+b+c

for operator

11

Сущесвует инициализатор - это конструктор с именем инициализации которого. Имя и ново-созданы

### Деструктор

Деструктор - это конструктор метод, который вызывается при уничтожении объекта  
(выход из области действия или delete)

`Date *ptr = new Date`

`delete ptr;` // вызов д-р + очищение памяти

```
~Stack()
{
    delete [] buffer;
}
```

- пример деструктора

Если же создавать объект, г-р, то...

= default

= delete // нечестно создавать объекты на стеке

Перегрузка оператора =

`Stack S1 = S2;` // конструирование

`S1 = S2;` // присваивание

### Радора с преобразованием

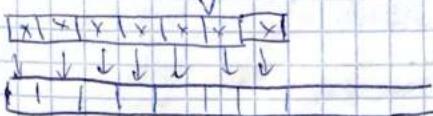
Правило Трех: Если в функции передаются объекты  
переделанные в-р конст. или ф-р или  
присваивание, то преобразование может бы.

### СЕМИНАР

18.02.2020

Explicit - запрещает неявные преобразования  
у конструктора с общим з-ром - обязательно,

### ~~Stack~~ Array



### ~~Stack~~ List



1. Функциональные наимен.

2. Практика по конст.

3) К-р копирования - К-р, который копирует  
один объект, создав при этом новый.

Stack S1;

:

Stack S2(S1); или Stack S2=S1;

Если не обозначить К-р концепт,  
то компилятор создаст свой,  
который

у которого скопиет все

хорошо известные концепт. от  
меню

или отдельно

```
class Stack {
public:
    const Stack(&other);
    ... = default; // концепт конструктора
};
```

компилятор сам  
создает

то бд. запускает концепт, который копирует или

... = delete;

Stack (const Stack& other) h

buffer = new int[other.work\_size];

work\_size = other.work\_size

size\_ = other.size

for (size\_t i = 0; i < size; ++i) {

buffer[i] = other.buffer[i]

}

и т.д.

Делегирующие конструкторы  
и классификация

class A {

main()

const int a;

public:

A(int x);

a = x;

y

Aobj(5); // CE. ошибки при динамике  
const => error

A(y);

A(y);

→ //OK

итогом изъятие,  
а все вышеизложенное => тип очевиден

Date(): date\_(5); ; Date(): Date(5, 6, 2010);  
month\_(6);  
year\_(2010); ⇔ ;

## Численические поля конструкторов

1) Сингларитичн к-р (к-р по умолчанию -

- конструктор без аргументов

date () { ... }

int main ()

Date date;

Если это не является ни определением конструктора, то компилятор создаст это за вас.

То делает конструктор:

49) Никак

50) Конструктор сущ. конструкторы и конст.  
конст

51) Для двух типов - никак

такими можно либо - конструктор

class Date {

;

public: Date (int day);

Date () = default; // создает конструктор конструктором

date () {};

2) Конструкторы - к-р с явно определенными аргументами

Типичное использование конструкторов

int f (Date date);

int main ()

f(1543); // компиляция  
ошиб

Date date = t; // ошибка

class Date {

;

public:

(explicit) Date (int day);

↑  
y

Недостаток предупреждение  
int t Date

class Complex {

double re;

double im;

public:

Complex (double ...);

f(); // ОК

Date date = 1; // ОК

Date date(1); // ОК

f(Date(1)); // ОК

3)

Без перегрузки 2 константы метода (const и  
не const), но при const есть перегрузка  
const констант.

Конст константы обеих методов констант  
перегрузку

S. TOP() //no const  
(const stack&) S. TOP() //coast Top

### Статические поля в методах

Применимы все константных операции, а  
массивы - в целом.

class Stack {

static size\_t size = 0;

Node\* head;

stack::size = 3

stack::head = L //CE

Внешние static методы не могут менять static константы

ODR (one definition rule)

static const Empty::const

,

### I Blagoveshchensk 8 grade

Конструкторы и деструкторы

13.02.20

int main () {

const Date date;

date. SetDay(5);

date. SetMonth(10)

}

- Конструктор - создает новый объект  
базируется при создании этого объекта.

class Date {

int day\_;

int month\_;

int year\_;

public:

Date (int day, int year, int month) {

day\_ = day

month\_ = month

year\_ = year; }

int main () {

Date date(5, 10, 2020);

### 3) struct Stack {

private:

Node\* head = nullptr;

size\_t size = 0;

public:

void push(int val){

};  
};

void Copy(const Stack\* s){

};  
};

};

Stack others

others. copy(s)

### • Классы

class  $\leftrightarrow$  struct (имя: 6 операций на генераторе  
public, 6 приват - private

### • Конструкторы (конструирование инициализации)

Stack s;

s.push(0);

s.Empty();

const Stack other = s;

other. Empty() // CF из-за неправильного макр-то  
известно

С

• Конструируемые методы - методы, которые  
могут генерировать константные объекты и которые  
возвращают неизменяемые объекты.

bool Empty() const{} // const метод

;

};

int & Top(){

return head->value;

};

s. Top() = 5;

1)

## Ибрағимов Булат

### ① Введение в ООП

- Приложения:

  - 1) Императивное
  - 2) Процедурное
  - 3) Модульное
  - 4) ООП

## ООП

- ООП - наработка языка, которая основана на описание пред. обл-ти в виде классов, объектов и их вз-й

### Сп-ва:

- 1) АБСТРАКСИЯ
- 2) ИНКАМУЛЯЦИЯ - механизм изоляции, изолят. обл-ти от внешн. и функций, работают с ними (+ обеспечено скр. данных)
- 3) Полиморфизм
- 4) Концептуализм

2)

## struct Stack {

node → [ Node \* head; ]  
size → size\_t size;

### метод → void Push(int val){}

Node \* new\_head = new Node(head, val);  
head = new\_head;

}

### метод Empty():

### int main(){

Stack s;

s.push(0)

bool Stack::Empty() {

return size == 0;

}

typedef unsigned long long, size\_t;

int 8-t

int 16-t

int 64-t

### • Модификаторы доступа (public, private)

private - вид виртуал., защищает методы от изменения

public - вид виртуал., доступные всем