
Яндекс

Яндекс

# C++ велосипедостроение для профессионалов

Antony Polukhin
Полухин Антон

Автор Boost библиотек TypeIndex, DLL, Stacktrace
Maintainer Boost Any, Conversion, LexicalCast, Variant
Представитель РГ21, national body

# О "велосипедах"

# Зачем?

# Зачем?

- Для самообучения

# Зачем?

- Для самообучения
- Особые требования к коду

# Зачем?

- Для самообучения
- Особые требования к коду
- Можем написать лучше

# Зачем?

- Для самообучения
- Особые требования к коду
- Можем написать лучше
- Кажется что можем написать лучше

# Зачем?

- Для самообучения
- Особые требования к коду
- Можем написать лучше
- Кажется что можем написать лучше
- ???

# О чём поговорим

- Устаревшие технологий
- Современные технологии
- Оптимизациях и “вредных” бенчмарках
- Новейших практиках C++ программирования
- Что делать с “идеальным” велосипедом

# С какого класса начнём?

# std::string

# string: с чего всё начиналось

```
class string {
    char* data;
    size_t size;
    size_t capacity;
    // ...
};
```

# string: C++98

```
class string {
    char* data;
    // ...
public:
    string(const string& s)
        : data(s.clone()) // new + memmove
    {}
    // ...
    string(const char* s); // new + memmove
};
```

# string: C++98

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)
);
```

# string: C++98 проблемы

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)   // string("Hello!")
);
```

# string: C++98 проблемы

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)   // make_pair(string("Hello!"), 1)
);
```

# string: C++98 проблемы

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)   // m.insert(make_pair(string("Hello!"), 1))
);
```

# string: C++98 проблемы

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1) // (new + memmove) * 2 + delete
);                                      // new + memmove + delete
```

# string: C++98 проблемы

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1) // копии котрые не надо копировать
);                                      // копии котрые не надо копировать
```

# string: COW

```
class string_impl {
    char* data;
    size_t size;
    size_t capacity;
    size_t use_count; // <===
    // ...
};
```

# string: COW

```
class string {
    string_impl* impl;
public:
    string(const string& s)
        : impl(s.impl)
    {
        ++ impl->use_count;
    }
};
```

# string: COW

```
class string {
    string_impl* impl;
public:
    char& operator[](size_t i) {
        if (impl->use_count > 1)
            *this =  clone();
        }
        return impl->data[i];
    }
};
```

# string: COW

```
class string {
    string_impl* impl;
public:
    ~string() {
        --impl->use_count;
        if (!impl->use_count) {
            delete impl;
        }
    }
};
```

# string: COW

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)
);
```

# string + COW: C++98

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)   // string("Hello!")
);
```

# string + COW: C++98

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)   // make_pair(string("Hello!"), 1)
);
```

# string + COW: C++98

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)   // m.insert(make_pair(string("Hello!"), 1))
);
```

# string + COW: C++98

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)
);                                          // 1 new + 1 memmove
```

COW более чем в 2 раза ускоряет код в C++98!

Ho...

Ho... 2002, 2005

# 2002, 2005

- Hyper-threading в процессорах Pentium 4.
- 2-ядерный процессор Opteron архитектуры AMD64, предназначенный для серверов.
- Pentium D x86-64 — первый 2-ядерный процессором для персональных компьютеров.

# string: COW MT fixes

```
class string_impl {
    char* data;
    size_t size;
    size_t capacity;
    size_t use_count; // <=== Ooops!
    // ...
};
```

# string: COW MT fixes

```
class string_impl {
    char* data;
    size_t size;
    size_t capacity;
    atomic<size_t> use_count; // <=== Fixed
    // ...
};
```

# string: COW MT fixes

```
class string {
    string_impl* impl;
public:
    string(const string& s)
        : impl(s.impl)
    {
        ++ impl->use_count; // atomic
    }
};
```

# string: COW MT fixes

```
class string {
    string_impl* impl;
public:
    ~string() {
        size_t val = --impl->use_count;  // atomic
        if (!val) {
            delete impl;
        }
    }
};
```

# string: COW MT fixes

```
class string {
    string_impl* impl;
public:
    char& operator[](size_t i) {
        if (impl->use_count > 1) {      // atomic
            string cloned = clone();    // new + memmove + atomic --
            swap(*this, cloned);
            // atomic in ~string for cloned; delete in ~string if other thread released
the string
        } // ...
```

# Чем плохи atomic?

Не атомарный INC — 1 такт [1]

[1] http://www.agner.org/optimize/instruction_tables.pdf

# Чем плохи atomic?

Не атомарный INC — 1 такт [1]

Атомарная операция на одном ядре [2]:

> ~5 - 20+ тактов, если это ядро "владеет" атомиком

[1] http://www.agner.org/optimize/instruction_tables.pdf

[2] https://htor.inf.ethz.ch/publications/img/atomic-bench.pdf

# Чем плохи atomic?

Не атомарный INC — 1 такт [1]

Атомарная операция на одном ядре [2]:

> ~5 - 20+ тактов, если это ядро "владеет" атомиком
>
> ~40 тактов, если другое ядро "владеет" атомиком

[1] http://www.agner.org/optimize/instruction_tables.pdf

[2] https://htor.inf.ethz.ch/publications/img/atomic-bench.pdf

# Чем плохи atomic? (часть 1)

Не атомарный INC — 1 такт [1]

Атомарная операция на одном ядре [2]:

- ~5 - 20+ тактов, если это ядро "владеет" атомиком
- ~40 тактов, если другое ядро "владеет" атомиком

Несколько атомарных операций на разных ядрах над одним atomic:

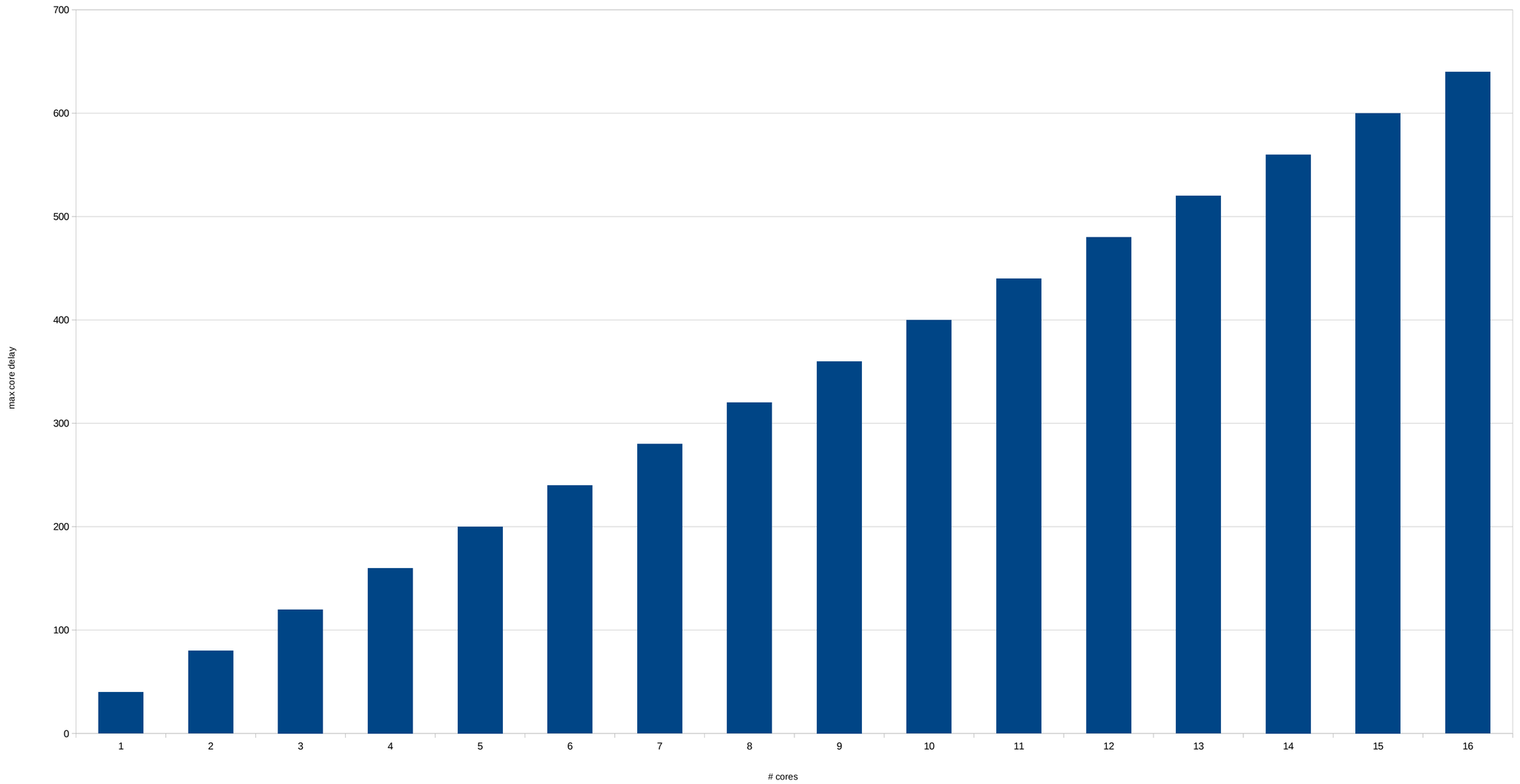
- retry later [3]

[1] http://www.agner.org/optimize/instruction_tables.pdf

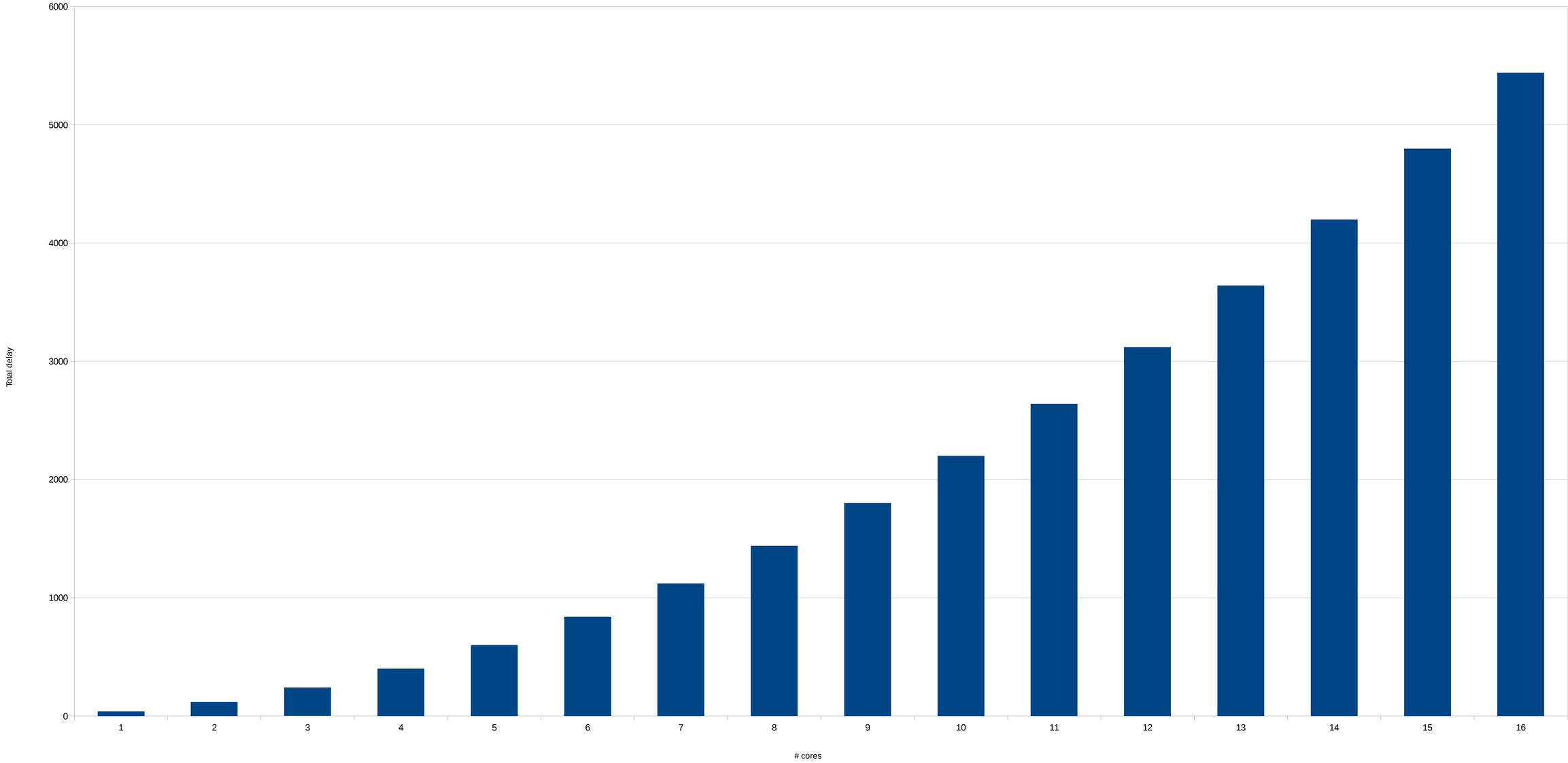
[2] https://htor.inf.ethz.ch/publications/img/atomic-bench.pdf

[3] https://en.wikipedia.org/wiki/MESI_protocol

# atomic (часть 1, макс. простой ядра)



# atomic (часть 1, суммарный простой ядер)



# Чем плохи atomic (часть 2)

- Компиляторы не очень умеют их оптимизировать.
- Full barrier для некоторых компиляторов.
- Накладывают дополнительные ограничения на близлежащие оптимизации.

https://gcc.gnu.org/wiki/Atomic/GCCMM/Optimizations

http://llvm.org/docs/Atomics.html

COW ~~более чем в 2 раза~~ ускоряет код в C++98!

# C++11

# string: C++11 and COW

- Временные объекты и без COW оптимизируются в C++11

```
string::string(string&& s) {
    swap(*this, s);
}
```

# string: C++11 and COW

Временные объекты и без COW оптимизируются в C++11

```
std::map<string, int> m;

m.insert(
    std::pair<string, int>("Hello!", 1) // string("Hello!")
);
```

# string: C++11 and COW

Временные объекты и без COW оптимизируются в C++11

```
std::map<string, int> m;

m.insert(
    std::pair<string, int>("Hello!", 1) // pair(string&&, int)
);
```

# string: C++11 and COW

Временные объекты и без COW оптимизируются в C++11

```
std::map<string, int> m;

m.insert(
    std::pair<string, int>("Hello!", 1) // m.insert(pair&&)
);
```

# string: C++11 and COW

Временные объекты и без COW оптимизируются в C++11

```
std::map<string, int> m;

m.insert(
    std::pair<string, int>("Hello!", 1)
);                                          // 1 new + 1 memmove
```

# string: C++11 and COW

- Временные объекты и без COW оптимизируются в C++11
- В C++11 если мы копируем объект, то скорее всего мы его будем менять (в остальных случаях rvalue + RVO + NRVO)

# string: C++11 and COW

- Временные объекты и без COW оптимизируются в C++11
- В C++11 если мы копируем объект, то скорее всего мы его будем менять (в остальных случаях rvalue + RVO + NRVO)
    - Без COW: new + memmove
    - COW: atomic + atomic(x?) + new + memmove

# string: C++11 and COW

- Временные объекты и без COW оптимизируются в C++11
- В C++11 если мы копируем объект, то скорее всего мы его будем менять (в остальных случаях rvalue + RVO + NRVO)
  - COW вызывает atomic удручающе часто на некоторых операциях

```
char& operator[](size_t i) {
    if (impl->use_count > 1) {        // atomic
        string cloned = clone();      // atomic--
        swap(*this, cloned);          // atomic in ~string for cloned
    }
    return impl->data[i];
}
```

# string: C++11 and COW

- Временные объекты и без COW оптимизируются в C++11
- В C++11 если мы копируем объект, то скорее всего мы его будем менять (в остальных случаях rvalue + RVO + NRVO)
  - COW вызывает atomic удручающе часто на некоторых операциях
  - COW конструктор копирования использует atomic
  - COW деструктор COW строки вызывает atomic

# string: C++11 and COW

- Временные объекты и без COW оптимизируются в C++11
- В C++11 если мы копируем объект, то скорее всего мы его будем менять (в остальных случаях rvalue + RVO + NRVO)
  - COW вызывает atomic удручающе часто на некоторых операциях
  - COW конструктор копирования использует atomic
  - COW деструктор COW строки вызывает atomic

Итого:

- С COW мы получаем множество дополнительных операций над атомиками

COW устарел!
Осторожнее с использованием.

# COW legacy

```
class bimap {
    std::map<string, int> str_to_int;
    std::map<int, string> int_to_str;

public:
    void insert(string s, int i) {
        str_to_int.insert(make_pair(s, i));
        int_to_str.insert(make_pair(i, std::move(s)));
    }
```

# COW legacy

```
class bimap {
    std::map<string, int> str_to_int;
    std::map<int, string> int_to_str;

public:
    void insert(string s, int i) {
        str_to_int.insert(make_pair(s, i));
        int_to_str.insert(make_pair(i, std::move(s)));
    }
```

# COW legacy

```
class bimap {
    std::map<string, int> str_to_int;
    std::map<int, string> int_to_str;

public:
    void insert(string s, int i) {
        str_to_int.insert(make_pair(s, i));
        int_to_str.insert(make_pair(i, std::move(s)));
    }
```

# COW legacy fix

```
class bimap {
    std::map<string, int> str_to_int;
    std::map<int, string_view> int_to_str;

public:
    void insert(string s, int i) {
        auto it = str_to_int.insert(make_pair(std::move(s), i));
        int_to_str.insert(make_pair(i, *it.first)));
    }
```

# Одна оптимизация для string

# Одна оптимизация для string

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)
);                                          // 1 new + 1 memmove
```

# Одна оптимизация для string

```
class string {
    char* data;
    size_t size;
    size_t capacity;
    // ...
};
```

# Одна оптимизация для string

```
class string {
    size_t  capacity;
    union {
        struct no_small_buffer_t {
            char*   data;
            size_t  size;
        } no_small_buffer;
        struct small_buffer_t {
            char data[sizeof(no_small_buffer_t)];
        } small_buffer;
    } impl;
```

# Одна оптимизация для string

```
class string {
    size_t  capacity;
    union {
        struct no_small_buffer_t {
            char*   data;
            size_t  size;
        } no_small_buffer;
        struct small_buffer_t {
            char data[sizeof(no_small_buffer_t)];
        } small_buffer;
    } impl;
```

# Одна оптимизация для string

```
class string {
    size_t  capacity;                                   // == 0
    union {
        struct no_small_buffer_t {
            char*   data;
            size_t  size;
        } no_small_buffer;
        struct small_buffer_t {
            char data[sizeof(no_small_buffer_t)];
        } small_buffer;
    } impl;
```

# Одна оптимизация для string

```
class string {
    size_t  capacity;                                   // == 0
    union {
        struct no_small_buffer_t {
            char*   data;
            size_t  size;
        } no_small_buffer;
        struct small_buffer_t {
            char data[sizeof(no_small_buffer_t)];       // <=======
        } small_buffer;
    } impl;
```

# Одна оптимизация для string

```
class string {
    size_t  capacity;                                   // != 0
    union {
        struct no_small_buffer_t {
            char*   data;
            size_t  size;
        } no_small_buffer;
        struct small_buffer_t {
            char data[sizeof(no_small_buffer_t)];
        } small_buffer;
    } impl;
```

# Одна оптимизация для string

```
class string {
    size_t  capacity;                                   // != 0
    union {
        struct no_small_buffer_t {
            char*   data;                               // <=======
            size_t  size;
        } no_small_buffer;
        struct small_buffer_t {
            char data[sizeof(no_small_buffer_t)];
        } small_buffer;
    } impl;
```

# Одна оптимизация для string

```
std::map<string, int> m;


m.insert(
    std::pair<string, int>("Hello!", 1)
);                                          // 1 memmove
```

# Разработка без оглядки на готовые решения

# std::variant

std::variant<T...> - класс который хранит один из типов T, и помнит тип данных:

```
union {
    T0 v0;
    T1 v1;
    T2 v2;
    // ...
};

int t_index;
```

# Кто видит ошибку?

```
template <class... T>
class variant {
    // ...
    std::tuple<T...> data;
};
```

# Кто видит ошибку?

```
variant<T...>& operator=(const U& value) {
    if (&value == this) {
        return *this;
    }

    destroy();                                  // data->~Ti()
    construct_value(value);                     // new (data) Tj(value);
    t_index = get_index_from_type<U>(); // t_index = j;

    return *this;
}
```

# Кто видит ошибку?

```
variant<T...>& operator=(const U& value) {
    if (&value == this) {
        return *this;
    }

    destroy();                          // data->~Ti()
    construct_value(value);             // throw; ...
    // ...
    // ~variant() {
    //     data->~Ti()                  // Oops!!!
```

# FORCE INLINE

# У процессора есть не только кеш данных

Большинство современных микропроцессоров для компьютеров и серверов имеют как минимум три независимых кэша: кэш инструкций для ускорения загрузки машинного кода, кэш данных для ускорения чтения и записи данных и буфер ассоциативной трансляции (TLB) для ускорения трансляции виртуальных (логических) адресов в физические, как для инструкций, так и для данных. [1]

[1] https://ru.wikipedia.org/wiki/Кэш_процессора

# Кеш инструкций здоровой программы

Sed ut perspiciatis, unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam eaque ipsa, quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt, explicabo. Nemo enim ipsam voluptatem, quia voluptas sit, aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos, qui ratione voluptatem sequi nesciunt, neque porro quisquam est, qui dolorem ipsum, quia dolor sit, amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt, ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur?

# Кеш инструкций с FORCE_INLINE

Sed ut perspiciatis, unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totFORCE_INLINE rem aperiFORCE_INLINE eaque ipsa, quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt, explicabo. Nemo enim ipsFORCE_INLINE voluptatem, quia voluptas sit, aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos, qui ratione voluptatem sequi nesciunt, neque porro quisquFORCE_INLINE est, qui dolorem ipsum, quia dolor sit, FORCE_INLINEet, consectetur, adipisci velit, sed quia non numquFORCE_INLINE eius modi tempora incidunt, ut labore et dolore magnFORCE_INLINE aliquFORCE_INLINE quaerat voluptatem. Ut enim ad minima veniFORCE_INLINE, quis nostrum exercitationem ullFORCE_INLINE corporis suscipit laboriosFORCE_INLINE, nisi ut aliquid ex ea commodi consequatur?

# Много бенчмарков игнорируют 2 кеша из 3х

Бенчмарк должен:

- в бинарном виде занимать столько же, сколько ваш production код
- иметь приблизительно такое же количество функций и переходов, как и ваш production код

# True story

Убрав часть шаблонных параметров из B-Tree и ощутимо уменьшив размер бинарника, получили двукратный прирост скорости.

# Мнение эксперта

Understanding Compiler Optimization - Chandler Carrut: ”To get an interesting example <on inlining that hurts performance> it has to be big <...>”[1]

"Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My! - Chandler Carrut: “Primary reason why modern processors are slow is due to cache misses. <...> One of those caches is actually a cache that feeds your program to the CPU. <...> When you skip over instructions, you may skip over the cache line <...> and you stall” [2]

[1] https://www.youtube.com/watch?v=FnGCDLhaxKU

[2] https://www.youtube.com/watch?v=nXaxk27zwlk

# Aliasing

# Aliasing

Можно встретить рекомендации:

> использовать -fno-strict-aliasing чтобы получать меньше проблем.

# Aliasing

```
bar qwe(foo& f, const bar& b) {



    f += b;


    return b * b;
}
```

# Aliasing

```
bar qwe(foo& f, const bar& b) {
    // load b
    // load f
    f += b;
    // load b
    return b * b;
}
```

# Aliasing

```
bar qwe(foo& f, const bar& b) {
    // load b
    // load f
    f += b;             // &f == &b ???
    // load b
    return b * b;
}
```

# Aliasing

5% - 30% Speedup

http://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1124&context=ecetr

https://habrahabr.ru/post/114117/

# Советы на каждый день

# vector

```
auto foo() {
    std::vector< std::shared_ptr<int> > res;



    for (unsigned i = 0; i < 1000; ++i)
        res.push_back(
            std::shared_ptr<int>(new int)
        );

    return res;
}
```

# vector

```
auto foo() {
    std::vector< std::shared_ptr<int> > res;
    res.reserve(1000);    // <======

    for (unsigned i = 0; i < 1000; ++i)
        res.push_back(
            std::shared_ptr<int>(new int)
        );

    return res;
}
```

# make_shared

```
auto foo() {
    std::vector< std::shared_ptr<int> > res;
    res.reserve(1000);

    for (unsigned i = 0; i < 1000; ++i)
        res.push_back(
            std::make_shared<int>()    // <======
        );

    return res;
}
```

# for (auto v: data)

```
auto foo() {
    std::vector<std::string> data = get_data();
    // ...

    for (auto v: data)
        res.insert(
            v
        );
    return res;
}
```

# for (auto v: data)

```
auto foo() {
    std::vector<std::string> data = get_data();
    // ...

    for (auto v: data)
        res.insert(
            std::move(v)      // <======
        );
    return res;
}
```

# for (auto&& v: data)

```
auto foo() {
    std::vector<std::string> data = get_data();
    // ...

    for (auto&& v: data)      // <======
        res.insert(
            std::move(v)
        );
    return res;
}
```

# vector

```
auto foo() {
    std::vector<std::string> data = get_data();
    // ...
    std::string res = data.back();
    data.pop_back();
    // ...
    return res;
}
```

# vector

```
auto foo() {
    std::vector<std::string> data = get_data();
    // ...
    std::string res = std::move(data.back());  // <======
    data.pop_back();
    // ...
    return res;
}
```

# Global constants

```
#include <vector>
#include <string>



static const std::vector<std::string> CONSTANTS = {
    "Hello",
    "Word",
    "!"
};
```

# Global constants

```
#include <string_view>
```

```
static const std::string_view CONSTANTS[] = {
    "Hello",
    "Word",
    "!"
};
```

# Global constants

```
#include <string_view>




constexpr std::string_view CONSTANTS[] = {
    "Hello",
    "Word",
    "!"
};
```

# Inheritance

```
// base.hpp


struct base {
    // ...
    virtual void act() = 0;
    virtual ~base(){}
};
```

# Inheritance

```
// something.cpp


struct some_implementation: base {
    // ...
    void act() { /* ... */ }
};
```

# Inheritance

```
// something.cpp


struct some_implementation: base {
    // ...
    void act() override { /* ... */ }      // <======
};
```

# Inheritance

```
// something.cpp


struct some_implementation final: base {      // <======
    // ...
    void act() override { /* ... */ }
};
```

# Inheritance

```
// something.cpp
namespace {                                          // <======
struct some_implementation final: base {
    // ...
    void act() override { /* ... */ }
};
} // anonymous namespace
```

# Move constructors

```
class my_data_struct {
    std::vector<int> data_;
public:
    // ...
    my_data_struct(my_data_struct&& other)
        : data_(other.data_)
    {}
};
```

# Move constructors

```
class my_data_struct {
    std::vector<int> data_;
public:
    // ...
    my_data_struct(my_data_struct&& other)
        : data_(std::move(other.data_))   // <======
    {}
};
```

# Move constructors

```
class my_data_struct {
    std::vector<int> data_;
public:
    // ...
    my_data_struct(my_data_struct&& other) noexcept       // <======
        : data_(std::move(other.data_))
    {}
};
```

# Move constructors

```
class my_data_struct {
    std::vector<int> data_;
public:
    // ...
    my_data_struct(my_data_struct&& ) = default;     // <======
};
```

# Caching

# Caching divs

```
// .h
extern const double fractions[256];


// .cpp
const double fractions[256] = { 1.0 / i, ... };
```

# Caching divs

```
      2 cycles        - MOV r32/64,m
=>    5 cycles        - L1 Data Cache Latency for complex address calcs (p[n])
      12 cycles       - L2 Cache Latency
=>    22-29 cycles    - DIV r32
      36-43 cycles    - L3 Cache Latency
```

# Caching divs

5 cycles vs 22-29 cycles

# Caching divs

5 cycles vs 22-29 cycles

x2-x4 faster

# Caching divs

5 cycles vs 22-29 cycles

x2-x4 faster

WOW!!!

# Caching divs

5 cycles vs 22-29 cycles

x2-x4 faster

WOW!!!

Чё правда?

# Caching divs (недостатки)

- Оптимизатор не видит значения

# Caching divs (недостатки)

- Оптимизатор не видит значения:
  - нет возможности оптимизировать ближайшие инструкции

# Caching divs (недостатки)

- Оптимизатор не видит значения:
  - нет возможности оптимизировать ближайшие инструкции
  - нет возможности constexpr

# Caching divs (недостатки)

- Оптимизатор не видит значения:
  - нет возможности оптимизировать ближайшие инструкции
  - нет возможности constexpr
  - aliasing (ссылка на double может указывать на ту же ячейку памяти)

# Caching divs (недостатки)

- **Серьёзное негативное влияние на производительность:**
  - нет возможности оптимизировать ближайшие инструкции
  - нет возможности constexpr
  - aliasing (ссылка на double может указывать на ту же ячейку памяти)

# Caching divs (недостатки)

- Серьёзное негативное влияние на производительность:
  - нет возможности оптимизировать ближайшие инструкции
  - нет возможности constexpr
  - aliasing (ссылка на double может указывать на ту же ячейку памяти)
- Мы потратили кучу времени на обсуждение этого...

# Caching divs (недостатки)

- Серьёзное негативное влияние на производительность:
  - нет возможности оптимизировать ближайшие инструкции
  - нет возможности constexpr
  - aliasing (ссылка на double может указывать на ту же ячейку памяти)
- Мы потратили кучу времени на обсуждение этого...
- Потратили 1Kb L1 кеша из 16/32Kb

# Caching divs (недостатки)

Потратили 1Kb L1 кеша из 16/32Kb

"Зачастую, **производительность системы ограничена не скоростью процессора**, а производительностью подсистем памяти. В то время, как традиционно компиляторы уделяли большее внимание оптимизации работы процессора, **сегодня больший упор следует делать на более эффективное использование иерархии памяти**." [1]

[1] Ахо, Сети, Ульман. Компиляторы. Принципы, технологии, инструменты. 2ed.2008

# Caching divs (недостатки)

Потратили 1Kb L1 кеша из 16/32Kb

x10 прирост производительности без изменения инструкций [1] => **Кеш очень важен**

[1] Ulrich Drepper. What Every Programmer Should Know About Memory. 2007

# Идаельный велосипед

# Велосипед должен быть

- Кроссплатформенный

# Велосипед должен быть

- Кроссплатформенный
- Быстрый

# Велосипед должен быть

- Кроссплатформенный
- Быстрый
- Полезный

# Велосипед должен быть

- Кроссплатформенный
- Быстрый
- Полезный
- Функциональный

# РГ21

https://stdcpp.ru/proposals

# РГ21

https://stdcpp.ru/proposals

- Поможем с формальностями
- Поможем с принятием в Boost
- Поможем с написанием proposal
- Защитим ваши интересы на заседнии, передадим отзывы

# РГ21. Идеи в разработке:

# РГ21. Идеи в разработке:

- P0539R0: wide_int by Игорь Клеванец

# РГ21. Идеи в разработке:

- P0539R0: wide_int by Игорь Клеванец
- P0275R1: Dynamic library load

# РГ21. Идеи в разработке:

- P0539R0: wide_int by Игорь Клеванец
- P0275R1: Dynamic library load
- ConcurrentHashMap by Сергей Мурылев, Антон Малахов

# РГ21. Идеи в разработке:

- P0539R0: wide_int by Игорь Клеванец
- P0275R1: Dynamic library load
- ConcurrentHashMap by Сергей Мурылев, Антон Малахов
- Constexpr: “D0202R2: Constexpr algorithm”, std::array, <numeric>, ...

# РГ21. Идеи в разработке:

- P0539R0: wide_int by Игорь Клеванец
- P0275R1: Dynamic library load
- ConcurrentHashMap by Сергей Мурылев, Антон Малахов
- Constexpr: “D0202R2: Constexpr algorithm”, std::array, <numeric>, ...
- ?dlsym?

# РГ21. Идеи в разработке:

- P0539R0: wide_int by Игорь Клеванец
- P0275R1: Dynamic library load
- ConcurrentHashMap by Сергей Мурылев, Антон Малахов
- Constexpr: “D0202R2: Constexpr algorithm”, std::array, <numeric>, ...
- ?dlsym?
- Stacktrace

# РГ21. Идеи в разработке:

- P0539R0: wide_int by Игорь Клеванец
- P0275R1: Dynamic library load
- ConcurrentHashMap by Сергей Мурылев, Антон Малахов
- Constexpr: “D0202R2: Constexpr algorithm”, std::array, <numeric>, ...
- ?dlsym?
- Stacktrace
- 100500 мелочей, которые делают все в ISO WG21

# Спасибо! Вопросы?

https://stdcpp.ru/