Цена абстракции

Константин Владимиров, Syntacore, 2024

Механизмы абстракции

- Абстракция является процессом обобщения и сокрытия конкретных деталей, позволяющим сосредоточиться на более важных высокоуровневых задачах.

```
const auto &pivot = arr[low];
int i = high;

for (int j = high; j > low; j--)
  if (arr[j] > pivot) {
    auto tmp = arr[i];
    arr[i] = arr[j];
    arr[j] = tmp;
    i -= 1;
  }
```

- Простейший механизм абстракции это вынос функции. Какую функцию вы бы вынесли?

Вынос функции

- Абстракция является процессом обобщения и сокрытия конкретных деталей, позволяющим сосредоточиться на более важных высокоуровневых задачах.

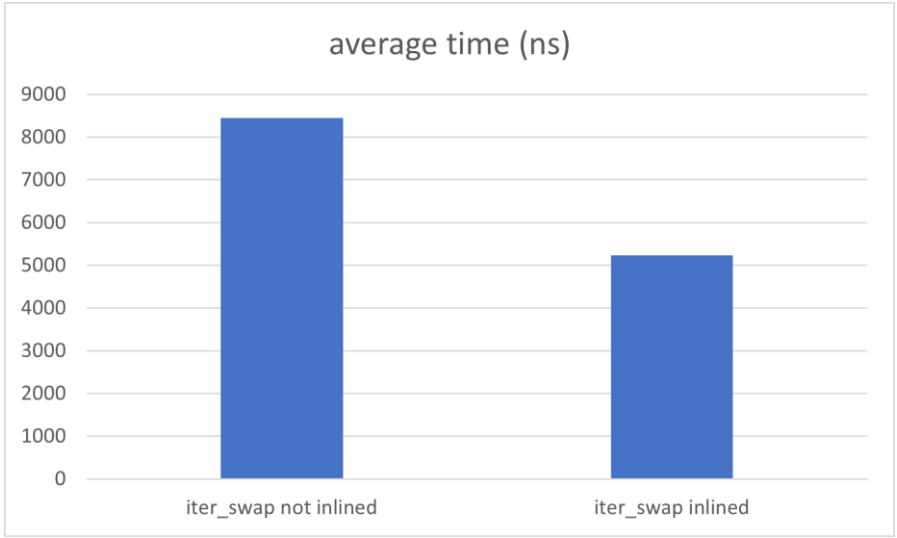
```
const auto &pivot = arr[low];
int i = high;

for (int j = high; j > low; j--)
  if (arr[j] > pivot) {
    std::iter_swap(arr + i, arr + j);
    i -= 1;
  }
```

- Внезапно такой вынос функции делает код не только проще но и **потенциально лучше**. Например внутри iter_swap может быть использована семантика перемещения.
- Ещё лучший вариант: заменить весь алгоритм на std::partition.

Разумная цена абстракции.

- Вынос функции делает хуже, если короткая функция не подставилась. Тогда у абстракции появляется **цена**.
- Мы должны учитывать, что:
  - Компилятор не обязан делать подстановку.
  - Вы можете написать код так, что компилятор не сможет сделать подстановку.
- Назовём **разумной ценой** абстракции её цену в тех случаях когда вы её правильно применили и компилятор сделал всё, что мог.
- За любую абстракцию можно заплатить неразумную цену если применять её неосторожно.



i5-1135G7

https://github.com/tilir/benchmarks/tree/master/inline

Отчего происходят проблемы при отсутствии инлайна

```
.LBB0_4:                                #
        dec     r13
        add     r15, -4
        cmp     r13, r12
        jle     .LBB0_5
.LBB0_2:                                # =
        cmp     dword ptr [r15], ebp
        jle     .LBB0_4
        movsxd  rbx, ebx
        lea     rdi, [r14 + 4*rbx]
        mov     rsi, r15
        call    iter_swap(int*, int*)@PLT
        dec     ebx
        jmp     .LBB0_4
```

```
.LBB0_11:                               #   in Loop
        add     rsi, -2
        cmp     rsi, rcx
        jle     .LBB0_12
.LBB0_7:                                # =>This In
        mov     r8d, dword ptr [rdi + 4*rsi]
        cmp     r8d, edx
        jle     .LBB0_9
        cdqe
        mov     r9d, dword ptr [rdi + 4*rax]
        mov     dword ptr [rdi + 4*rax], r8d
        mov     dword ptr [rdi + 4*rsi], r9d
        dec     eax
.LBB0_9:                                #   in Loop
        mov     r8d, dword ptr [rdi + 4*rsi - 4]
        cmp     r8d, edx
        jle     .LBB0_11
```

https://godbolt.org/z/fWeYGc5sv

Три игрока в цене абстракции

- Микроархитектура.
  - Одни и те же инструкции могут иметь разную относительную цену на разных вычислительных платформах.
- Компилятор и библиотеки.
  - Один и тот же код может быть по разному соптимизирован разными компиляторами (или в разных библиотеках).
  - Пример по ссылке внизу слайда. GCC 13.2 генерирует 331 ассемблерную инструкцию, Clang 17.0.1 генерирует 26 ассемблерных инструкций.
- Семантика языка программирования.
  - Например наличие в языке шаблонов магическим образом влияет на качество стандартной сортировки.

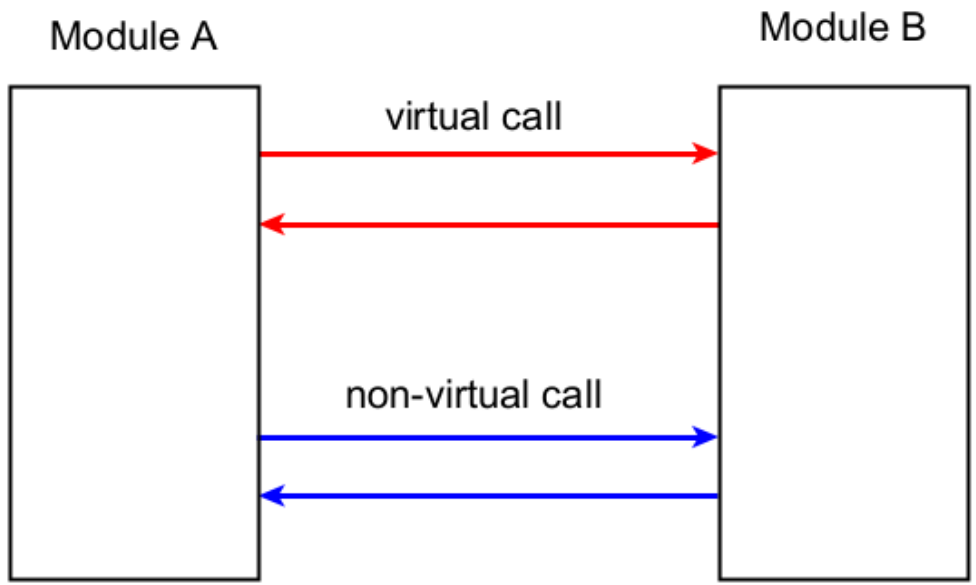
https://godbolt.org/z/4xsveWM7z

Предварительные решения

- Дональд Кнут писал, что предварительная оптимизация это корень зла.
- Но в наших проектах мы должны принимать большие решения относительно используемых механизмов и изменять их потом может быть очень дорого.
- Ниже перечислены механизмы абстракции специфичные для языка C++. Каждая из них это проектное решение.
  - Виртуальные функции (virtual functions).
  - Исключения (exceptions).
  - Стандартные диапазоны (ranges).
  - Сопрограммы (coroutines).
- Какие из них кажутся вам дорогими? Какие могут иметь отрицательную стоимость?

Начнём с виртуальных функций

```
int foo(int x) maybe override;
int bar(int x) override or not {
  return (x > 0) ? foo(x - 1) : 0;
}
```



- Виртуальный вызов.

```
mov rax, QWORD PTR [rdi]
jmp [QWORD PTR [rax]]
```

- Невиртуальный вызов.

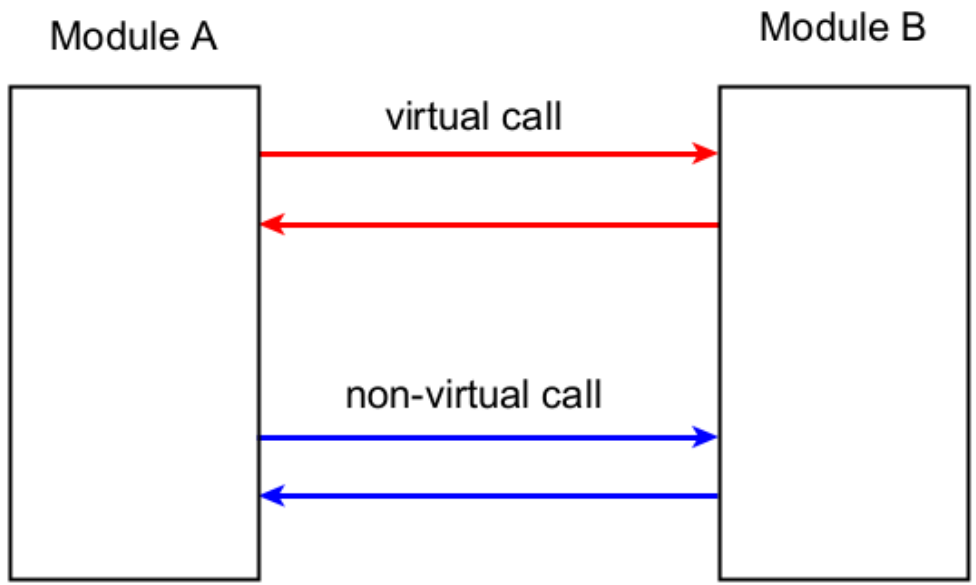
```
jmp S::foo
```

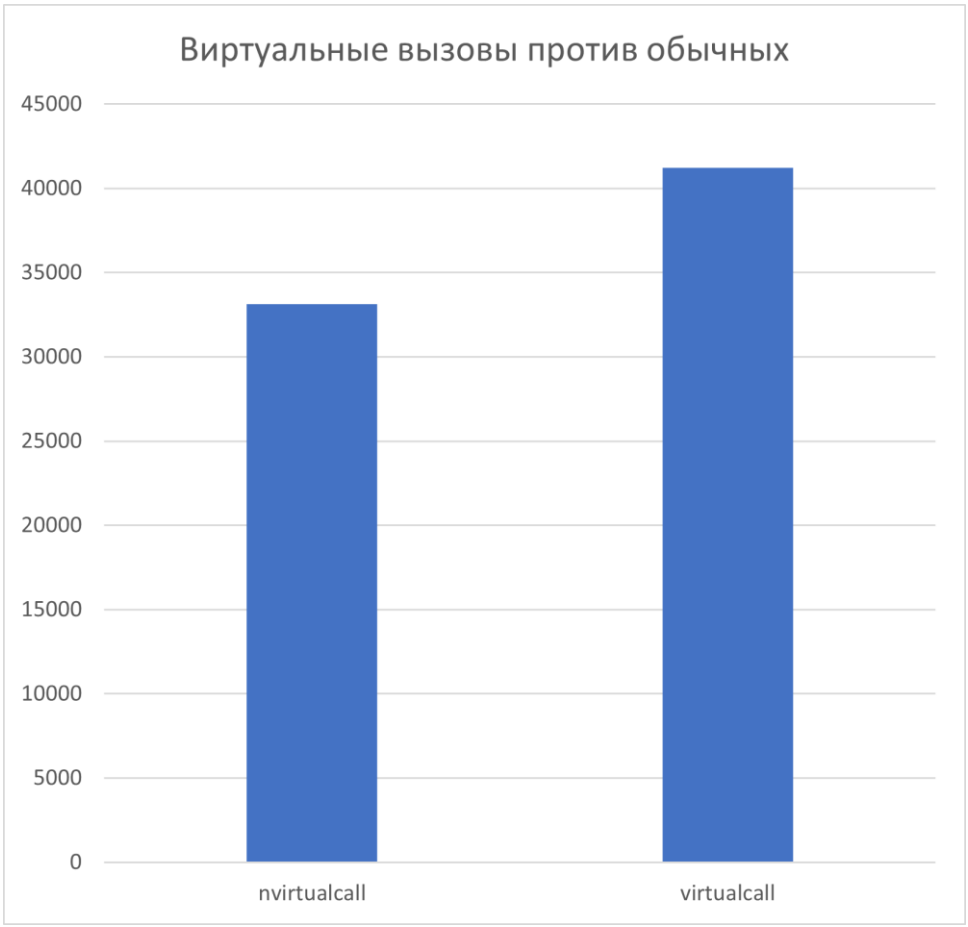
- Невиртуальный вызов выглядит существенно проще и не идёт дополнительно через память.
- Ожидаем ли мы существенной разницы?

Начнём с виртуальных функций

```
int foo(int x) maybe override;

int bar(int x) override or not {
  return (x > 0) ? foo(x - 1) : 0;
}
```





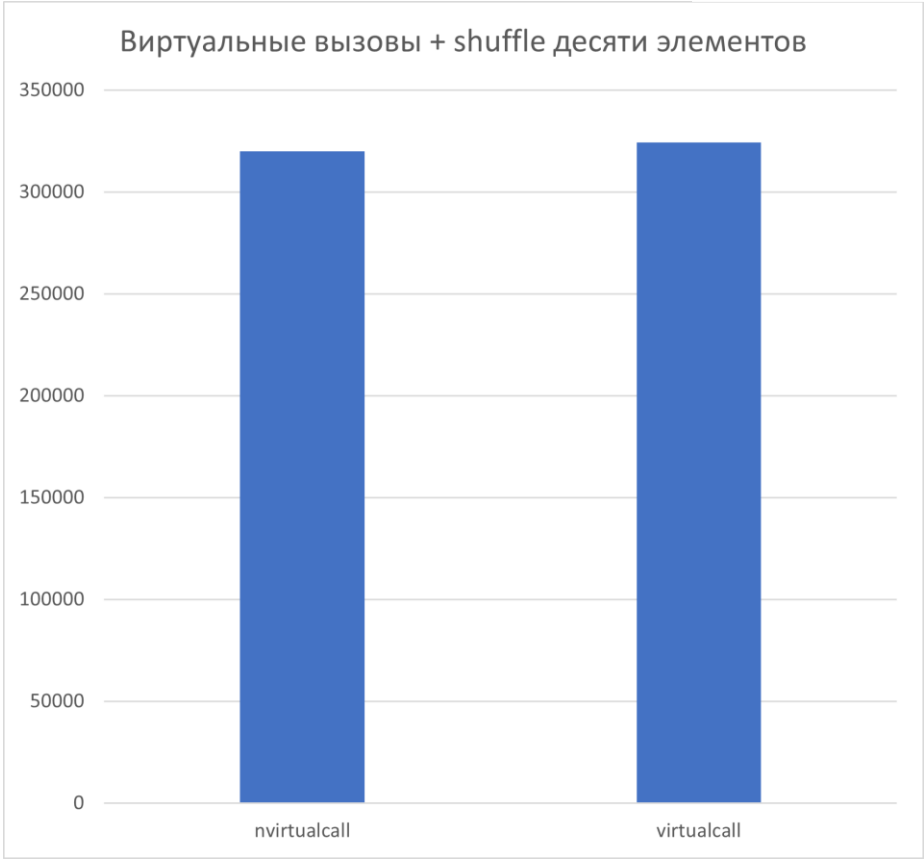
https://github.com/tilir/benchmarks/tree/master/virtual-overhead

Добавим совсем небольшой shuffle

```
struct NonVirt {
  std::array<int, 10> a;
  std::mt19937 g;
  // .....
};

int NonVirt::foo(int x) {
  std::shuffle(a.begin(), a.end(), g);
  return (x > 0) ? bar(x - 1) : 0;
}

int NonVirt::bar(int x) {
  std::shuffle(a.begin(), a.end(), g);
  return (x > 0) ? foo(x - 1) : 0;
}
```



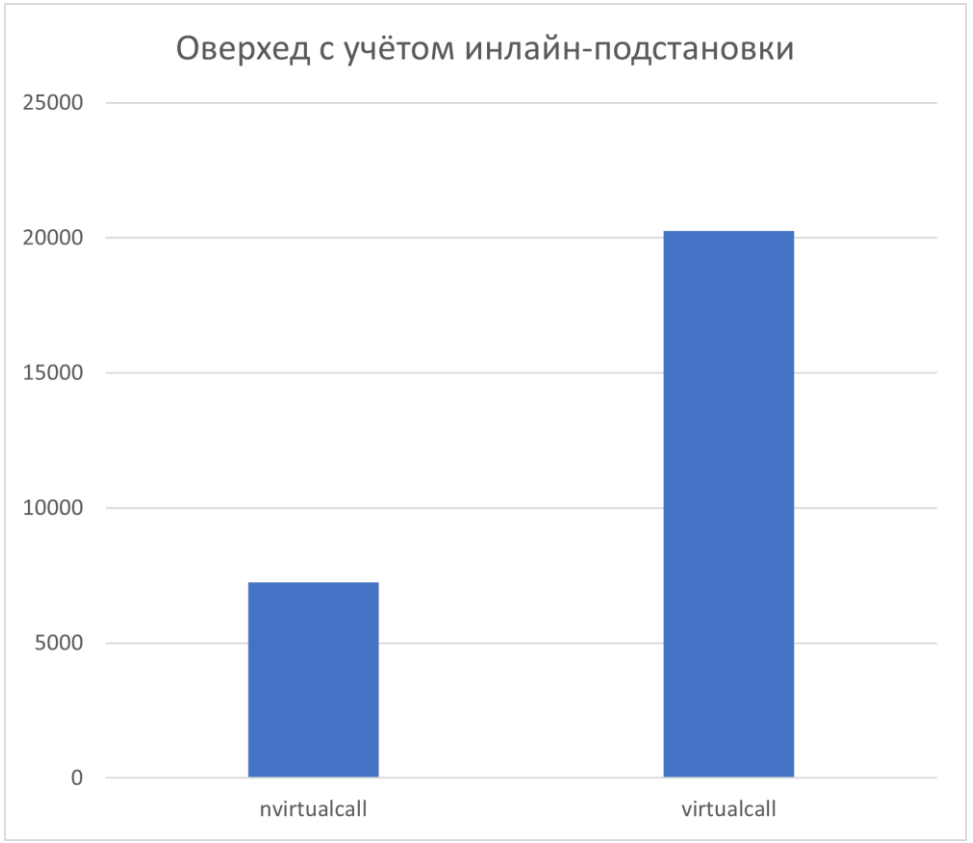
https://github.com/tilir/benchmarks/tree/master/virtual-overhead

Виртуальные функции и инлайн

- Главная проблема производительности виртуальных функций – их влияние на инлайн.

```
Derived *vd = откуда-то получили;
int sum = 0;
for (int i = 0; i < NBMK; ++i)
  sum += vd->bar(NCALL);
```

- Мы тут не можем проинлайнить Derived::bar так как не уверены в том что это не один из пока не известных нам наследников.
- Это несколько митигируется возможностями **девиртуализации** в современных компиляторах.



https://github.com/tilir/benchmarks/tree/master/virtual-inline

Девиртуализация

- В некоторых случаях компилятор справляется с девиртуализацией очень легко.

```
struct Base {
  virtual int foo();
};
struct Derived final {
  int foo() override { return 42; }
};
int bar(Derived *vd) {
  return vd->foo(); // devirtualized, inlined!
}
```

- Что если мы не хотим делать final всю структуру?

Девиртуализация

- В некоторых случаях компилятор справляется с девиртуализацией очень легко.

```
struct Base {
  virtual int foo();
};
struct Derived {
  int foo() override final { return 42; }
};
int bar(Derived *vd) {
  return vd->foo(); // devirtualized, inlined!
}
```

- Можно сделать вопрос на собеседовании: "когда override final имеют смысл вместе"?

Спекулятивная девиртуализация

- Представим ситуацию когда виртуальный вызов может быть девиртуализован только если нам повезло.

```
struct Derived : Base {
  int bar() override {
    return 42;
  }
  int foo(Base *b) override {
    // if (b is Derived)
    //   return this->bar();
    return b->bar();
  }
};
```

```
Derived::foo(Base*):
        mov     rax, QWORD PTR [rsi]
        mov     rax, QWORD PTR [rax]
        cmp     rax, OFFSET FLAT:Derived::bar()
        jne     .L5
        mov     eax, 42
        ret
.L5:
        mov     rdi, rsi
        jmp     rax
```

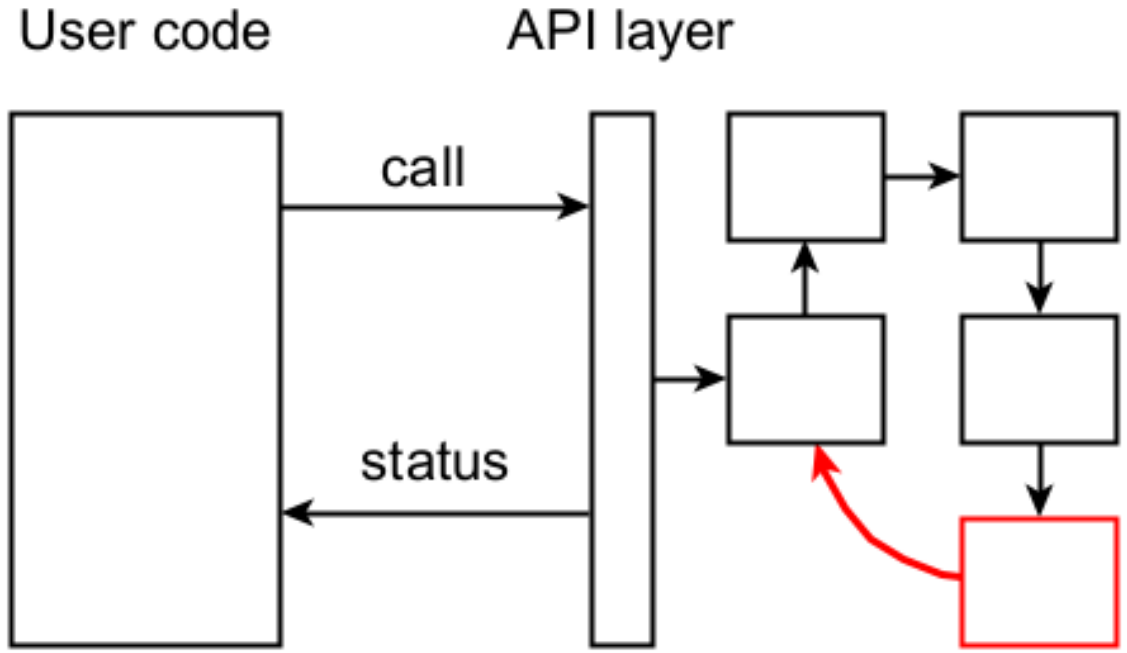
https://godbolt.org/z/MTxr7hTGe

Резюме по виртуальным функциям

- Виртуальные функции отлично изучены и прекрасно поддержаны в компиляторах.
- Ключевое слово final позволяет нам до некоторой степени управлять девиртуализацией.
- Главная проблема виртуальных функций – плохой инлайн.
- Если в вашей программе нет коротких и лёгких виртуальных функций с многоуровневыми вызовами, вы, скорее всего, ничего не платите.

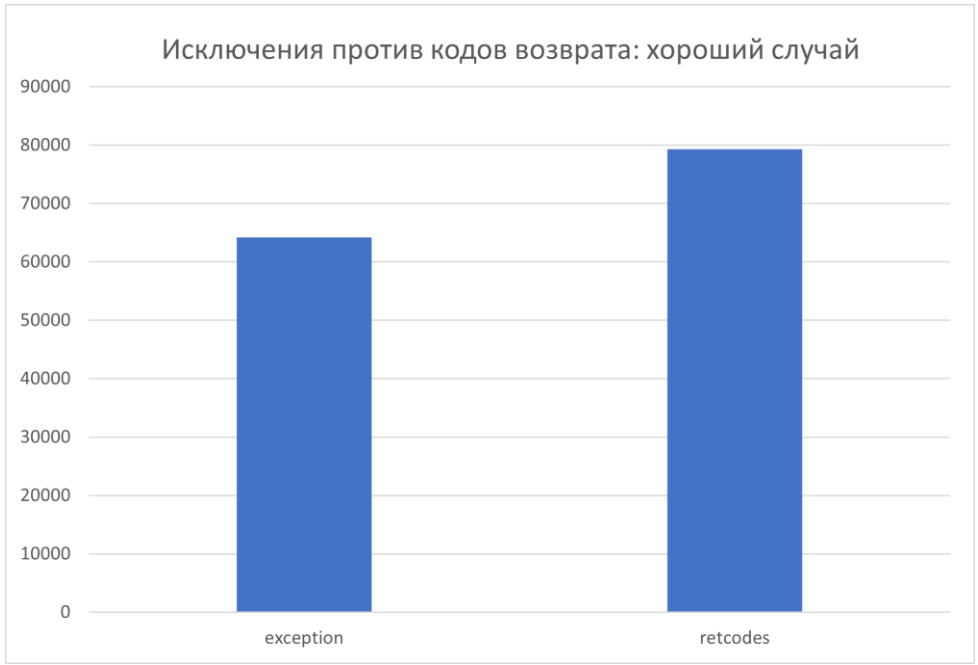
Исключения

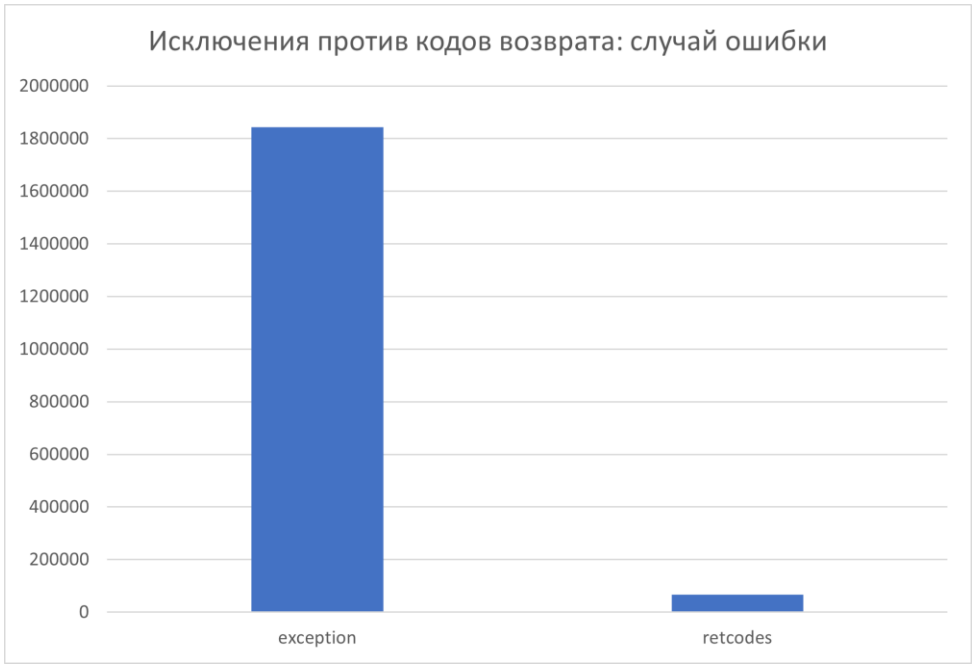
- Обработка ошибок в конструкторах.
- Обработка ошибок в перегруженных операторах.
- Невозможность тихого игнорирования ошибки.
- Выход из сложного потока управления в точку принятия решения с размоткой стека.
- Исключения это пожалуй самый критикуемый механизм C++



Исключения против кодов возврата

- Насколько дорого путешествие кодами возврата против исключения?
- Если исключение произошло – очень дорого. Но если нет, то мы выигрываем не исполняя бесчисленные проверки.





https://github.com/tilir/benchmarks/tree/master/excret

Исключения не бесплатны даже если их не кидать?

- Компилятор должен создать некоторый код для исполнения при бросании исключения.

```
struct S {
  int foo();
  ~S();
};
void bar() {
  S s;
  s.foo();
}
```

- Насколько существенен этот оверхед?

```
bar():
        push    rbx
        sub     rsp, 16
        lea     rdi, [rsp + 15]
        call    S::foo()@PLT
        lea     rdi, [rsp + 15]
        call    S::~S()@PLT
        add     rsp, 16
        pop     rbx
        ret
        mov     rbx, rax
        lea     rdi, [rsp + 15]
        call    S::~S()@PLT
        mov     rdi, rbx
        call    _Unwind_Resume@PLT
```

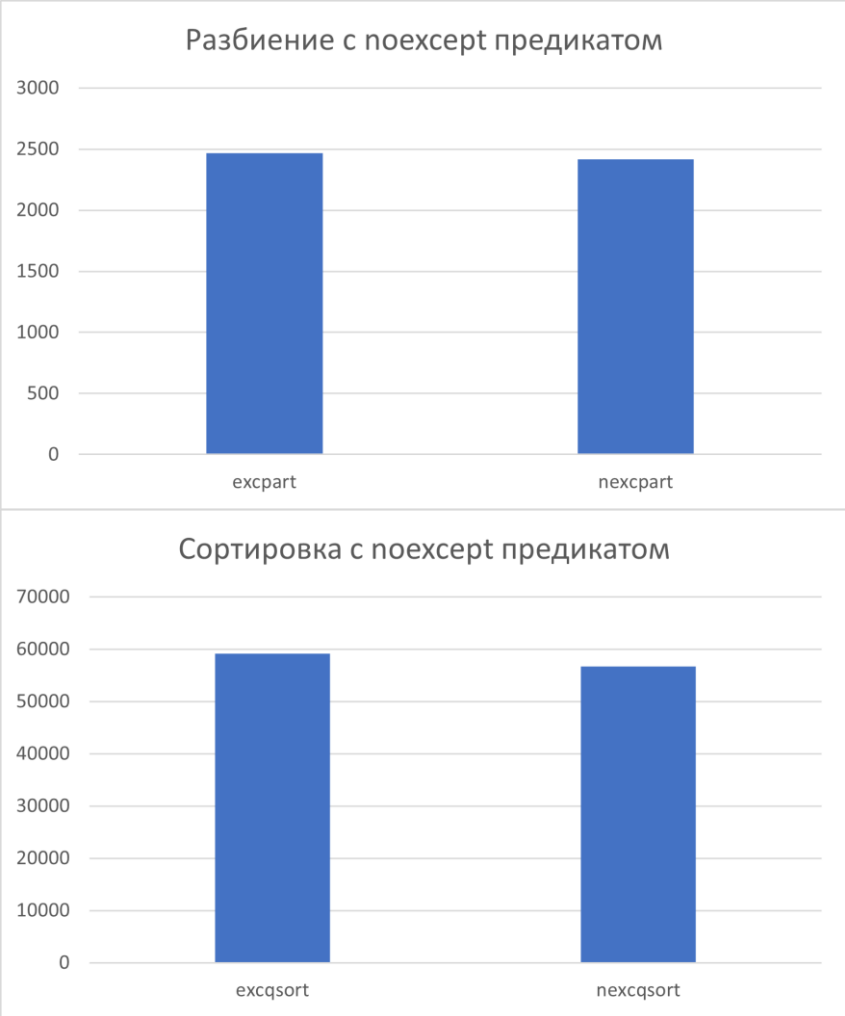
https://godbolt.org/z/oe9ozTqMd

Noexcept в реалистичном примере

- Реалистичный пример это std::sort в котором может быть noexcept предикат.

```
auto nep = [] (int x, int y) noexcept {
   return x < y;
}
```

- Ещё один реалистичный пример это std::partition.
- Предикат сортировки очень простая функция, он часто инлайнится и много оптимизируется.
- Можно видеть, что наличие noexcept не очень заметно в обоих примерах.



https://github.com/tilir/benchmarks/tree/master/noexcept-qsort

Резюме по исключениям

- Запрет исключений в вашей программе означает одну из двух вещей.
  1. Либо вы готовы руками прокинуть коды возврата к слою API.
  2. Либо ваш код нельзя использовать как библиотеку.
- Если исключение вылетело, это очень дорого. Прокидывание назад кодов возврата внезапно может быть гораздо дешевле даже несмотря на существенное увеличение кода.
- Оверхед на упущенные оптимизации в компиляторе не так велик (по сравнению с другими источниками оверхеда).
- Вызов любой не-noexcept функции приводит к построению landing pad.

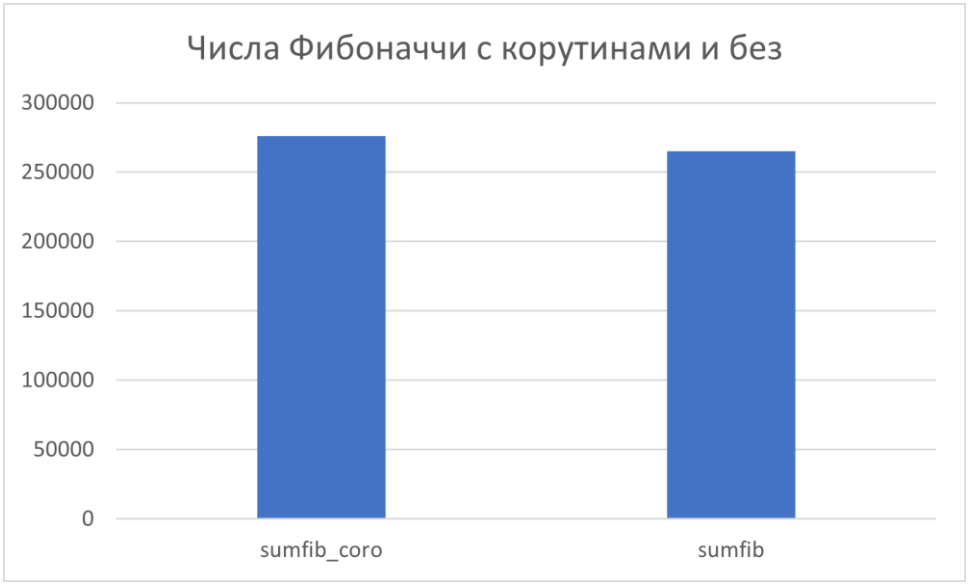
Корутины: числа Фибоначчи

- Базовый пример для работы генераторов.

```
constexpr int K = 10;

generator<int> fibs() {
  int a = 1, b = 0;

  for (;;) {
    co_yield b;
    b = std::exchange(a, a + b) % K;
  }
}
```

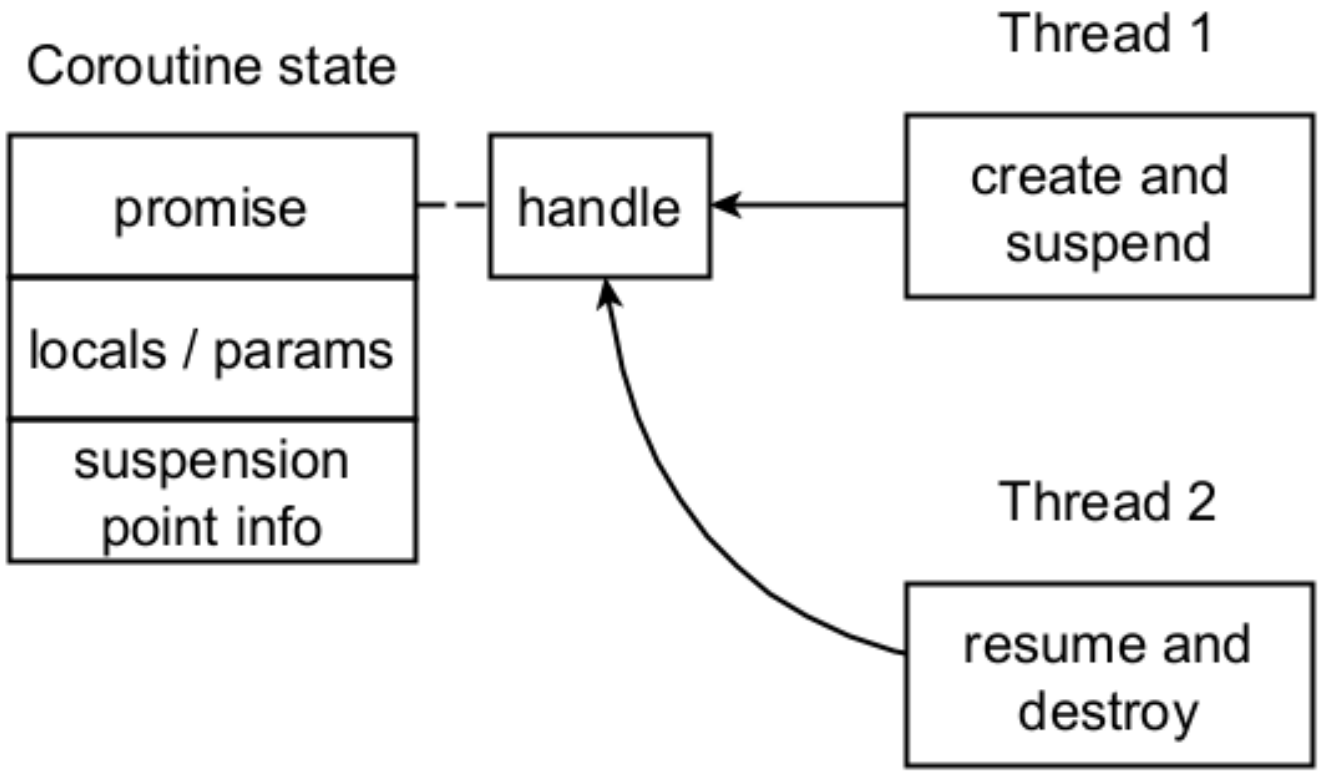


- Может ли быть большая просадка? Чем она может быть продиктована – принципиальными проблемами корутин или недостатками компилятора?

https://github.com/tilir/benchmarks/tree/master/coro-fibs

Главная проблема оптимизации корутин

- Главной проблемой оптимизации корутин является их главное преимущество.
- Корутина может быть приостановлена в одном потоке и проснуться на другом.
- Поэтому по определению её фрейм не может жить на стеке в общем случае.
- В некоторых условиях он может быть соптимизирован и помещён на стек.

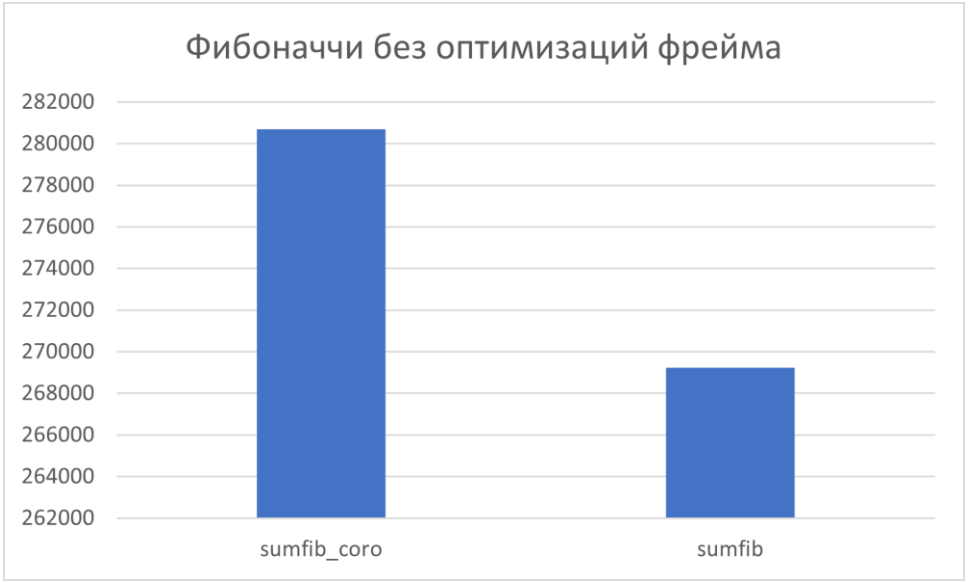


https://godbolt.org/z/6odT34sd3

Условия оптимизации корутин

- Если одновременно:
  1. Время жизни состояния корутины (а значит и её фрейма) вложено в область видимости её вызывающей функции.
  2. И при этом размер фрейма известен на этапе компиляции.
- То фрейм корутины может быть аллоцирован на стеке вызывающей функции.

```
int sumfib_coro(int n) {
   auto nums = fibs();
   // работа с генератором
   nums.move_next();
} // тут фрейм корутины умирает
```



Старые знакомые возникают в продолжении

- Следующий код выглядит совершенно безобидно.

```
Generator<bool>
sequence(unsigned n) {
  size_t cnt = 0; int i;

  for (i = 0; i < n; i++) {
    auto sqrt = std::sqrt(i);
    co_yield(sqrt > cnt);

    if (sqrt > cnt)
        ++cnt;
  }
}
```

```
int main() {
  auto gen = sequence(10);
  int sum = 0;
  while(gen)
    sum += gen();
  cout << sum << endl;
}
```

- Но возникает существенная разница в этом коде с исключениями и без исключений.

https://godbolt.org/z/nq876cG1e

Резюме по корутинам

- Тема для компиляторов новая и многие оптимизации пока что применяются очень неровно.
- Есть простые и достаточные условия, которые делают корутинные генераторы теоретически нулевыми по стоимости (если справится инлайнер).
- Взаимодействие с исключениями у корутин сложное и неоднозначное.
- Нет возможности руками управлять происходящим. Держите пальцы крестиком и бенчмаркайте ваш код.

Ranges: transform + filter

- Задача стандартная: перебросить из одного контейнера в другой, по дороге отфильтровать.

```
dst.reserve(src.size());
for (auto& elt : src)
  if (f(elt))
    dst.push_back(g(elt));
```

- Задачу можно решать через вспомогательный контейнер (planar solution).

```
std::vector<int> v; v.reserve(src.size());
std::copy_if(src.begin(), src.end(), std::back_inserter(v), f);
dst.resize(v.size());
std::transform(v.begin(), v.end(), dst.begin(), g);
```

Ranges: transform + filter

- Задача стандартная: перебросить из одного контейнера в другой, по дороге отфильтровать.

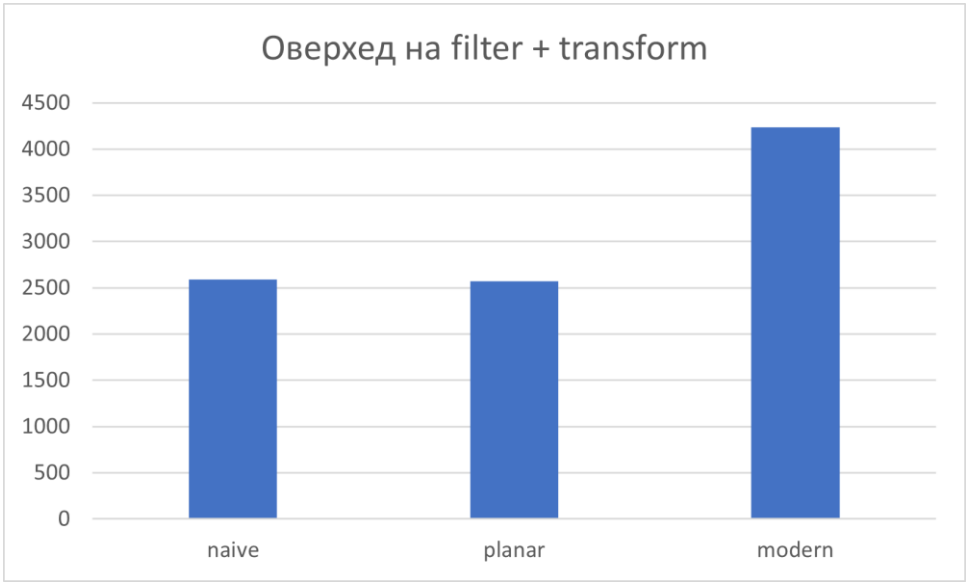
```
dst.reserve(src.size());
for (auto& elt : src)
  if (f(elt))
    dst.push_back(g(elt));
```

- Задачу можно решать на более современный манер (modern solution).

```
dst.reserve(src.size());
auto v = views::all(src) | views::filter(f) |
         views::transform(g);
ranges::copy(v, dst.begin());
```

Неутешительные результаты

- Современный способ существенно проигрывает.
- Является ли это
  1. Неизбежным проигрышем
  2. Случайностями микроархитектуры
  3. Проблемами компилятора.
- На что бы вы здесь поставили?




```
dst.reserve(src.size());
auto v = views::all(src) | views::filter(f) | views::transform(g);
ranges::copy(v, dst.begin()); // что здесь происходит?
```

https://github.com/tilir/benchmarks/tree/master/ranges-filter

Наивная реализация filter_view

- Мы могли бы подумать, что filter_view реализован как-то вот так.

```
template<input_range R, unary_predicate Fn>
class filter_view : public view_interface<filter_view<R, Fn>> {
  R base; Fn f;

public:
   Iterator begin() {
     auto it = ranges::find_if(base, f); // O(N)
     return {this, std::move(it)};
   }
```

- Увы стандарт недвусмысленно требует begin за $O(1)$.

Неизбежный оверхед на filter_view по памяти

- По проектированию filter_view обязан кешировать часть состояния.

```
template<input_range R, unary_predicate Fn>
class filter_view : public view_interface<filter_view<R, Fn>> {
  R base; Fn f;
  std::optional<Iterator> cached_begin;

public:
   Iterator begin() {
     if (cached_begin.has_value)
       return {this, cached_begin.get()};
     auto it = ranges::find_if(base, f); // O(1)+ amortized
     return cached_begin.set_value({this, std::move(it)});
   }
```

Детали комбинации

- Теперь посмотрим как работает комбинация.

```
views::all(src) | views::filter(f) | views::transform(g)
```

- Что такое pipe?

```
template <typename R>
auto operator | (R r, const to_closure& c) { return c(r); }
```

- Это функциональная композиция вывернутая наизнанку.

```
views::transform(views::filter(views::all(src), f), g);
```

- В итоге мы имеем
  1. кеширующий фильтр, вытягивающий данные из all
  2. transform, вытягивающий из фильтра.

Цена проектора

- Проекторы это ещё одна замечательная возможность ranges.

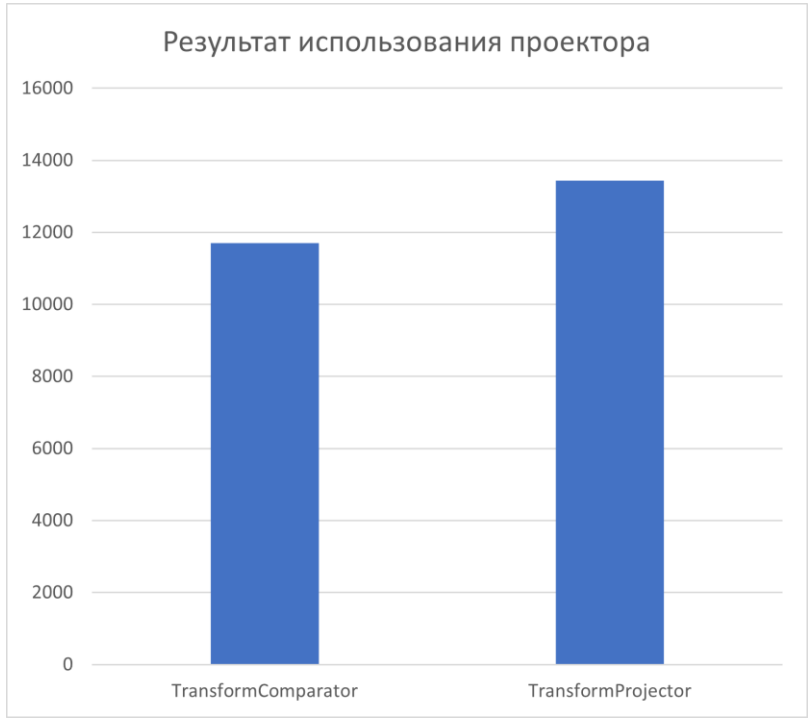
```
struct S { int x, y; };

std::vector<S> v;

ranges::transform(v.begin(), v.end(),
    w.begin(),
    [](const auto& a) { return a.x * 2; });
```

- Вариант с проектором несколько упрощает предикат.

```
ranges::transform(v.begin(), v.end(),
   w.begin(),
   [](auto x) { return x * 2; },
   &S::x);
```



Резюме по ranges

- Неизбежное кеширующее поведение и особенности функциональной композиции пока что ставят компиляторы в тупик.
- Принципиальных проблем с тем чтобы стать zero-cost абстракцией у диапазонов нет, но путь выглядит сложным и не близким.
- Никаких способов управлять этим нет, бенчмаркаем код и держим пальцы крестиком.

В поисках отрицательной стоимости

- Единственный рассмотренный сегодня механизм абстракции в котором я нашёл какое-то подобие отрицательной стоимости это исключения.
- Так ли это? Хотите повторить и расширить мои эксперименты?

```
conan install conanfile.txt --build=missing

cmake -S . -B build/Release \
    --toolchain build/Release/generators/conan_toolchain.cmake \
    -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_COMPILER=g++-12

cmake --build build/Release

env CTEST_OUTPUT_ON_FAILURE=1 \
cmake --build build/Release --target test --parallel 1
```

- Слово за вами.

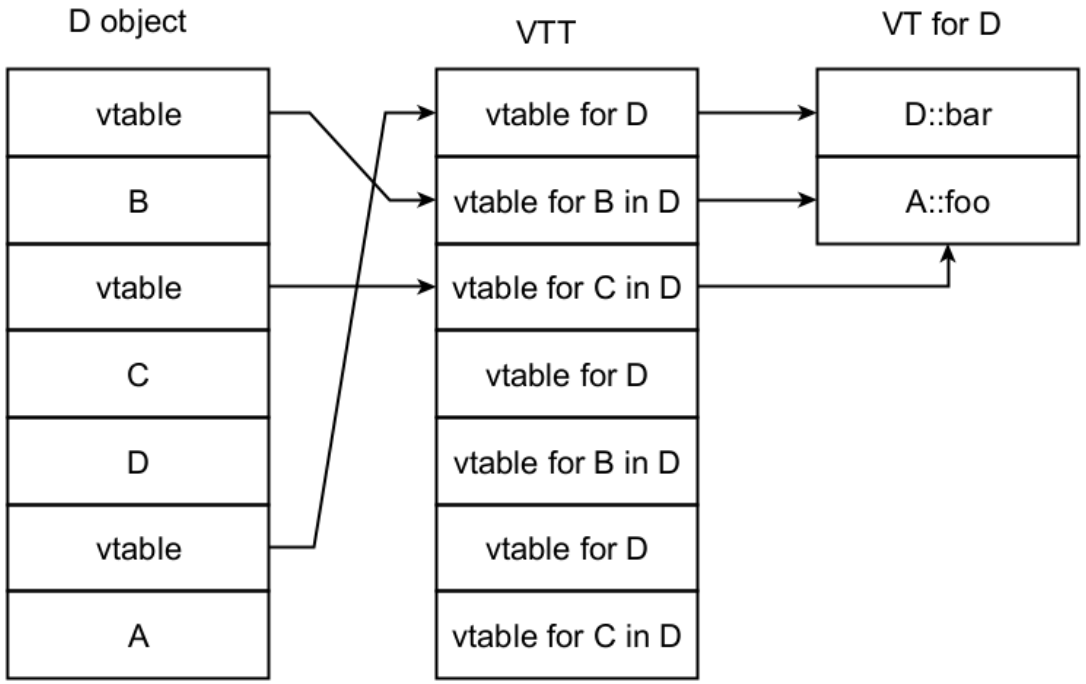
https://github.com/tilir/benchmarks

Q & A

Backup: виртуальное наследование

- Сильно ли отличаются затраты при виртуальном наследовании?

```
struct A { virtual int foo(); };
struct B : virtual public A {
  int x;
};
struct C : virtual public A {
  int y;
};
struct D : public B, public C {
  virtual int bar();
}
```

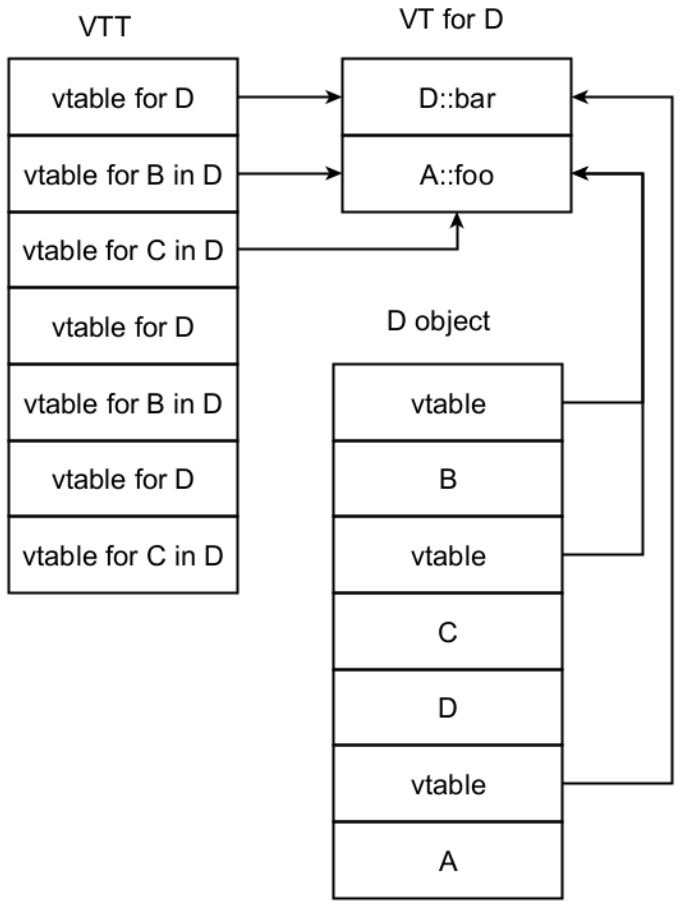


- Казалось бы теперь виртуальная таблица вынуждена быть двухуровневой.

Виртуальное наследование

- После завершения работы конструктора объекта, таблица виртуальных функций для каждого из подобъектов становится одноуровневой.
- VTT используется только на этапе конструирования.

```
        mov     ebx, NBMKS
.L2:
        mov     rax, QWORD PTR [rbp+0]
        mov     esi, NCALLS
        mov     rdi, rbp
        call    [QWORD PTR [rax]]
        add     r12d, eax
        sub     ebx, 1
        jne     .L2
```



https://godbolt.org/z/9bdGq8aYq

https://ww2.ii.uj.edu.pl/~kapela/pn/cpp_vtable.html