Андрей Карпов

Лекция 8/12

Стандарт кодирования PVS-Studio и приёмы при разработке эффективных C++ диагностик

Докладчик

Карпов
Андрей Николаевич

- Сооснователь компании PVS-Studio, технический директор
- Разработал первые реализации ядра C++ анализатора
- Автор статей о качестве кода и статическом анализе

Зачем это нужно

- Это интересно - заглянуть к нам на "кухню"
- Некоторые практики можно применить и в других проектах

Стандарт кодирования

- Что это такое
- Цели:
  - Легко читать код коллег
  - Единообразие
  - Высвобождает время (не надо каждый раз думать как что-то оформить)

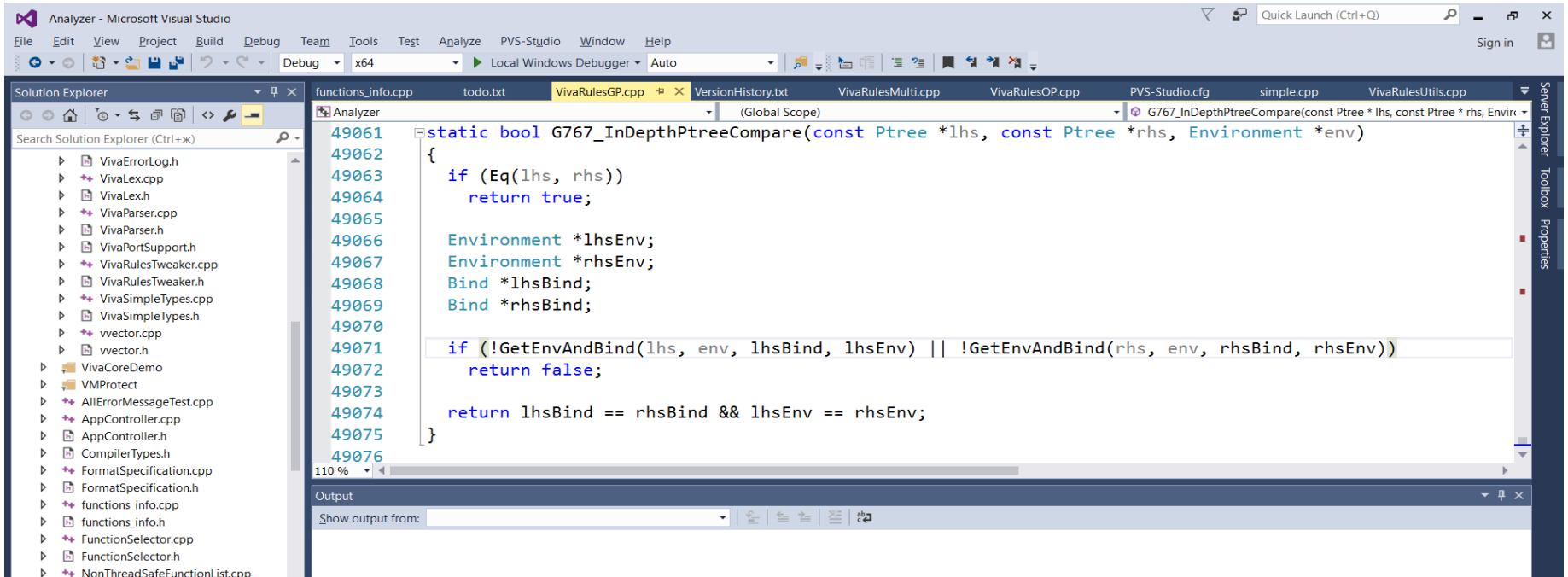
Стандарт: общие положения

- Недопустимо писать код в духе "Смотрите, как я могу!"
- Профессионализм разработчика состоит в том, что он пишет код, который может понять даже человек с меньшей квалификацией
- Правил без исключений не бывает

Наш стандарт кодирования на C++ (кратко)

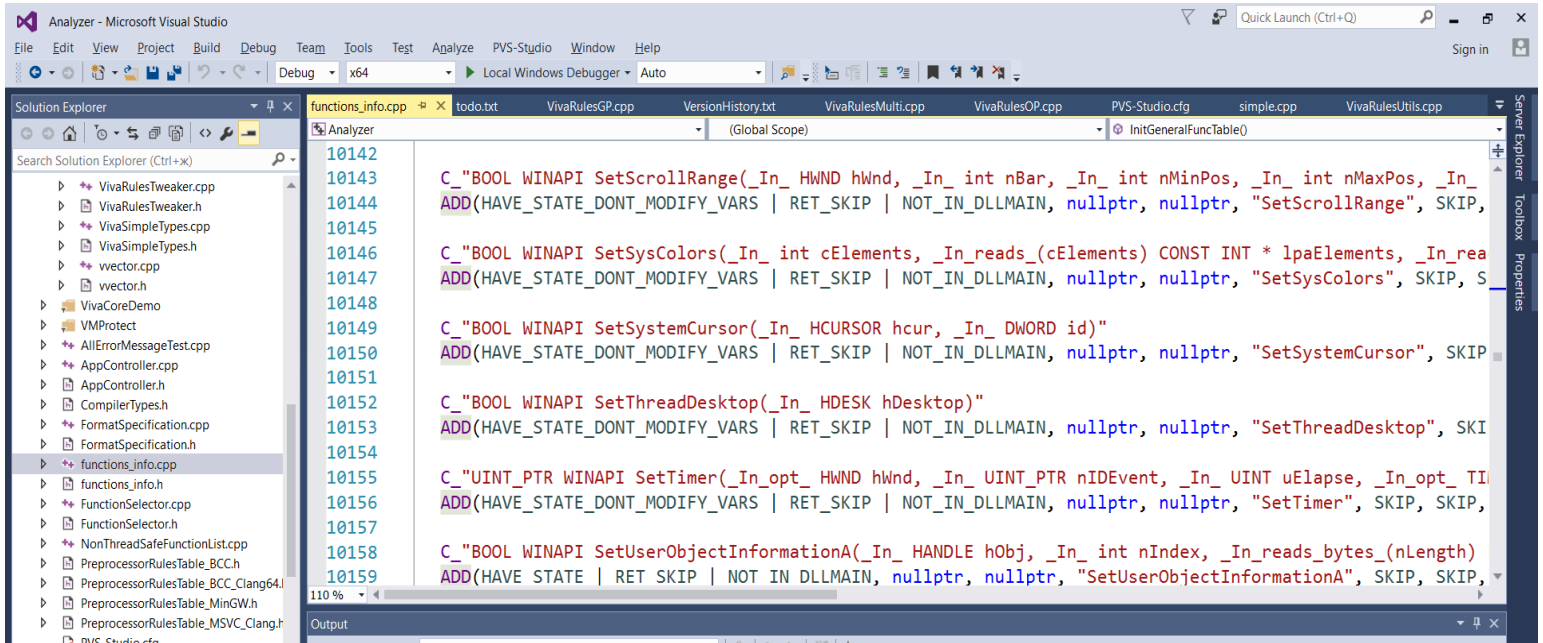
Стандарт: строки

- Для форматирования кода используются пробелы
- Стараться не делать строки длиннее 120 символов (раньше было 80)
- Пример нормальной по длине строк функции (самая длинная строка состоит из 93 символов):



Стандарт: строки

- Если началась большая вложенность блоков и код сильно сдвигается вправо, то подумайте над возможностью вынести что-то в отдельную функцию
- Исключения. Иногда делать строки короткими нет смысла. Пример - файл аннотирования функций functions_info.cpp



Стандарт: именование переменных

- Имена локальных переменных и формальных аргументов функций начинаются с маленькой буквы. Составные слова отделяются большой буквой.

  Примеры: variable, leftPtree, mySuperVar

- Массивы именуются с большой буквы: Array, StrTemplateNumberArgChars

- Члены классов начинаются с префикса m_, что означает что это "member". Далее следует имя с маленькой буквы. Составные слова отделяются большой буквой. Примеры: m_variable, m_leftPtree

Стандарт: именование переменных

- Глобальных переменных следует избегать. Но если такая переменная нужна, то её имя должно выделяться и указывать, что это что-то особенное. Пример: GlobalKeyValue
- Переменным следует давать осмысленные имена. Однако, в PVS-Studio требуется очень много вспомогательных переменных при работе с узлами дерева, и назвать их осмысленно бывает сложно. Поэтому часто можно встретить такие имена, как p, q, l, r, a, b

Стандарт: именование переменных

Оставляем подобные случаи на усмотрение программиста. Пояснение на примере:

```
bool Equal(const Ptree* p, const Ptree* q)
{
  while (p != q)
  {
    if (p == 0 || q == 0)
      return false;
    else if (p->IsLeaf() || q->IsLeaf())
      return Eq(p, q);

    if (!Equal(p->Car(), q->Car()))
      return false;
    p = p->Cdr();
    q = q->Cdr();
  }
  return true;
}
```

Стандарт: именование переменных

- Я сторонник использования префикса 'p' для обозначения указателей
- Однако, в PVS-Studio указателей так много, что этот префикс теряет всякий смысл и непрактично призывать его использовать
- Но иногда 'p' может помочь при именовании однотипных переменных, когда фантазия начинает подводить. Пример:

```
SimpleType type = pType->m_wiseType.m_simpleType;
```

Стандарт: именование типов

- Классы именуются с большой буквы. Слова отделяются в названии также большой буквой. Примеры: Ptree, NonLeaf, PtreeArray
- Если нужно создать синоним простого типа, то правила формирования имён такие же, как и для классов, например: using LineNumber = size_t;
- Имена перечислений начинаются с буквы E. В остальном правила именования совпадают с правилами именования классов. Пример: EGetArrayName

Стандарт: именование функций

Функции пишутся с заглавной буквы. Слова в названии также отделяются заглавной буквой. Примеры: `Length()`, `IsTruePointer()`, `GetNumericLimitsArg()`

Если в названии присутствует аббревиатура или особая смысловая единица, то её можно отделить подчеркиванием. Например, здесь подчеркивание отделяет два номера диагностик: `Reset506_507()`. Здесь подчеркивание подсказывает, что тело цикла берется у оператора while: `GetBody_While()`.

Стандарт: выравнивание кода

Отступ каждого очередного вложенного блока составляет 2 пробела

```
void Foo()
{
  if (a)
  {
    Foo1();
    Foo2();
  }
  Foo3();
}
```

Стандарт: выравнивание кода

Если строка слишком длинная, то переносим формальные и фактические аргументы функции на следующую строку, сдвигая их правее скобки

```
Type Fooooooooo(MyType argumentNumber1, MyType argumentNumber2,
                MyType argumentNumber3, MyType argumentNumber4,
                MyType argumentNumber5)
{
  if (Fooooooooo(argumentNumber1--, argumentNumber2--,
                 argumentNumber3++, 44, 55))
  {
    Foo(1, 2, 3);
```

Табличное оформление сложного условия

```
if (    !isLeaf
     && kind != ntUnaryExpr
     && kind != ntInfixExpr
     && kind != ntFuncallExpr
     && kind != ntDeclarationStatment)
{
  return GET_NONE;
}
```

Разделяющие пробелы

- После любой запятой всегда следует пробел:
```
Function(1, 2, x);
```
- Арифметические и логические операторы отделяются от операндов пробелами. Операторы присваивания тоже отделяются. Скобки пробелами не отделяются. Примеры:
```
x = 1 + z / (a + 1);
Foo(x / y, Do(a), Do(b ? 2 : 3));
```

Разделяющие пробелы

- Исключение составляют префиксные и постфиксные операторы:

```
x = *p;
x++;
p = &integer;
z = -z;
```

- Пробелы не ставятся рядом с квадратными и угловыми скобками:

```
Array[10] = X[i][j + q];
int a = Foo<long long>(10);
```

Разделяющие пробелы

- После операторов, таких как if, while, for, пробел всегда ставится:

```
if (!b)
for (size_t x = 0; x != 10; ++x)
if (a < b + c && x)
switch (W)
```

Разделяющие пробелы

- При описании указателей и ссылок, значки прижимаются к имени переменных, а не к типу:

```
const int **A;
float &f = z;
```

- Соображения: у нас две переменных - указатель и int, но смотрится так, как будто это 2 указателя:

```
int* p, n;
```

- Поэтому и следует прижимать звездочку к имени.

```
int *p, n;
```

Фигурные скобки

- Тела функций и таких операторов, как for, if, while и так далее, всегда оборачиваются в фигурные скобки, которые начинаются на следующей строке:

```
for (size_t i = 0; i != N; ++i)
{
  A[i] = q;
}
```

Комментарии

- Идеальный код не должен содержать комментарии. Все должно быть понятно из кода.
- В реальном мире это невозможно. Поэтому комментарии поясняют неочевидные моменты, тонкости.
- **Комментарий должен пояснять, что и зачем делается, а не то, как это делается.**

Неправильно (капитан очевидность)

```
// Заплатка!!
// Если рекурсивная вложенность достигла 10, то останавливаемся.
// Выдадим предупреждение (в Debug-версии) и выйдем из функции.
if (level > 10)
{
  VivaAssert(false);
  return false;
}
```

Нормальный комментарий

```
// Заплатка!! Бывает зацикливание на хитрых шаблонах.
// Здесь лечим последствия, а не причину.
// Экспериментально установил,
// что все наши тесты нормально проходят при магическом числе 7.
// На всякий случай сделал остановку при 10.
if (level > 10)
{
  VivaAssert(false);
  return false;
}
```

Комментарии

- Мы пишем комментарии на русском языке.
- После комментария мы ставим пробел и начинаем предложение с большой буквы. В конце ставим точку, если она уместна:

```
// Проверим, что тип объявлен в классе.
// A::A
```

- Большие (и только большие) комментарии можно написать с помощью `/* */`.
  - Таких комментариев стоит избегать, так как они мешают быстро закомментировать какой-то участок кода.

Разное

- Не делайте макрос там, где можно сделать обыкновенную функцию
- Используйте оператор `?:` только в простых выражениях
- Используйте для итераторов префиксный оператор инкремента (`++i`) вместо постфиксного (`i++`)
- Не используйте оператор `goto`
- Используем полную форму проверки (кроме bool)

```
if (!n)
if (n == 0)
```

Разное

- Используйте nullptr вместо NULL
- Там, где возможно, вместо enum используем enum class
- Все функции, которые используются только в одном модуле, должны быть объявлены как static
- Используйте using вместо typedef
- И так далее (всё в презентацию не уместить)

Главное

- Стандарт кодирования - это не догма, а помощник в написании хорошего кода

Преждевременная оптимизация

Преждевременная оптимизация

- 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!

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

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

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
};
```

Переменные с 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;
}
```

Переменные с 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;
}
```

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;
}
```

Константы и ссылки на константы

- Это больше относится к надёжности кода, а не к скорости
- Защита от изменения не той переменной

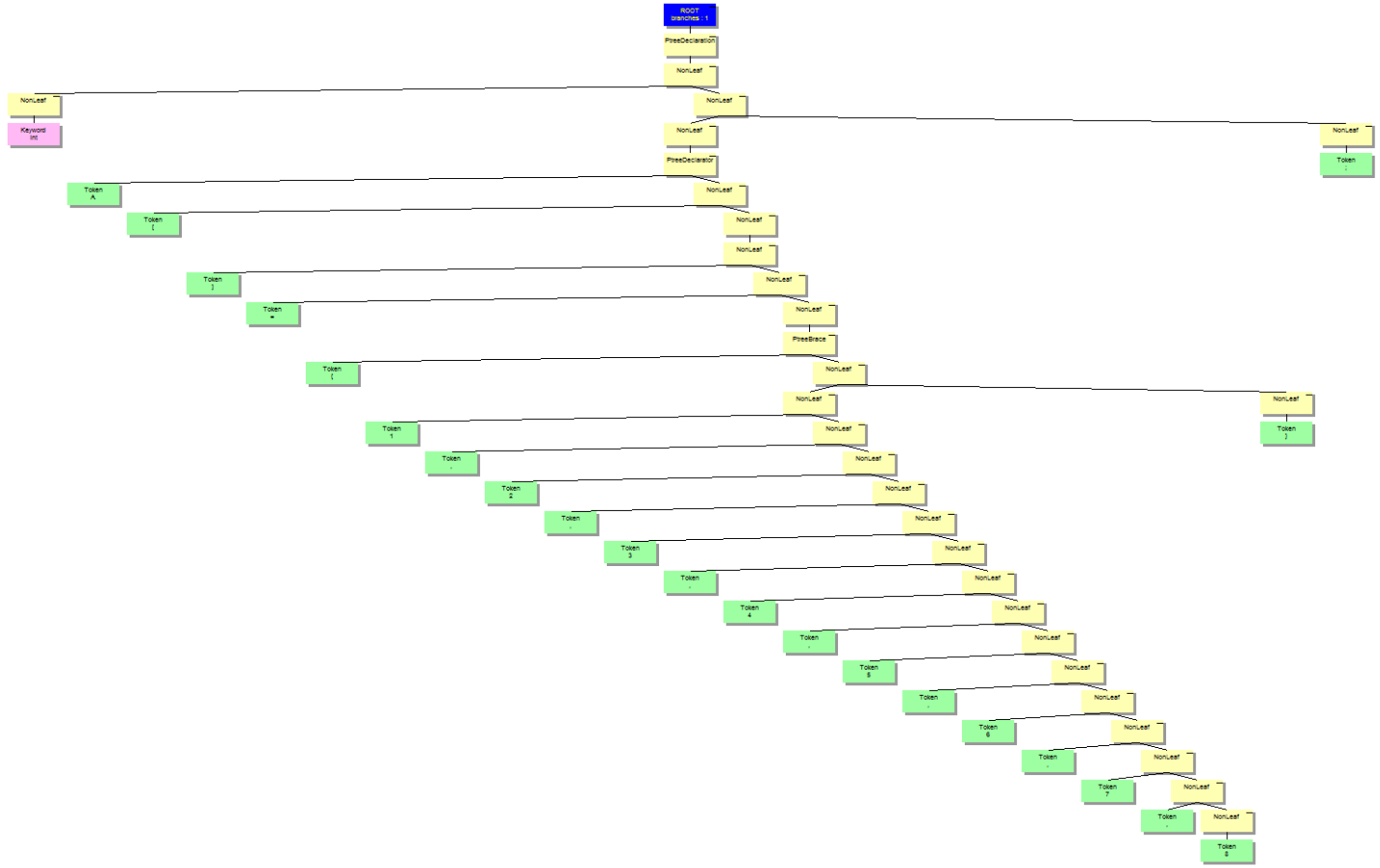
```
const v_uint64 n = (bufSize - arg_3_value) / baseElementSize;

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

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

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);
}
```

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;
}
```

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

Пример

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

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;
```

Как надо написать

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

  TypeInfo t;
  walker.Typeof(deleteExpr, t);
  t.Dereference();
  Class *pClass = nullptr;
  if (!t.IsClass(pClass))
    return;
  ....
```

«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 стиле 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;
}
```

Результаты войны за скорость?

- За 8 лет количество C++ диагностик увеличилось в 4,5 раз
- Производительность просела приблизительно в 4 раза

Выводы

- Стандарты кодирования - не догма, но полезный инструмент
- Важно писать код понятный для человека
- Иногда преждевременная оптимизация оправдана

Дополнительные ссылки

1. Главный вопрос программирования, рефакторинга и всего такого http://www.viva64.com/ru/b/0391/
2. Стив Макконнелл "Совершенный Код"

END

Q&A