Преждевременная оптимизация - зло! Да здравствует преждевременная оптимизация!

Андрей Карпов

CTO, PVS-Studio

karpov@viva64.com

www.viva64.com

Преждевременная оптимизация — корень всех зол

Статья Дональда Кнута «Structured Programming with go to Statements» в сборнике «Computing Surveys» (Vol. 6, № 4, декабрь 1974, стр. 268).

"Premature optimization is the root of all evil."

Преждевременная оптимизация

- Код более сложный
- Сложный код более подвержен ошибкам
- Трата времени на код, который может не понадобиться
- Кто сказал, что ваша оптимизация вообще что-то делает?

```
static int rr_cmp(uchar *a,uchar *b)
{
  if (a[0] != b[0])
    return (int) a[0] - (int) b[0];
  if (a[1] != b[1])
    return (int) a[1] - (int) b[1];
  if (a[2] != b[2])
    return (int) a[2] - (int) b[2];
  if (a[3] != b[3])
    return (int) a[3] - (int) b[3];
  if (a[4] != b[4])
    return (int) a[4] - (int) b[4];
  if (a[5] != b[5])
    return (int) a[1] - (int) b[5];
  if (a[6] != b[6])
    return (int) a[6] - (int) b[6];
  return (int) a[7] - (int) b[7];
}
```

Пример неудачной преждевременной оптимизации

Проект MySQL

На мой взгляд, лучше так

```
int rr_cmp(unsigned char *a, unsigned char *b)
{
  for (int i = 0; i < 7; ++i)
  {
    if (a[i] != b[i])
      return a[i] - b[i];
  }
  return a[7] - b[7];
}
```

https://32bit-me.livejournal.com/124489.html

<table>
<tr>
<td>

```
rr_cmp(unsigned char*, unsigned
char*):
@ @rr_cmp(unsigned char*,
unsigned char*)
        ldrb    r2, [r1]
        ldrb    r3, [r0]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #1]
        ldrb    r3, [r0, #1]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #2]
        ldrb    r3, [r0, #2]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #3]
        ldrb    r3, [r0, #3]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #4]
        ldrb    r3, [r0, #4]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #5]
        ldrb    r3, [r0, #5]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #6]
        ldrb    r3, [r0, #6]
        cmp     r3, r2
        ldrbeq  r1, [r1, #7]
        ldrbeq  r0, [r0, #7]
        subeq   r0, r0, r1
        bxeq    lr
.LBB0_7:
        sub     r0, r3, r2
        bx      lr
```

</td>
<td>

```
rr_cmp(unsigned char*, unsigned
char*):
@ @rr_cmp(unsigned char*,
unsigned char*)
        ldrb    r2, [r1]
        ldrb    r3, [r0]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #1]
        ldrb    r3, [r0, #1]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #2]
        ldrb    r3, [r0, #2]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #3]
        ldrb    r3, [r0, #3]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #4]
        ldrb    r3, [r0, #4]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #5]
        ldrb    r3, [r0, #5]
        cmp     r3, r2
        bne     .LBB0_7
        ldrb    r2, [r1, #6]
        ldrb    r3, [r0, #6]
        cmp     r3, r2
        ldrbeq  r1, [r1, #7]
        ldrbeq  r0, [r0, #7]
        subeq   r0, r0, r1
        bxeq    lr
.LBB0_7:
        sub     r0, r3, r2
        bx      lr
```

</td>
</tr>
</table>

Теория

- Оптимизируйте алгоритм, а не код
- Будет медленно - тогда и оптимизируйте
- Только профайлер!

Когда тайком преждевременно оптимизируешь

Не переживайте!

Мы тоже так делаем!

О докладчике и что мы делаем

- Карпов Андрей Николаевич
- Присутствую на Habr под именем Andrey2008
  habr.com/users/andrey2008/
- Технический директор ООО «СиПроВер»
- Microsoft MVP

- Intel Black Belt Software Developer

- Один из основателей проекта PVS-Studio

Теперь вы не одиноки!

Почему мы сразу занимаемся оптимизацией?

- У нас нет "супер алгоритма"
- У нас есть отдельные алгоритмы сбора данных
- И сотни диагностик

Диагностика

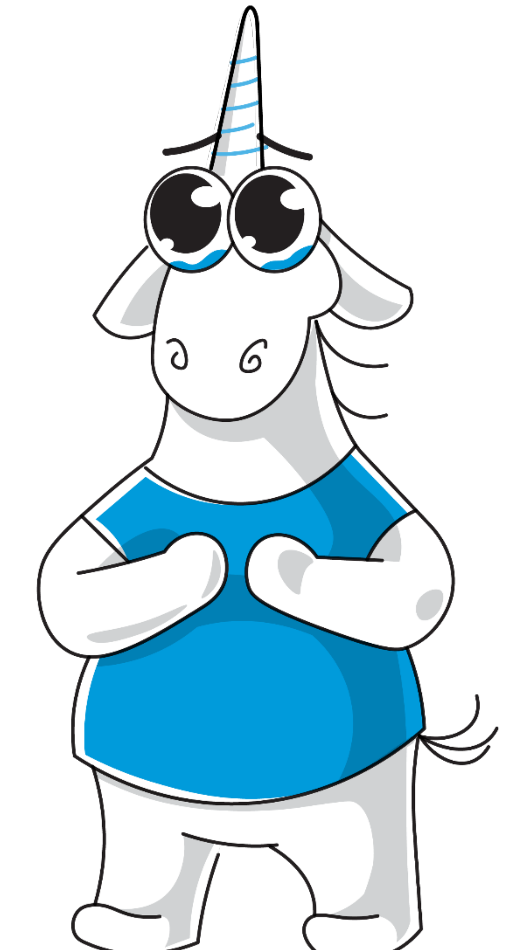
- Добавление одной диагностики никак не сказывается на производительности
- Время одной диагностики меньше погрешности измерения
- Тесты занимают около двух часов и выполняются с погрешностью ±3 минуты
- Предположим, одна неоптимизированная диагностика даёт прибавку в одну минуту
- **Заметить сложно!**

100 диагностик = 100 минут

- Когда диагностики добавляются десятками, замедление ой как заметно!
- Вот только нет виноватого!
- Профайлер вообще не помогает!
- "Процессорное время размазано тонким слоем по сотням функций"
- Нет узкого места...

Про профайлер

- Из него давно выжато, что было возможно
- История про std::string
- Это так мило



«Оптимизация после» невозможна?

- Возможна, но весьма ограниченно
- Иногда видим просадку производительности на одном из проектов, имеющем некую особенность:
  - Длинные строки (26 Мб)
  - Сгенерированный код (тысячи условий в одной функции)
- Главное - нет смысла тянуть с оптимизацией
- Всё, что может работать неэффективно на каких-то входных данных, проявит себя!

Что мы решили для себя?

- Лучше сразу писать максимально эффективно!
- Мы не освобождаем память (узлы дерева и сопутствующая им информация)

Используемые нами подходы

std::string_view

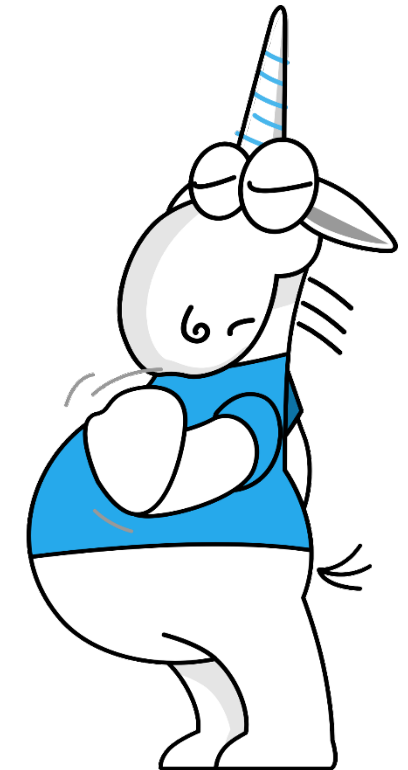
- Мы используем std::string
- Но во многих задачах настоящие строки (std::string) для нас ооочень медленные
- Мы начали использовать свой аналог std::string_view ещё 10 лет назад
- Сейчас везде перешли на std::string_view

Оптимизация размеров классов и структур

- Дерево разбора хранится всё время анализа
- Узлы дерева хранит разнообразную информацию:
  - тип (короткий, полный)
  - собранную информацию для поддерева
  - «виртуальные значения» для анализа потока данных
  - автоматически построенная информация для тел функций
- Это заставляет аккуратно относиться к формату хранения данных

Оптимизация размеров классов и структур

- Нет «дыркам» в структурах
- Кэш
- #pragma pack - не вариант



Оптимизация размеров классов и структур

```
typedef unsigned TypeInformation;

struct Argument
{
  bool used = false;
  const Ptree *tree = nullptr;
  const Ptree *interproceduralTree = nullptr;
  const Annotations::BasicAnnotation *annotation = nullptr;
  TypeInformation type;
};
```

40 байт

1	7	8	8	8	4	4

Оптимизация размеров классов и структур

```
typedef unsigned TypeInformation;

struct Argument
{
  const Ptree *tree = nullptr;
  const Ptree *interproceduralTree = nullptr;
  const Annotations::BasicAnnotation *annotation = nullptr;
  TypeInformation type;
  bool used = false;
};
```

32 байта

8	8	8	4	1	3

Оптимизация размеров классов и структур

- 4-байтовый enum - это часто расточительно
- Лучше явно подсказать

```
enum UnaryOperationName : uint8_t
{
  UN_OP_NOT, UN_OP_BIN_NOT, UN_OP_ADD,
  UN_OP_SUB, UN_OP_INC, UN_OP_DEC
};
```

Оптимизация размеров классов и структур

- Битовые поля не прижились
- Неудобно (появляются неестественные структуры данных)
- Медленный доступ (но это не точно)

Создаём переменные с максимально коротким временем жизни

- Ваш "капитан очевидность"
- Но это работает!

Создаём переменные с максимально коротким временем жизни

```
WiseType childWiseType;
WiseType baseWiseType;
if (!CreateWiseType(childArgType, childWiseType,
                    true, false, false, 0, false))
{
  return false;
}
if (!CreateWiseType(baseArgType, baseWiseType,
                    true, false, false, 0, false))
{
  return false;
}
```

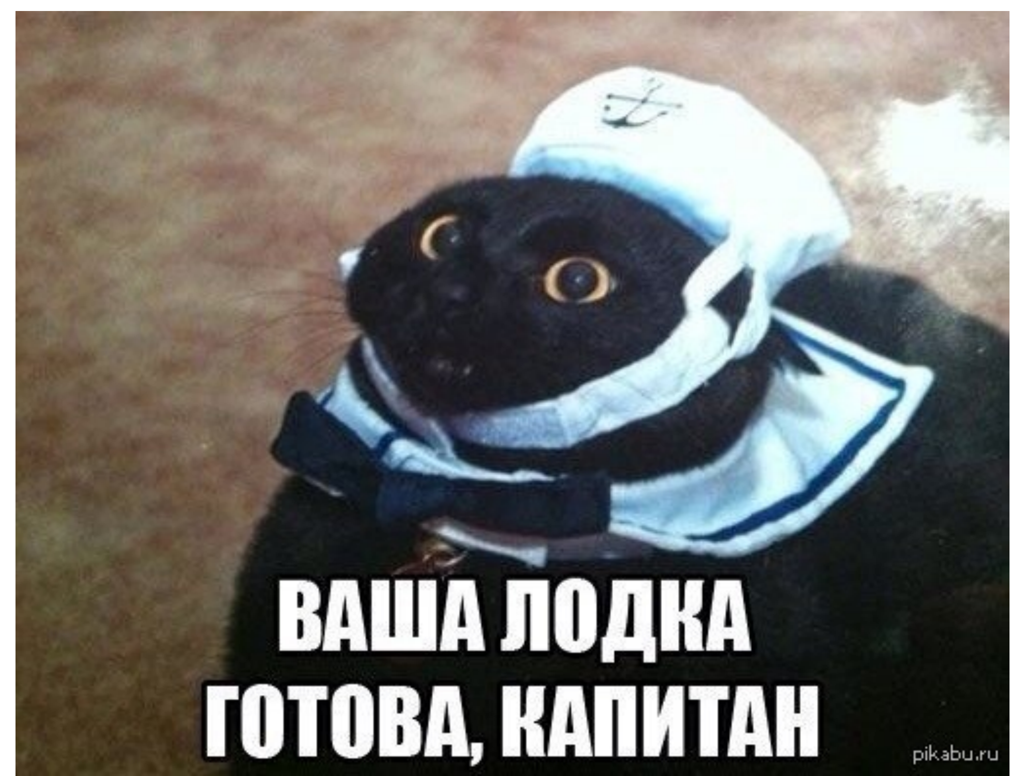
Создаём переменные с максимально коротким временем жизни

```
WiseType childWiseType;
if (!CreateWiseType(childArgType, childWiseType,
                    true, false, false, 0, false))
{
  return false;
}

WiseType baseWiseType;
if (!CreateWiseType(baseArgType, baseWiseType,
                    true, false, false, 0, false))
{
  return false;
}
```

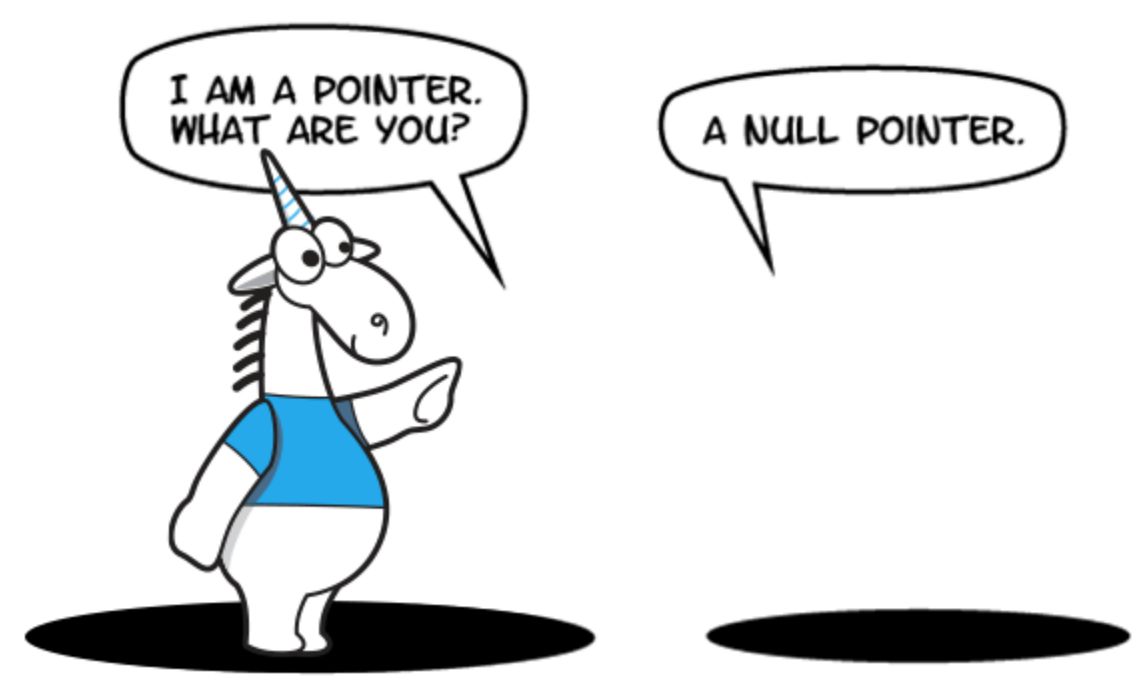
Ссылки вместо передачи классов по значению

- Очередное капитанство
- История клиентов про ускорение в приложения в 2 раза



Ссылки вместо указателей везде, где можно

```
static const AttributeInfo& FindAttributeInfo(const Ptree *p);
static const AttributeInfo& FindAttributeInfo(const Ptree &p);
```



Или небезопасные функции для проверенных указателей

- Да, это не красиво. Так исторически сложилось.
- Используем префикс в имени функции: `Unsafe`

Или небезопасные функции для проверенных указателей

```
bool Eq(const Ptree* p, const char c)
{
  return p != nullptr &&
         p->IsLeaf() &&
         p->GetLeafLength()==1 &&
         *(p->GetLeafPosition())==c;
}

bool UnsafeEq(const Ptree* p, const char c)
{
  VivaAssert(p && p->IsLeaf());
  return p->GetLeafLength()==1 && *(p->GetLeafPosition())==c;
}
```

Везде, где можно: константы и ссылки на константы

- Хотя, скорее, это относится к надёжности кода, а не скорости
- Защита от изменения не той переменной

```
const v_uint64 n = (bufSize - arg_3_value) / baseElementSize;

// Массив размерностью меньше 2 не интересен.
if (n < 2)
  return false;

for (size_t i = 0; i != n; ++i)
```

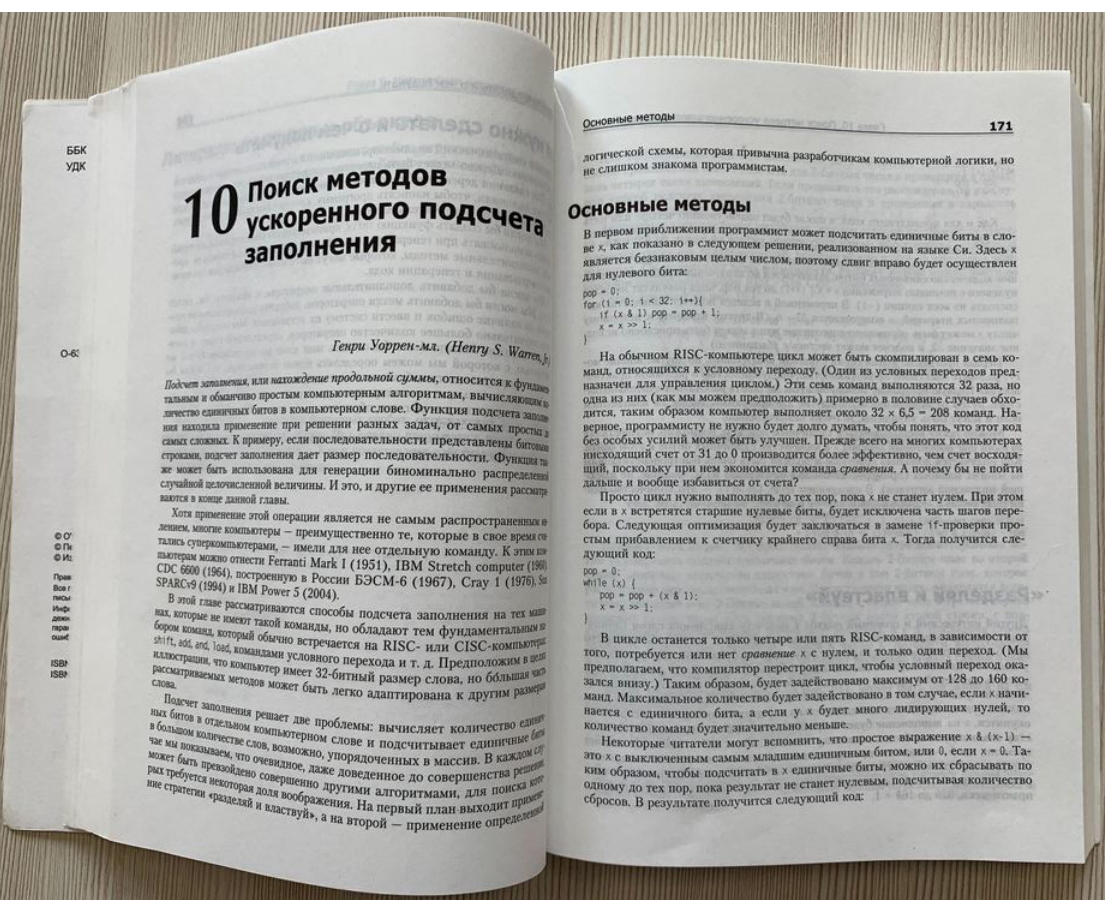
Не чураемся магии

- Количество единичных битов в 32-битной переменной
- Bit Twiddling Hacks: http://graphics.stanford.edu/~seander/bithacks.html

```
inline size_t CountingBitsSet(v_uint32 v)
{
  v = v - ((v >> 1) & 0x55555555);
  v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
  return static_cast<size_t>(
          (((v + (v >> 4)) & 0xF0F0F0F) * 0x1010101) >> 24);
}
```

Количество единичных битов в 32-битной переменной

- Кстати, это хорошее задание для собеседования



10 Поиск методов ускоренного подсчета заполнения

Генри Уоррен-мл. (Henry S. Warren, Jr.)

Основные методы — 171

логической схемы, которая привычна разработчикам компьютерной логики, но не слишком знакома программистам.

Основные методы

В первом приближении программист может подсчитать единичные биты в слове x, как показано в следующем решении, реализованном на языке Си. Здесь x является беззнаковым целым числом, поэтому сдвиг вправо будет осуществлен для нулевого бита:

```
pop = 0;
for (i = 0; i < 32; i++){
   if (x & 1) pop = pop + 1;
   x = x >> 1;
}
```

На обычном RISC-компьютере цикл может быть скомпилирован в семь команд, относящихся к условному переходу. (Один из условных переходов предназначен для управления циклом.) Эти семь команд выполняются 32 раза, но одна из них (как мы можем предположить) примерно в половине случаев обходится, таким образом компьютер выполняет около 32 × 6,5 = 208 команд. Наверное, программисту не нужно будет долго думать, чтобы понять, что этот код без особых усилий может быть улучшен. Прежде всего на многих компьютерах нисходящий счет от 31 до 0 производится более эффективно, чем счет восходящий, поскольку при нем экономится команда *сравнения*. А почему бы не пойти дальше и вообще избавиться от счета?

Просто цикл нужно выполнять до тех пор, пока x не станет нулем. При этом если в x встретятся старшие нулевые биты, будет исключена часть шагов перебора. Следующая оптимизация будет заключаться в замене if-проверки простым прибавлением к счетчику крайнего справа бита x. Тогда получится следующий код:

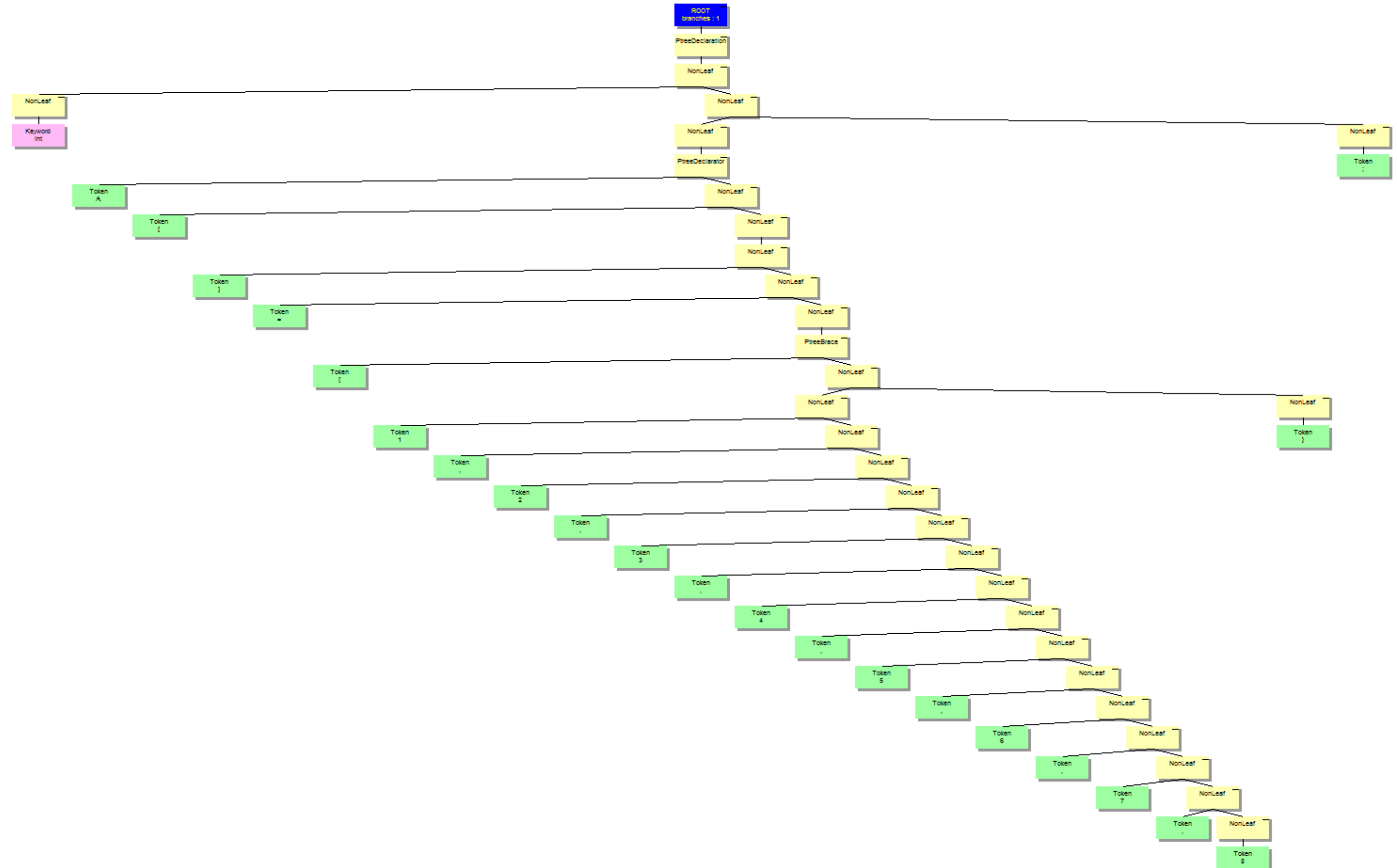
```
pop = 0;
while (x) {
   pop = pop + (x & 1);
   x = x >> 1;
}
```

В цикле останется только четыре или пять RISC-команд, в зависимости от того, потребуется или нет *сравнение* x с нулем, и только один переход. (Мы предполагаем, что компилятор перестроит цикл, чтобы условный переход оказался внизу.) Таким образом, будет задействовано максимум от 128 до 160 команд. Максимальное количество будет задействовано в том случае, если x начинается с единичного бита, а если у x будет много лидирующих нулей, то количество команд будет значительно меньше.

Некоторые читатели могут вспомнить, что простое выражение x & (x-1) — это x с выключенным самым младшим единичным битом, или 0, если x = 0. Таким образом, чтобы подсчитать в x единичные биты, можно их сбрасывать по одному до тех пор, пока результат не станет нулевым, подсчитывая количество сбросов. В результате получится следующий код:

Борьба с глубокими рекурсиями

- int A[] = { 1, 2, 3, 4, 5, 6, 7, 8 };



Борьба с глубокими рекурсиями

```
bool FindLeaf(const Ptree *tree, const char leaf)
{
  if (tree == nullptr)
    return false;
  if (tree->IsLeaf())
    return Eq(tree, leaf);
  return FindLeaf(tree->Car(), leaf) ||
         FindLeaf(tree->Cdr(), leaf);
}
```

Глубокая рекурсия

Борьба с глубокими рекурсиями

```
bool FindLeaf(const Ptree *tree, const char leaf)
{
  while (tree != nullptr)
  {
    if (tree->IsLeaf())
      return Eq(tree, leaf);
    if (FindLeaf(tree->Car(), leaf))
      return true;
    tree = tree->Cdr();
  }
  return false;
}
```

Сразу заботимся о сверхдлинных списках

```
// maxLen - после какого элемента остановить счёт
// return -1: p is a pair, but not a list.
// return -2: p is not a pair
// return -3: Слишком длинный список
ptrdiff_t QuickLength(const Ptree* p, ptrdiff_t maxLen);


// В больших массивах/enum незачем искать магические числа
// Если больше 20 элементов. 40 - так как есть запятые.
if (body != nullptr && QuickLength(body, 40) == -3)
  return;
```

Царство кэшей

- Пример задачи: нельзя просто сравнить имена типов
- Но честный вывод и сравнение типов - это долго

```
typedef int T;
{
  T a = 1;
  typedef long T;
  T b = 2;
  foo(a + b);
}
```

Складываются переменные одинаковых типов?

Царство кэшей (быстрый вывод типа)

```
bool TypeInfo::IsEq_IgnoreXXAndEnv(
  const TypeInfo &inf, bool bIgnoreRcvde) const
{
  ....
  TypePair newPair(m_env, m_encode, inf.m_env, inf.m_encode);
  bool isTypeEq;
  if (m_cmpTypeCache[bIgnoreRcvde].IsPairEq(newPair, isTypeEq))
    return isTypeEq;
  ....
  m_cmpTypeCache[bIgnoreRcvde].AddPair(newPair, result);
  return result;
}
```

Написание условий

- Важен порядок усложнения для максимально быстрого отсечения неинтересных вариантов
- Это так просто и очевидно, что про это легко забыть

- Беда в том, что от перестановки двух if ничего не изменится.
  Почти не изменится....
  Пока не появится 100 не оптимально расставленных if-ов

В начале делаем проверки, которые отсекают больше неинтересных случаев

```
if (pOp->GetLeafLength() == 2)
{
  if (UnsafeEq(pOp, "<=", 2) || UnsafeEq(pOp, ">=", 2) ||
      UnsafeEq(pOp, "!=", 2) || UnsafeEq(pOp, "==", 2))
  {
    ++n;
  }
}
```

Ещё важнее вначале сделать быстрые «бесплатные» проверки

- V599. Явно удаляем объект класса. При этом в классе нет виртуального деструктора, хотя есть виртуальные функции.
- Актуально только для C++
- Вначале быстрые проверки, потом медленные

Можно смело написать так

```
void ApplyRuleG_599(....)
{
  TypeInfo t;
  walker.Typeof(deleteExpr, t);
  t.Dereference();
  Class *pClass = nullptr;
  if (!t.IsClass(pClass))
    return;
  ....
  const bool containVirtualTablePtr = pClass->m_data.m_containVirtualTablePointer;
  const bool destructorPresent = pClass->m_data.m_containDestructor;
  ....
```

Продуктивнее добавить «лишнюю», но моментальную проверку

```
void ApplyRuleG_599(....)
{
  if (CompilerTypeIsAnyC())
    return;

  TypeInfo t;
  walker.Typeof(deleteExpr, t);
  t.Dereference();
  Class *pClass = nullptr;
  if (!t.IsClass(pClass))
    return;
  ....
  const bool containVirtualTablePtr = pClass->m_data.m_containVirtualTablePointer;
  const bool destructorPresent = pClass->m_data.m_containDestructor;
  ....
```

«Векторные» функции

```
bool Eq(const Ptree* p, char c)
{
  return p != nullptr &&
         p->IsLeaf() &&
         p->GetLeafLength()==1 &&
         *(p->GetLeafPosition())==c;
}

bool Eq(const Ptree* p, char c1, char c2)
{
  return p != nullptr &&
         p->IsLeaf() &&
         p->GetLeafLength()==1 &&
         (*(p->GetLeafPosition())==c1 || *(p->GetLeafPosition())==c2);
}
```

```
// Пример:
Eq(op, '<', '>')
```

«Векторные» функции

```
template<size_t N>
[[nodiscard]] inline bool Find(const Ptree *tree, std::array<ptrdiff_t, N> kinds)
{
  while (tree != 0)
  {
    auto kind = tree->What();
    if (std::find(kinds.begin(), kinds.end(), kind) != kinds.end())
      return true;
    if (tree->IsLeaf())
      return false;
    if (Find(tree->Car(), kinds))
      return true;
    tree = tree->Cdr();
  }
  return false;
}
```

«Векторные» функции в режиме write-only

```
template <typename PtreeKind, typename ...PtreeKinds, typename Enable>
[[nodiscard]] bool Ptree::UnsafeIsA(PtreeKind kind, PtreeKinds ...kinds) const noexcept
{
  return ((What() == kind) || ... || (What() == kinds));
}

template <typename PtreeKind, typename ...PtreeKinds,
          typename = std::void_t<decltype(std::declval<std::invoke_result_t<decltype(std::mem_fn(&Ptree::What)), Ptree&>>() == std::declval<PtreeKind>()),
            decltype(std::declval<std::invoke_result_t<decltype(std::mem_fn(&Ptree::What)), Ptree&>>() == std::declval<PtreeKinds>())...>>
[[nodiscard]] bool IsA(const Ptree &p, PtreeKind kind, PtreeKinds... kinds) noexcept
{
  return p.UnsafeIsA(kind, kinds...);
}

template <typename PtreeKind, typename ...PtreeKinds,
          typename = std::void_t<decltype(std::declval<std::invoke_result_t<decltype(std::mem_fn(&Ptree::What)), Ptree&>>() == std::declval<PtreeKind>()),
            decltype(std::declval<std::invoke_result_t<decltype(std::mem_fn(&Ptree::What)), Ptree&>>() == std::declval<PtreeKinds>())...>>
[[nodiscard]] bool IsA(const Ptree *p, PtreeKind kind, PtreeKinds... kinds) noexcept
{
  return p != nullptr ? p->UnsafeIsA(kind, kinds...) : false;
}
```

Жалоба

- Как всё легко испортить, поддерживая, например, стандарт MISRA

- Реализовано около 50 диагностик
- Если запустить на LLVM, то уже получается более 500 000 предупреждений

Свой менеджер памяти

- Кайф, что не требуется освобождать память
- Менеджер очень простой, быстрый и эффективный
  - 128 пулов для объектов размером от 1 до 128 байт
  - Каждый пул начинается от 1000 объектов, и затем выделяется новый в 2 раза больше предыдущего

```
const size_t MaxObjectSizeForQuickAlloc = 128;
const size_t ObjectsCountInBuf = 1000;

static void* Buffers[MaxObjectSizeForQuickAlloc + 1];
static size_t BuffersFullness[MaxObjectSizeForQuickAlloc + 1];
```

Нерешенные проблемы: повторная реализация аналогичного функционала по незнанию

- Количество C++ программистов увеличивается
- Появляются схожие вспомогательные функции
- В новых диагностиках делаются алгоритмы, которые уже есть в старых диагностиках

Важность тестов скорости

- Я говорил, что мерить бесполезно
- Это не совсем правда
- По возможности мы следим за скоростью тестов и ловим явные аномалии
- Помогает большой набор тестов
  - 120 проектов (Windows)
  - 40 проектов (Linux)
- Проверяется множество особых случаев, на которых алгоритмы могут медленно работать

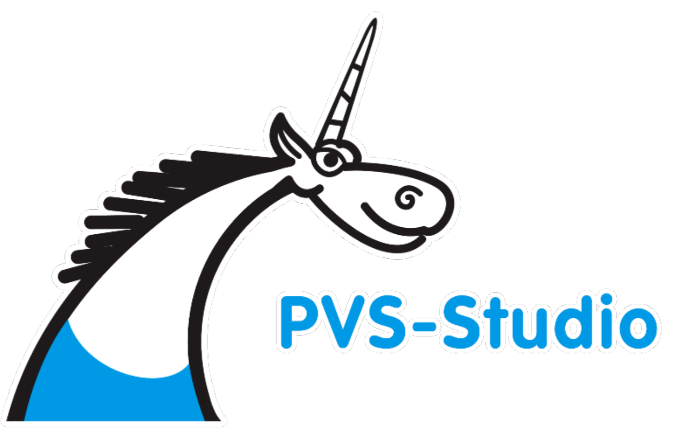
Каковы результаты войны за скорость?

- Посчитать сложно
- Количество тестов увеличилось в 3 раза
- Производительность компьютера выросла раз в 5
- Время анализа выросло с 40 минут до 90 минут
- Итого:
  - За 8 лет количество диагностик увеличилось в 4,5 раз
  - Производительность просела приблизительно в 4 раза

Заключение

- Полная фраза Дональда Кнута:
  We should forget about small efficiencies, say about **97%** of the time: premature optimization is the root of all evil.
- Мы попадаем с нашим проектом в оставшиеся **3%**
- **И преждевременная оптимизация нам помогает!**

Ответы на вопросы



- Андрей Карпов
- CTO, PVS-Studio, www.viva64.com
- E-Mail: karpov@viva64.com
- Twitter: @Code_Analysis
- LinkedIn: andrey-karpov-pvs-studio