

Объектно-ориентированное программирование

Object-oriented programming

XIX. Дизайн относительно данных

Data-oriented design

<https://youtu.be/rX0ltVEVjHc&t=4580s>

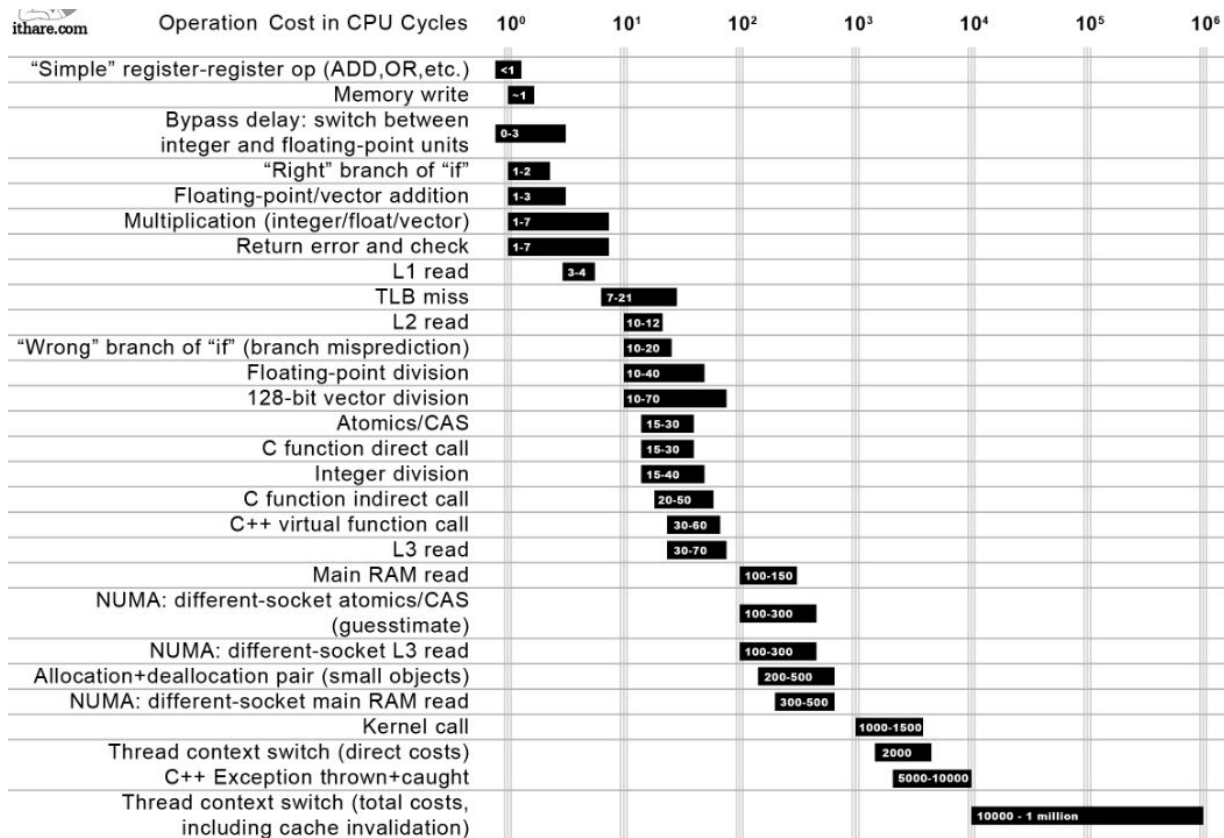


"You don't care how long it takes. Great. But **people who don't care** how long it takes is also **the reason why I have to wait 30 seconds for Word to boot.**"

"Software is getting slower more rapidly than hardware becomes faster."

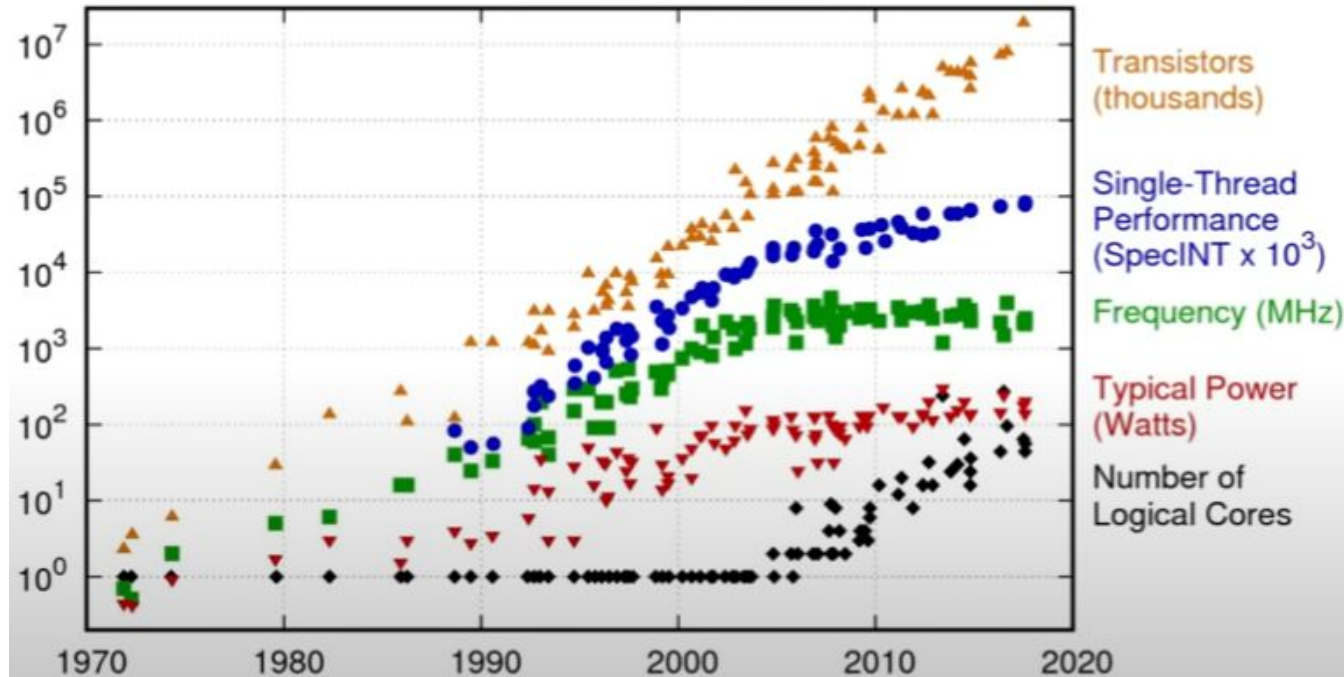
N. Wirth

Мотивация



<http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/>

Тренды в архитектуре процессоров



<https://youtu.be/ICKIMHCw--Y>

Промежуточные выводы

1. Использованные данные кэшируются, кэш бывает нескольких уровней
2. Кэш сохраняет данные в “линию” (cache line), используя **хэш-таблицу с открытой адресацией**, количество “бакетов” сильно ограничено
3. Механизм угадывания пытается заранее определить линию, которая понадобится процессору
 - a. кэшируется то, что понадобится в будущем (temporal locality)
 - b. соседние байты тоже кэшируются на всякий случай (spatial locality)
4. Кэш не всегда общий для разных ядер/потоков
5. Не все типы данных одинаково хорошо кэшируются

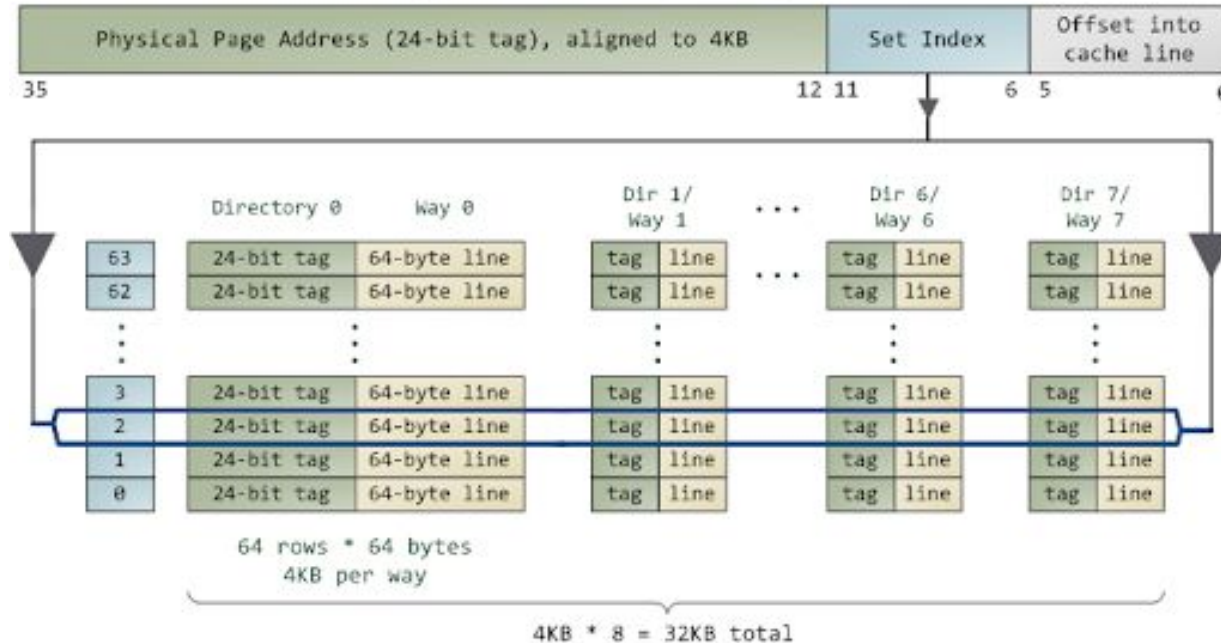
<https://youtu.be/Nz9SiF0QVKY>

Устройство кэша

L1 Cache – 32KB, 8-way set associative, 64-byte cache lines

1. Pick cache set (row) by index

36-bit memory location as interpreted by the L1 cache:



<https://manybutfinite.com/post/intel-cpu-caches/>

Ключевые моменты

1. Работа с памятью – это медленно
2. Работа с процессором – это быстро
3. Кэш процессора ограничен в объеме
4. Объекты в памяти нужно делать меньше, особенно в тех местах программы, где этих объектов много

Хитрости чтобы выжать все до капли из процессора

Взятие обратного квадратного корня

```
float Q_rsqrt( float number ) {  
    long i;  
    float x2, y;  
    const float threehalfs = 1.5F;  
    x2 = number * 0.5F;  
    y = number;  
    i = * ( long * ) &y;           // evil floating point bit level hacking  
    i = 0x5f3759df - ( i >> 1 ); // what the fuck?  
    y = * ( float * ) &i;  
    y = y * ( threehalfs - ( x2 * y * y ) );  
    return y;  
}
```

<https://www.beyond3d.com/content/articles/8/>

Минимальный элемент без ветвления (останется не обнуленным):

```
int min(int a, int b) {  
    return a * (a < b) + b * (b <= a);  
}
```

```
int min(int a, int b) {  
    // precondition: INT_MIN <= (a - b) <= INT_MAX  
    return b + ((a - b) & ((a - b) >> (sizeof(int) * CHAR_BIT - 1)));  
}
```

<https://graphics.stanford.edu/~seander/bithacks.html>

Двоичный поиск без рекурсии:

```
size_t bsearch(size_t needle, const size_t haystack[16]) {  
    size_t i = (haystack[8] <= needle) ? 8 : 0;  
    i += (haystack[i + 4] <= needle) ? 4 : 0;  
    i += (haystack[i + 2] <= needle) ? 2 : 0;  
    i += (haystack[i + 1] <= needle) ? 1 : 0;  
    return i;  
}
```

Выбор без ветвления:

```
#define  BRANCHLESS_IF(f,x) ((x) & -((typeof(x))!!(f)))

#define  BRANCHLESS_IF_ELSE(f,x,y) (((x) & -((typeof(x))!!(f))) | \
                                     ((y) & -((typeof(y)) !(f))))
```

Почему так не надо делать

```
void filter_loop(const std::vector<Type>& data, ...) {  
    Type sum = 0;  
    // ...  
    {  
        for(auto x : data) {  
            if(x < 6) {  
                sum += x;  
            }  
        }  
    }  
}
```

Современный компилятор знает лучше вас

Type = float:

```
pxor    xmm1, xmm1
movss   xmm2, DWORD PTR .LC4[rip]
mov     rax, rcx
cmp     rcx, rdx
je      .L9
movss   xmm0, DWORD PTR [rax]
comiss  xmm2, xmm0      ; if(x < 6)
jbe     .L10
addss   xmm1, xmm0      ; sum += x
```

Type = int:

```
xor     ebx, ebx
mov     rax, r8
cmp     r8, rdi
je      .L9
mov     edx, DWORD PTR [rax]
cmp     edx, 6          ; sum += x
lea     ecx, [rbx+rdx]
cmovl   ebx, ecx
add     rax, 4
cmp     rdi, rax
jne     .L11
```

<https://godbolt.org/z/nCx7st>

Современный компилятор знает лучше вас

<https://youtu.be/1X88od0miHs>



https://www.youtube.com/playlist?list=PL2HVqYf7If8cY4wLk7JUQ2f0JXY_xMQm2

Объекты в памяти должны быть маленькими

```
int a;                                // 4, 4
bool b;                               // 1, 1
struct { int a; };
struct { bool b; };
union { int a; bool b; }
struct { int a; bool b; } // массив этих структур?
struct { int a; int *p; int b; }
struct { int a; int b; int *p; }
struct { int a; int b; int *p; bool b; }
```

Объекты в памяти должны быть маленькими

```
int a; // 4, 4
bool b; // 1, 1
struct { int a; };
struct { bool b; };
union { int a; bool b; } // 4, 4
struct { int a; bool b; } // 4, 8
struct { int a; int *p; int b; } // 8, 24
struct { int a; int b; int *p; } // 8, 16
struct { int a; int b; int *p; bool b; } // 8, 24
```

<https://youtu.be/IroPQ150F6c>

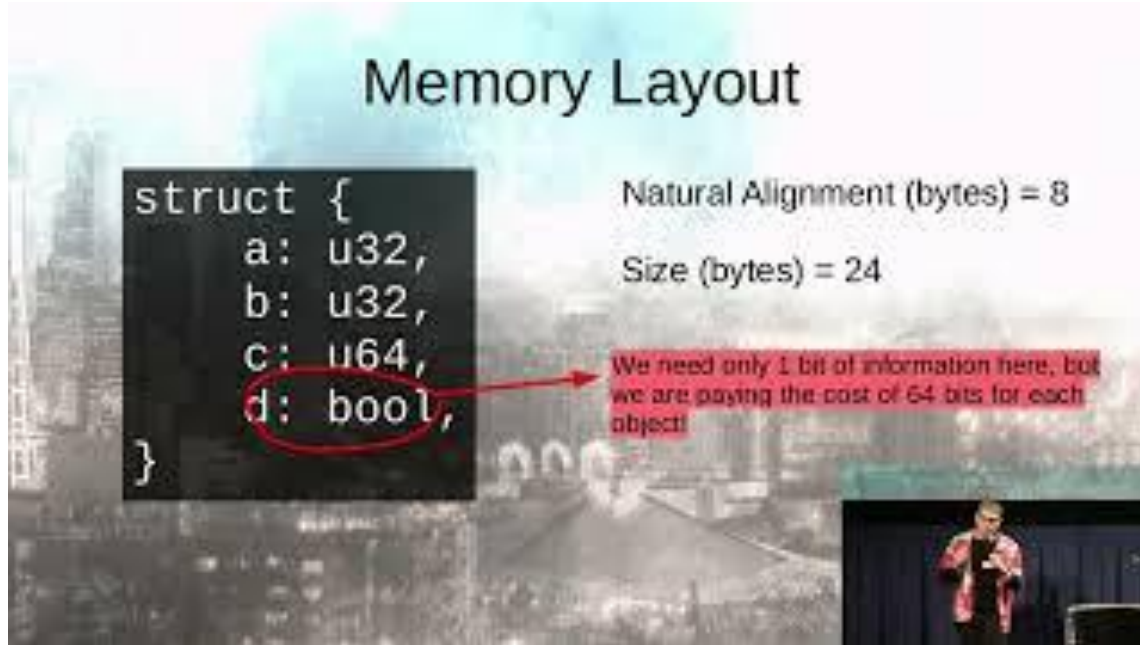
Memory Layout

```
struct {  
    a: u32,  
    b: u32,  
    c: u64,  
    d: bool,  
}
```

Natural Alignment (bytes) = 8

Size (bytes) = 24

We need only 1 bit of information here, but we are paying the cost of 64 bits for each object!



<https://vimeo.com/649009599>



```
const std = @import("std");
const parseInt = std.fmt.parseInt;

test "parse integers" {
    const input = "123 67 89,99";
    const ally = std.testing.allocator;

    var list = std.ArrayList(u32).init(ally);
    // Ensure the list is freed at scope exit.
    // Try commenting out this line!
    defer list.deinit();

    var it = std.mem.tokenizeAny(u8, input, " ,");
    while (it.next()) |num| {
        const n = try parseInt(u32, num, 10);
        try list.append(n);
    }

    const expected = [_]u32{ 123, 67, 89, 99 };

    for (expected, list.items) |exp, actual| {
        try std.testing.expectEqual(exp, actual);
    }
}
```

Объекты **is_invisible()** тоже обрабатываются

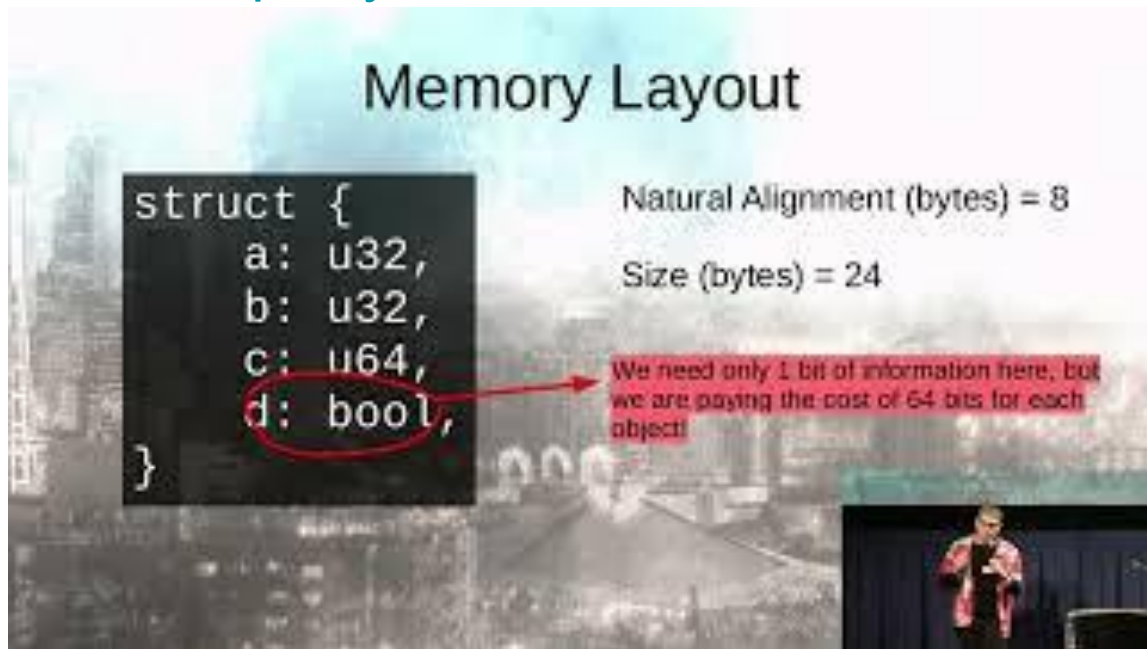
<https://youtu.be/IroPQ150F6c>

Memory Layout

```
struct {  
    a: u32,  
    b: u32,  
    c: u64,  
    d: bool,  
}
```

Natural Alignment (bytes) = 8
Size (bytes) = 24

We need only 1 bit of information here, but we are paying the cost of 64 bits for each object!



Состояние объекта вне данных (out of band)

Трюк 1. Вместо флагов в данных, группируйте сами данные в таблицах

```
// Булево поле вынуждает терять 3 байта на каждом объекте:  
struct s { int a; bool b; };  
s arraylist[100] = {}; // sizeof(arraylist) == 800  
  
// сама информация может храниться без булевого поля:  
struct s { int a; }  
s a_list[50] = {}; // sizeof(a_list) == 200  
s b_list[50] = {}; // sizeof(b_list) == 200
```

* Проверки флага тоже не нужны при такой схеме хранения

“From a **database** perspective, the **out-of-band** method is like **normalizing** your data with **foreign key** indexes; damn those RDS architects back in the 60s & 70s were on to something”

<https://youtu.be/IroPQ150F6c>

Относительная адресация против абсолютной

Трюк 2. Пользуйтесь целыми числами вместо указателей (x2 экономия памяти)

```
// вместо хранения конкретного адреса:  
struct s { String *a; String *b; } str;  
str.a = malloc(sizeof(String)); str.b = malloc(sizeof(String));  
  
// хранится расстояние от одного объекта до другого:  
struct s { unsigned a; unsigned b; } str;  
String n[2]; str.a = 0; str.b = 1;
```

<https://floooh.github.io/2018/06/17/handles-vs-pointers.html>

Отдельно про массивы

Трюк 3. Вместо массива разнотипных данных – структура всех данных, организованных по типу

```
enum v { fast, slow };
struct s { int *a; v b; }; // sizeof(s) == 16
s arraylist[100] = {};    // sizeof(arraylist) == 1600

// информация о классификации может быть частью структуры:
struct s { int *as[100]; v vs[100]; };
s multiarray = {};        // sizeof(s) == 800 + 400
```

* Шаблон “Structure of Arrays” (**Multi-array** в некоторых языках)

Ассоциативные массивы

Трюк 4. Если составное поле объекта часто пустоет (**sparse array**), его можно хранить отдельно в ассоциативном массиве

```
// вместо хранения пустых списков в объекте:  
struct s { int a; int b; int list[4]; }; // sizeof(s) == 24  
s arraylist[100] = {}; // sizeof(arraylist) == 2400  
// можно хранить в другой таблице по ключу:  
struct s { int a; int b; }; // sizeof(s) == 8  
s arraylist[100] = {}; // sizeof(arraylist) == 800  
map<unsigned, int[4]> m = {}; // sizeof(m) == [0, (20 * n)]
```

* Не считая служебной информации самого контейнера `map<>`

Чуть более сложный пример

```
const Monster = struct { // 32 bytes
    x: u32; y: u32;
    extra: union(enum) { // 24 bytes
        bee: Bee, human: Human,
    };
    const Bee = struct { // 1 byte
        color: Color,
        const Color = { yellow, black, red };
    };
    const Human = struct { // 20 bytes
        hat: u32, shoes: u32, shirt: u32, pants: u32,
        has_braces: bool,
    };
};
```

Чуть более сложный пример

```
const Monster = struct {           // 12 bytes
    x: u32; y: u32; tag: Tag,
    const Tag = enum { bee, human };
    const Bee = struct {           // 16 bytes
        base: Monster,
        color: Color,
        const Color = { yellow, black, red };
    };
    const Human = struct {         // 32 bytes
        base: Monster,
        hat: u32, shoes: u32, shirt: u32, pants: u32,
        has_braces: bool,
    };
};
```

Чуть более сложный пример

```
const Monster = struct { // 13-29 bytes
    tag: Tag; common: Common;
    const Tag = enum {
        bee_yellow, human_naked,
        bee_black, human_braces_naked,
        bee_red, human_clothed,
        human_braces_clothed,
    };
    const Common = struct { // 12 bytes
        x: u32, y: u32, index: u32,
    };
    const HumanClothed = struct { // 16 bytes
        hat: u32, shoes: u32, shirt: u32, pants: u32,
    };
};
```

Выводы

- Облегчайте процесс кэширования – храните данные в простых блоках байтов (contiguous memory)
- Группируйте данные в зависимости от порядка доступа к ним (packed cache space)
- Инструкции тоже кэшируются – редко используемый (cold) код должен вызываться отдельно от часто используемого (hot) кода
- Структуры массивов – ваши друзья

10 шагов к успеху

Шаг 1. Признать, что у вас есть проблема

```
class Base { int a; int b; };
class X : public Base {
    bool hidden; std::string id; bool special;
public:
    std::function<void(std::string)> on_hide; // strategy pattern
    void update() {
        if (hidden) { on_hide(id); }
        else if (get_config().SPEC || special) {...} else {...}
    }
};
using px = std::unique_ptr<X>;

std::vector<px> items; for (auto e : items) e.update();
```

10 шагов к успеху

Шаг 2. От объекта с поведением к **просто** объекту (plain ol' data)

```
struct Y : public Base {  
    bool hidden; std::string id; bool special;  
    std::function<void(std::string)> on_hide;  
};  
  
using py = std::unique_ptr<Y>;  
  
void update(const std::vector<py>& items) {  
    for (auto e : items) {  
        if (e.hidden) e.on_hide(e.id);  
        else if (get_config().SPEC || e.special) {...} else {...}  
    }  
}
```

<https://www.youtube.com/watch?v=WwkuAqObpIU>

Intro To Data Oriented Design



The image displays a comparison of two code snippets for a function named `ProcessFrenzy`, illustrating a performance improvement through Data Oriented Design (DoD).

Left Snippet (Original Code):

```
void ProcessFrenzy(List<Enemy> enemies) {  
    for (int i = 0; i < enemies.count; i++) {  
        var enemy = enemies[i];  
        if (enemy.isDead) {  
            enemies.RemoveAt(i--);  
        } else if (!GlobalConfig.globalFrenzy) {  
            // ...  
        } else {  
            // ...  
        }  
    }  
}
```

Right Snippet (Optimized Code):

```
void ProcessFrenzy(List<Enemy> enemies) {  
    var globalFrenzy = GlobalConfig.globalFrenzy;  
    for (int i = 0; i < enemies.count; i++) {  
        var enemy = enemies[i];  
        if (enemy.isDead) {  
            enemies.RemoveAt(i--);  
        } else if (globalFrenzy) {  
            // ...  
        } else {  
            // ...  
        }  
    }  
}
```

39x Speedup

10 шагов к успеху

Шаг 3. Сохранять объекты в кэше как можно дольше

```
struct Y : public Base {
    bool hidden; std::string id; bool special;
    std::function<void(std::string)> on_hide;
};

using py = std::unique_ptr<Y>;

void update(std::list<py>& items) {
    for (auto e : items) {
        if (e.hidden) items.erase(e); // observer pattern
        else if (get_config().SPEC || e.special) {...} else {...}
    }
}
```

10 шагов к успеху

Шаг 4. Мемоизация результатов (10% прирост производительности)

```
struct Y : public Base {  
    bool hidden; std::string id; bool special;  
};  
  
using py = std::unique_ptr<Y>;  
  
void update(std::list<py>& items) {  
    auto is_special = get_config().SPEC;  
    for (auto e : items) {  
        if (e.hidden) items.erase(e); // observer pattern  
        else if (is_special || e.special) {...} else {...}  
    }  
}
```

10 шагов к успеху

Шаг 5. Отказаться от объектов “на куче” (30%)

```
struct Y : public Base {  
    bool hidden; std::string id; bool special;  
};  
  
void update(std::list<Y>& items) {  
    auto is_special = get_config().SPEC;  
    for (auto e : items) {  
        if (e.hidden) items.erase(e); // observer pattern  
        else if (is_special || e.special) {...} else {...}  
    }  
}
```

10 шагов к успеху

Шаг 6. Иметь ввиду организацию объектов в памяти (10%)

```
struct Y : public Base {  
    bool hidden; bool special; std::string id;  
};  
  
void update(std::list<Y>& items) {  
    auto is_special = get_config().SPEC;  
    for (auto e : items) {  
        if (e.hidden) items.erase(e); // observer pattern  
        else if (is_special || e.special) {...} else {...}  
    }  
}
```

10 шагов к успеху

Шаг 7. Не использовать string без необходимости (20%)

```
struct Y : public Base {  
    bool hidden; bool special; unsigned id;  
};  
  
void update(std::list<Y>& items) {  
    auto is_special = get_config().SPEC;  
    for (auto e : items) {  
        if (e.hidden) items.erase(e); // observer pattern  
        else if (is_special || e.special) {...} else {...}  
    }  
}
```

Можно было обратить внимание

```
bool hidden; bool special;
```

```
!hidden && special // OK
```

```
!hidden && !special // OK
```

```
hidden && special // ???
```

```
hidden && !special // ???
```

10 шагов к успеху

Шаг 8. Избавиться от некорректных состояний и лишних удалений (66%)

```
enum states { shown, shown_special, hidden };  
struct Y : public Base {  
    unsigned id; states state;  
};  
  
void update(std::list<Y>& items) {  
    for (auto e : items) {  
        if (is_special || e.state == shown_special) {...} else {...}  
    }  
    items = items | std::ranges::filter(  
        [](auto e){ return e.state != hidden; })  
        | std::views::to<std::list>();  
}
```

10 шагов к успеху

Шаг 9. Использовать контейнер, более подходящий под задачу (75% прирост)

```
enum states { shown, shown_special, hidden };

struct Y : public Base {
    unsigned id; states state;
};

void update(std::priority_queue<Y>& items) {
    auto is_special = get_config().SPEC;
    for (auto e : items) {
        if (is_special || e.state == shown_special) {...} else {...}
    } for ( ; items.size(); items.pop())
        if (items.top().state == shown) break;
}
```

10 шагов к успеху

Шаг 10. Кодировать состояние объектов в коллекциях

```
class Base { int a; int b; };

struct Y : public Base { unsigned id; };

std::vector<Y> norm;
std::vector<Y> spec;
std::vector<Y> hidden;

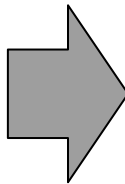
void update_norm  (std::vector<Y>&);
void update_spec  (std::vector<Y>&);
void update_hidden(std::vector<Y>&);
```

Data-oriented vs. Object-oriented

```
struct Shape {
    Shape(const Shape&) = delete;
    Shape& operator=(const Shape&) = delete;
    virtual ~Shape() = 0;
    virtual void draw(Window&) const = 0;
    virtual float area() const = 0;
private:
    Color c;
    bool is_visible;
};

struct Circle : Shape {
    void draw(Window &w) const override {
        if(is_visible()) { ... }
    }
};

struct Square : Shape { ... };
```



```
std::vector<std::unique_ptr<Shape>> data;

for(auto& ptr : data)
    ptr->draw(window);
```

```
struct Button {
    std::string text;
    std::unique_ptr<Shape> shape;
};
```

```
struct Button {
    std::string text;
    std::variant<Circle, Square> shape;
};
```

DOD vs. OOD

- Вызовы извне данных в кэше (pointer indirection)
- Объем кэша используется неэффективно (alignment, padding)
- Виртуальные вызовы внутри работающего цикла (“hot” code)
- Объекты **is_invisible()** тоже обрабатываются

DOD vs. OOD (продолжение)

```
struct Circle {
    Point center;
    float radius;
};

struct Square {
    Point top;
    float size;
};

struct Shape {
    int type;
    size_t index;
};

struct Shapes {
    std::vector<Circle> circles;
    std::vector<Squares> squares;
};
```

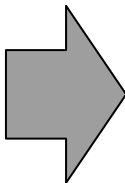
```
float area(const Shapes& geometry) {
    float res = 0;
    for(auto& c : geometry.circles) ...
    for(auto& s : geometry.squares) ...
    return res;
}
```

```
struct ShapesRender {
    std::vector<std::pair<Shape, Color>> visible;
};

void draw(Window& window, const ShapesRender& render,
          const Shapes& geometry)
{
    for(auto [id, color] : render.visible)
        draw(window, geometry.circles[id.index], color);
        draw(window, geometry.square[id.index], color);
}
```

Structure of arrays (SoA)

```
struct Circle {  
    Point center;  
    float radius;  
};  
  
struct Square {  
    Point top;  
    float size;  
};  
  
struct Shape {  
    int type;  
    size_t index;  
};  
  
struct Shapes {  
    std::vector<Circle> circles;  
    std::vector<Squares> squares;  
};
```



```
struct ShapesRender {  
    std::vector<std::pair<Shape, Color>> visible;  
};  
  
struct ShapesRender {  
    std::vector<int> type;  
    std::vector<size_t> index;  
    std::vector<Color> color;  
};
```

"Object-oriented programming aficionados think that everything is an object.... this [isn't] so. There are things that are objects. Things that have *state* and *change their state* are **objects**. And then there are things that are not objects. A binary search is not an object. It is an **algorithm**."

A. Stepanov

Everything is an object?

<https://www.youtube.com/watch?v=LC1yxlicWUs>

