# Правила кодирования на языке C++ (PVS-Studio, ООО "ПВС")

В рамках рассмотрения 8-го процесса РБПО по ГОСТ Р 56939-2024 решили выложить в качестве приложения наш стандарт написания C++ кода. Стандарт C++ команды у нас наиболее детализированный, поэтому лучше всего подходит для примера.

Стандарт выгружен из нашей базы знаний 19.08.2025. Это внутренний документ, поэтому перед публикацией из него удалены отдельные элементы: имена, некоторые названия, внутренние ссылки и другая информация непубличного характера. Данные фрагменты заменены на пометку (DEL).

Подробнее про разработку безопасного ПО и ГОСТ Р 56939-2024 вы можете узнать из:

- цикла публикаций в TG канале "Бестиарий программирования";
- цикла "Вокруг РБПО за 25 вебинаров: ГОСТ Р 56939-2024".

# Таблица контента

# Предисловие

Данный стандарт является обязательным к исполнению всеми сотрудниками. Каждый может вносить предложения по его усовершенствованию и изменению, но каждое предложение должно сопровождаться серьезным обоснованием. Ответственный за стандарт и внесение в него правок: (DEL). Любой сотрудник, в чьи должностные обязанности входит проведение code review, должен также следить за соблюдением данного стандарта в проверяемом коде.

# (DEL)

# Философия

Программисты своё основное рабочее время тратят не на написание, а на чтение кода. Если код будет красиво и единообразно оформлен, то увеличивается скорость его понимания. Причем единообразие не менее важно, чем красота. Нет ничего хуже, чем работать с кодом, который оформлен по-разному в разных местах.

Во-первых, его сложнее читать. Во-вторых, сложно подстраиваться под стиль предыдущего автора. Намного лучше, когда код единообразен. Даже если данный стандарт покажется чем-то странен и неудобен поначалу, то со временем к нему привыкаешь, и он начинает не мешать, а наоборот, помогать писать код.

Из важности чтения кода вытекает то, что 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.

Надо понимать, что правил без исключений не бывает. Если, нарушив стандарт, можно получить более изящный/понятный код, то его можно нарушить. Но это должно быть именно исключением, приносящим большой бонус в каждом конкретном случае.

# Прочитать дополнительно

Крайне рекомендуется ознакомиться с небольшой электронной книгой "Главный вопрос программирования, рефакторинга и всего такого". В ней затронут ряд вопросов, которые неуместно включать в стандарт кодирования, но материал будет полезен с точки зрения написания качественного кода.

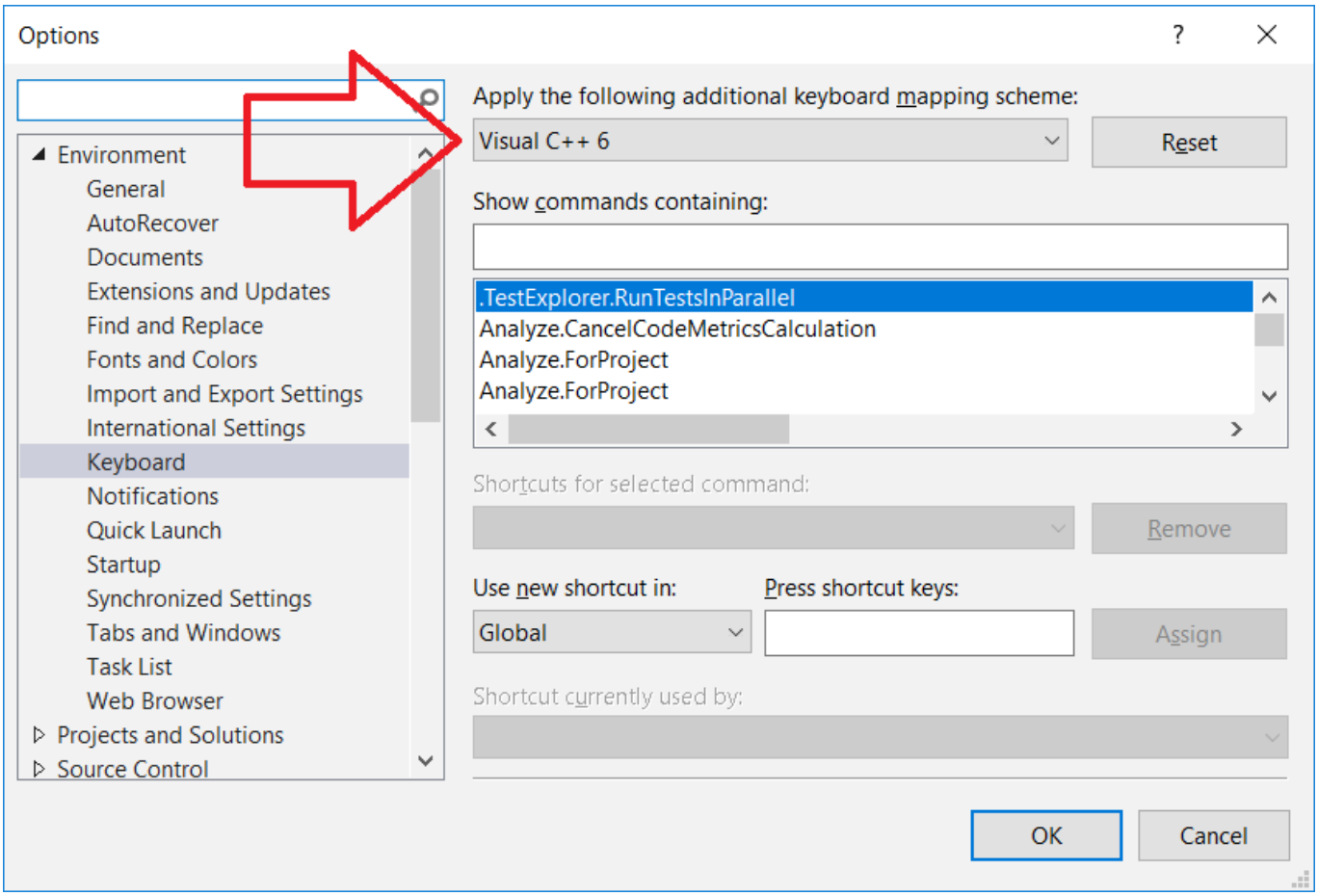
Желающим продвинуться ещё дальше предлагается прочитать фундаментальную книгу Стива Макконнелла "Совершенный Код".

# Горячие клавиши в IDE

Чтобы коллеги могли, сев за вашу клавиатуру, что-то сделать, горячие клавиши в разных IDE должны быть настроены одинаково.

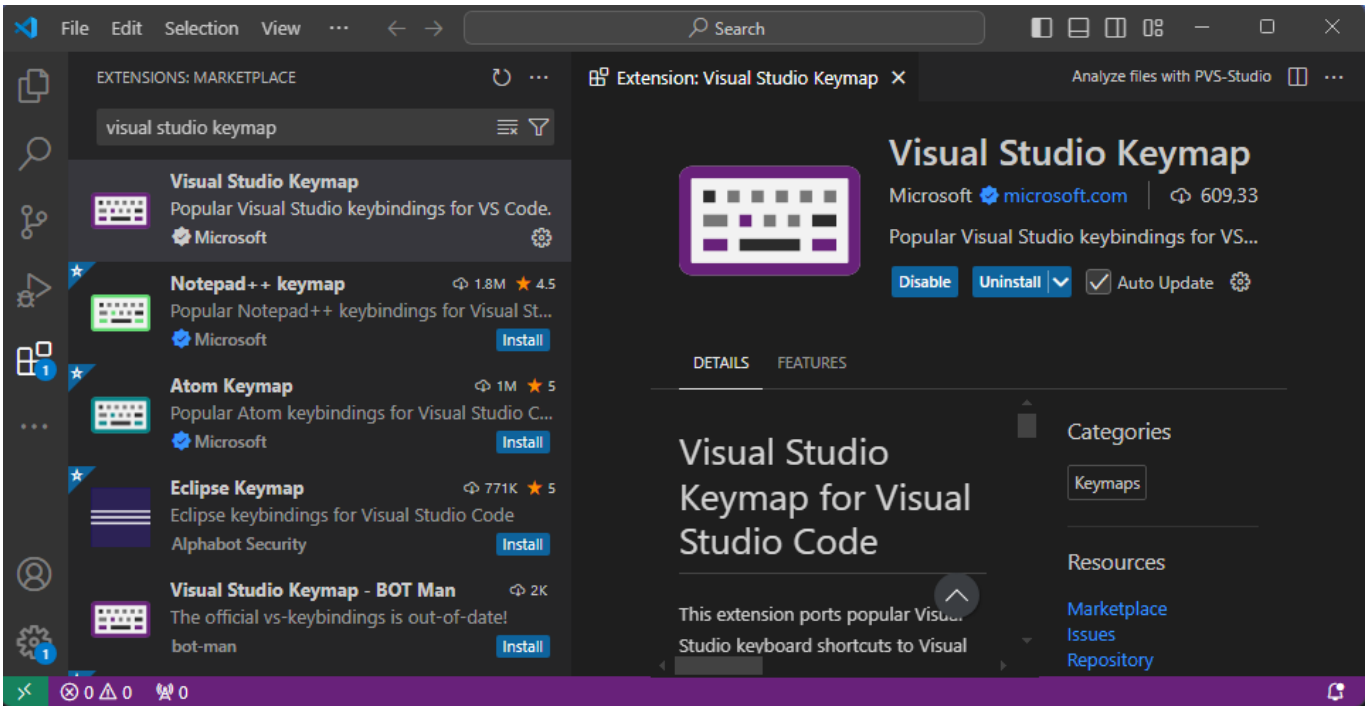
## Visual Studio

▶ Мы используем раскладку стандарта Visual C++ 6




## Visual Studio Code

▶ Установите плагин Visual Studio Keymap

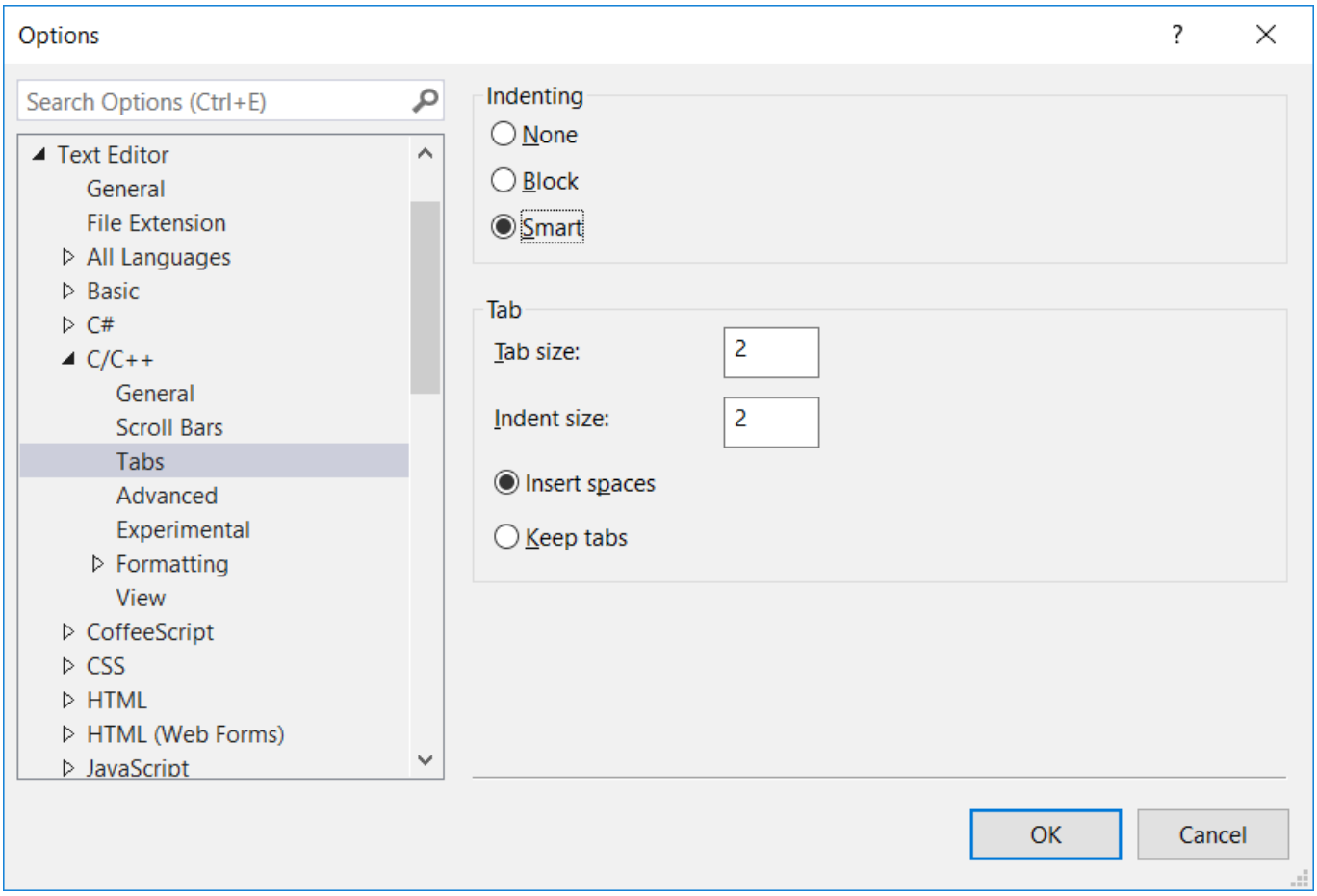



# Способы форматирования кода

Для форматирования кода используются пробелы. Табы недопустимы.

▶ Типовые настройки Visual Studio

Перейдите в `Tools > Options... > Text Editor > C/C++ > Tabs`. В `Indenting` должно быть выбрано значение `Smart`. В `Tab size` и `Indent size` выставлено значение 2.

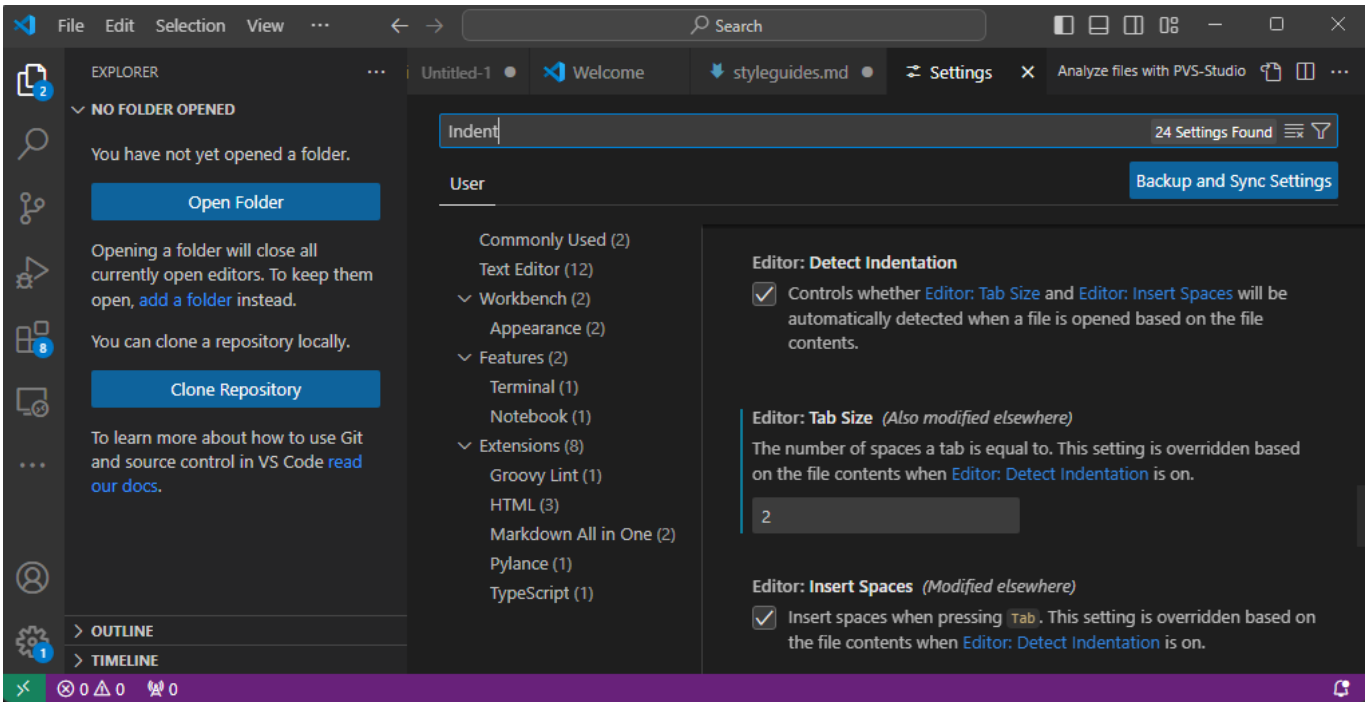



Тоже самое следует проделать и в настройках `Tools > Options... > Text Editor > Plain Text`.

Помимо C/C++ следует сделать аналогичную настройку и для txt-файлов.

▶ Типовые настройки Visual Studio Code

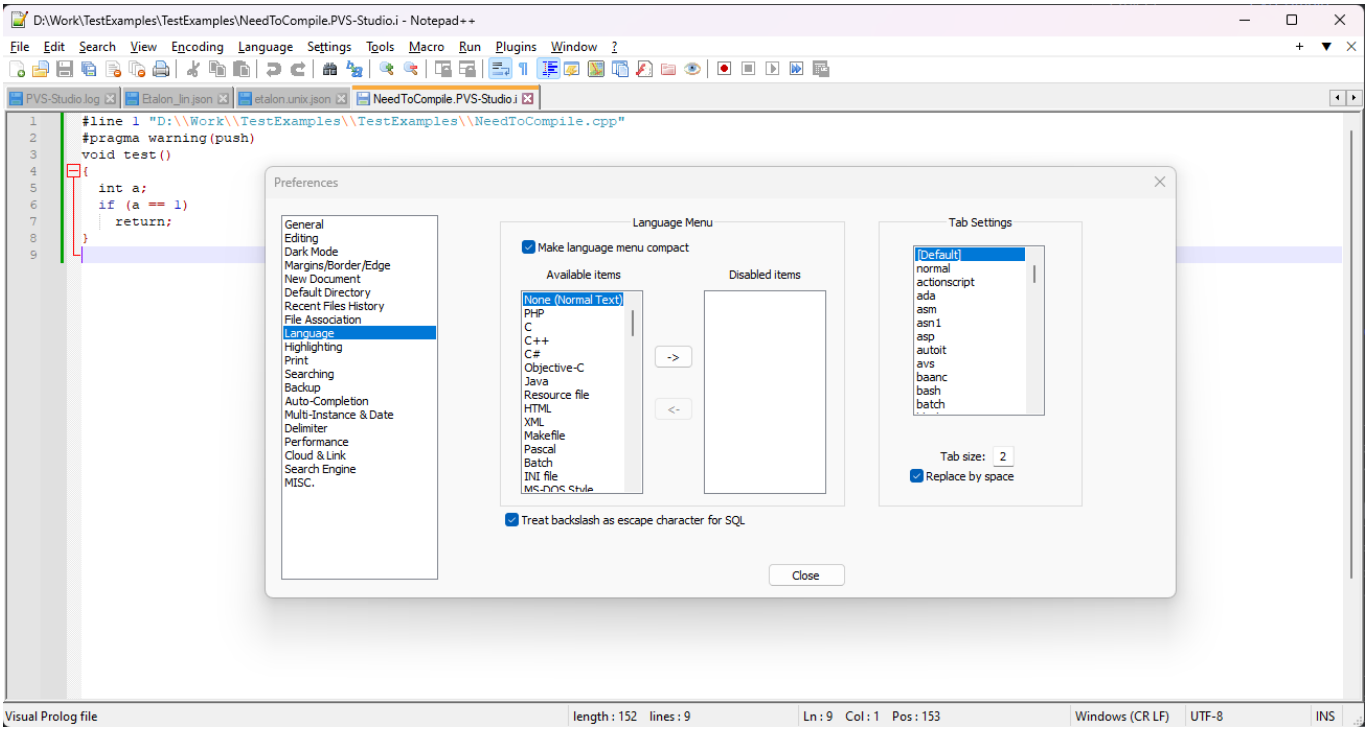
Перейдите в `File > Preferences > Settings...`. В поисковой строке введите `Indent` и нажмите слева `Text Editor`. Справа в поле `Tab size` выставьте значение 2 и установите галку `Insert Spaces`.



▶ Типовые настройки Notepad++

Перейдите в `Settings > Preferences... > Language > Tab Settings`

Установите активную ячейку в значение `[Default]`. Установите значение `Tab size` в 2 и поставьте галку `Replace by space`.

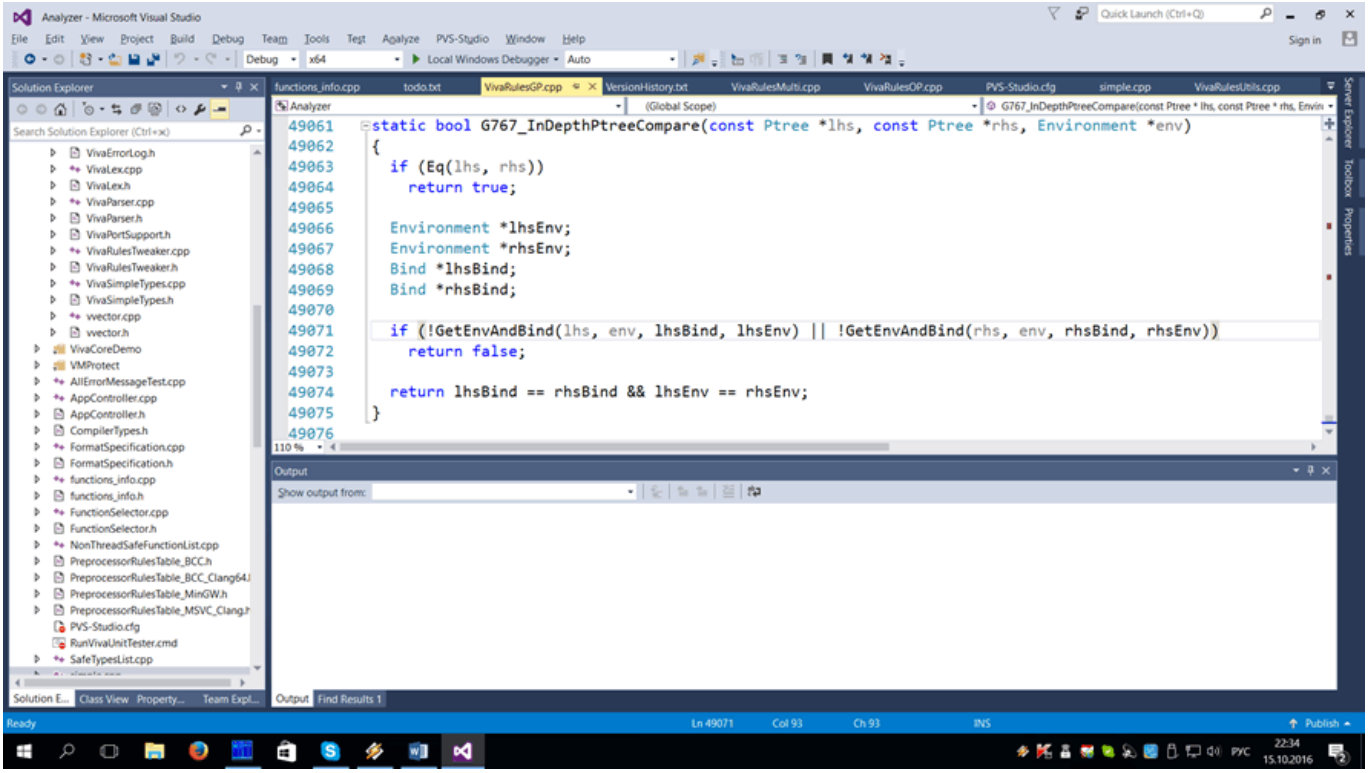



Благодаря использованию пробелов, код везде (в разных редакторах, с разными настройками) выглядит одинаково, и это настолько важно, что другие аргументы в пользу пробелов и не требуются.

# Длина строк

Нет четкого ограничения на длину строк, но надо стараться, чтобы они вмещались на экран. В стародавние времена рекомендовали 80 символов, но сейчас мониторы стали большие, поэтому разумнее придерживаться значения около 120 символов.

▶ Пример нормальной по длине строк функции (самая длинная строка состоит из 93 символов)




Да, то что можно написать до 120 символов в строку, не означает, что в строку надо пихать как можно больше. Чем короче строка, тем лучше. Например, вот такой код:

```
for (size_t i = 0; i != Array[k].GetFoo()->m_len; ++i)
```

лучше записать так:

```
const auto len = Array[k].GetFoo()->m_len;
for (size_t i = 0; i < len; ++i)
```

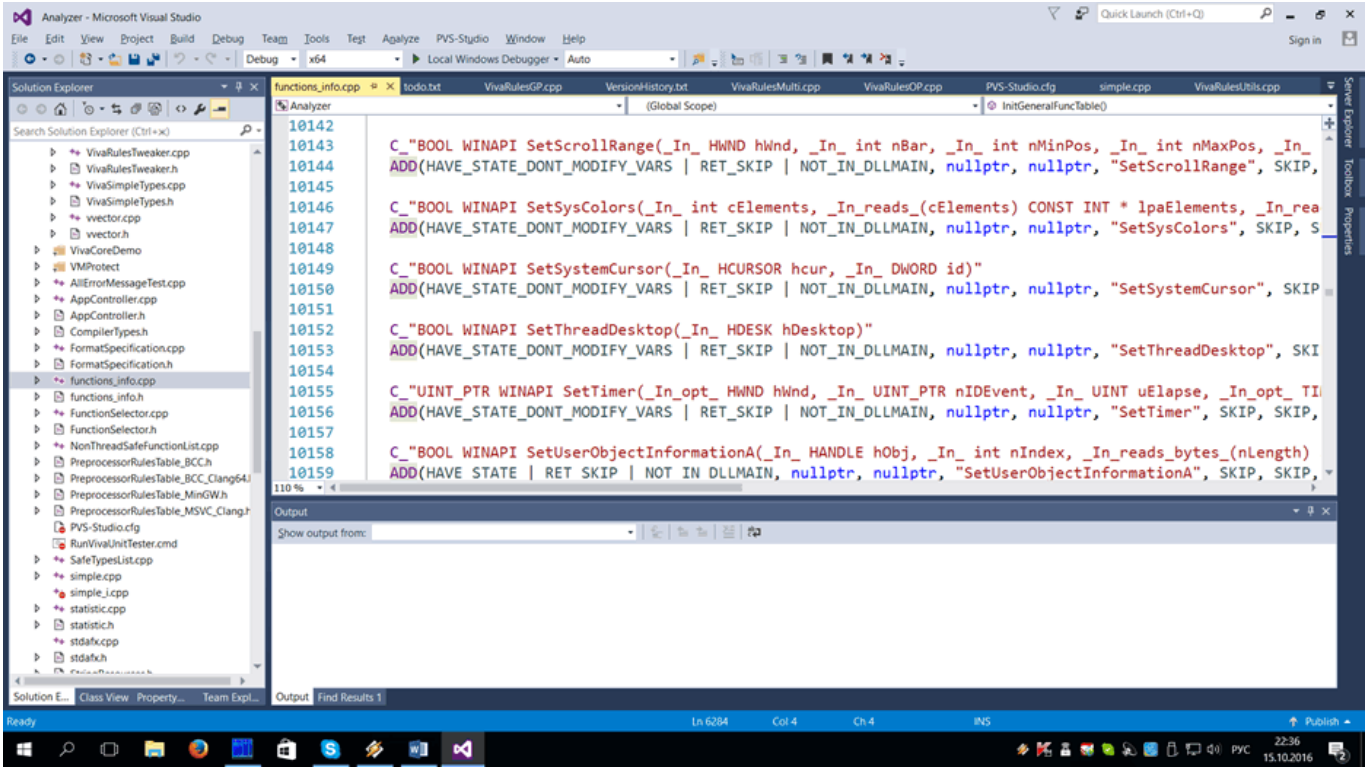
В таком коде не только строчки короче, но мы получаем ещё два преимущества. Во-первых, код проще отлаживать (можно легко посмотреть, сколько будет итераций). Во-вторых, компилятор может лучше оптимизировать код.

Совет. Если началась большая вложенность блоков и код сильно сдвигается вправо, то есть два способа решить проблему:

- Вынесите какой-то из блоков в отдельную функцию/лямбду.
- Примените технику раннего возврата из функции.

**Исключение**. Иногда делать строки короткими нет смысла. Пример - файл аннотирования функций `functions_info.cpp`. Этот файл не предназначен для частого чтения и редактирования, и если отредактировать весь его текст, чтобы всё вмещалось в ширину монитора, никому лучше не станет.

▶ Вот что имеется в виду




# Именование переменных

Имена локальных переменных и формальных параметров функций начинаются с маленькой буквы. Составные слова отделяются большой буквой. Примеры: `variable`, `leftPtree`, `mySuperVar`.

Приватные статические и нестатические данные-члены классов/структур начинаются с префикса `m_` (member, т.е. член класса). Далее следует имя с маленькой буквы. Составные слова отделяются большой буквой. Примеры: m_variable, m_leftPtree. Публичные данные-члены можно именовать согласно правилам локальных переменных и формальных параметров.

Глобальных переменных следует избегать. Но если такая переменная нужна, то её имя должно выделяться и указывать, что это что-то особенное. Пример: `GlobalKeyValue`.

Иногда при записи аббревиатур те сливаются с другим словами, и читать их сложно. В этом случае можно использовать для отделения слов в названии подчеркивание. Например, такое название смотрится плохо: `TRUEMacros`. Потому лучше написать: `TRUE_Macros`. Примечание: ещё лучше поменять слова местами и тогда можно отказаться от подчеркивания: `MacrosTRUE`.

Переменным следует давать осмысленные имена. Исключение - устоявшиеся имена в функциях, в которых семаника параметров/переменных понятны:

- `p`, `q` для узлов дерева;
- `l`, `r` для операндов бинарной операции;
- `min`, `max` для диапазонов значений;
- `i`, `j`, `k` для счётчиков циклов;
- `x`, `y`, `z`, `I`, `V`, .... для переменных в формулах

Рассмотрим пример исключения:

```
bool Equal(const Ptree *p, const Ptree *q)
{
  while (p != q)
  {
    if (EqualsAnyOf(nullptr, p, q))      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;
}
```

Эта функция для сравнения двух деревьев между собой. Станет ли лучше, если назвать указатели на деревья не `p` и `q`, а, скажем, `firstTree` и `secondTree`. Нет, код просто разбухнет, лучше не станет.

# Именование типов и функций

Классы, перечисления и псевдонимы типов (алиасы) именуются в стиле *UpperCamelCase* (*PascalCase*). Примеры: `Ptree`, `NonLeaf`, `PtreeArray`, `Length`, `IsTruePointer`, `GetNumericLimitsArg`.

Иногда при записи аббревиатур, те сливаются с другим словами, и читать их сложно. В этом случае можно использовать подчеркивание для отделения слов в названии. Например:

- Здесь подчеркивание отделяет два номера диагностик: `Reset_506_507`.
- Здесь подчеркивание подсказывает, что тело цикла берется у конструкции `while`: `GetBody_While`.
- Такое название типа смотрится непонятно: `GCQuickAlloc`. Потому лучше написать: `GC_QuickAlloc`.

Элементы перечислений пишутся в стиле *UpperCamelCase*: `Red`, `Green`, `Blue`, `PointerUnknownNonnull`.

**Примечание N1**. Исторически сложилось, что элементы перечисления `VivaCore::TokenNames` начинаются с `tk` для терминалов, а для нетерминалов с `nt`. Примеры: `tkIdentifier`, `ntForStatement`. Их так много, что менять эту традицию уже нет смысла.

**Примечание N2**. Для написания переносимого кода используйте типы из стандартной библиотеки: `std::uint8_t`, `std::int32_t`, ... - и пишите свои алиасы.

**Примечание N3**. Правила наименования функций в диагностических правилах см. ниже в разделе ("Оформления файлов с диагностическими правилами и их тестами)".

# Отступы и выравнивание кода

Отступ каждого очередного вложенного блока составляет 2 пробела. Пример:

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

Если строка слишком длинная, то переносим формальные и фактические аргументы функции на следующую строку, сдвигая их к другим аргументам. Пример:

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

Ещё один возможный вариант деклараций функции, если возвращаемое значение и имя функции занимают слишком места:

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    MyType argumentNumber1, MyType argumentNumber2,           // 4 пробела
    MyType argumentNumber3, MyType argumentNumber4,
    MyType argumentNumber5)
{
  if (Fooooooooo(argumentNumber1--, argumentNumber2--,        // 2 пробела
                 argumentNumber3++, 44, 55))
    Foo(1, 2, 3);
}
```

Когда имеем дело с длинными выражениями, то стараемся выравнивать табличным методом.

Данная тема подробно разобрана в документе "Главный вопрос программирования, рефакторинга и всего такого" (см. главу: Выравнивайте однотипный код "таблицей"). Поэтому здесь кратко.

Правильное и удобное табличное оформление сложного условия:

```
if (   !isLeaf
    && kind != ntUnaryExpr
    && kind != ntInfixExpr
    && kind != ntFuncallExpr
```

```
    && kind != ntDeclarationStatment)
{
  return GET_NONE;
}
```

Подобное оформление поначалу смотрится непривычно, но к нему быстро привыкаешь. Преимущества - все оформлено столбцами и легко заметить ошибку (например, дважды использовали одно и то же значение).

Другой вариант, чтобы не писать простыню сравнений, можно воспользоваться функциями `EqualsNoneOf` / `EqualAnyOf` / `EqualsAllOf`:

```
if (   !isLeaf
    && EqualsNoneOf(kind, ntUnaryExpr, ntInfixExpr, ntFuncallExpr,
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));
```

Исключение составляют префиксные и постфиксные операторы. Их отделять не надо:

```
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);
```

Между ключевым словом `if` / `while` / `for` и левой открывающей скобкой `(` всегда ставится пробел. Пример:

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

Запрещено в пределах одной декларации объявлять несколько деклараторов:

```
int foo(), bar(int i); // bad

int foo()
{
  int i, *p;           // bad
}

struct Foo
{

} a, b;                // bad
```

Исключение - безымянный пользовательский тип:

```
struct Foo
{
  struct
  {
    // your data members
  } a, b;                // ok
};
```

При декларации указателей и ссылок, разрешено прикреплять символы `*` или `&` или к типу, или к переменной. При этом вариант прикрепления к типу предпочтительнее. Примеры:

```
const int ** A; // bad
const int** A;  // good
const int **A;  // good

float & f = z; // bad
```

```
float& f = z;  // good
float &f = z;  // good
```

# Фигурные скобки

Тела функций и конструкций `for` / `if` / `while` / `switch` всегда оборачиваются в фигурные скобки, которые начинаются на следующей строке:

```
void Fill()
{
  for (size_t i = 0; i != N; ++i)
  {
    A[i] = q;
  }
  if (cond1)
  {
    // ....
  }
  else if (cond2)
  {
    // ....
  }
  else
  {
    // ....
  }
}
```

Скобки позволяют защититься от ошибок и делают код менее "плотным", что улучшает его чтение. Однако, как было сказано в самом начале, этот документ не догма, а способ писать единообразный красивый код.

**Исключение**. При написании ранних возвратов можно не переносить конструкцию на следующую строку с оборачиванием в фигурные скобки:

```
void foo(const Ptree *p)
{
  if (!p) return;
  if (!p->IsLeaf()) return;
  // ....
}
```

Согласитесь, код выглядит компактно, легко читается и не занимает много места. **ВАЖНО**: не пишите вызов макроса без фигурных скобок. Больше вероятности, что совершите ошибку, неверно реализовав макрос.

# Применение макросов

Надо всячески избегать использования макросов. Если вместо макроса можно написать обыкновенную функцию, то это нужно сделать.

**Совет**. В написании аналогов макросов могут помочь шаблоны функций/классов и `constexpr`-функции.

Единственное нормальное применение макросов - написание бойлерплейт-кода. Типичиным примером может быть использование макросов `MAKE_PTREE_TRAITS` и `MAKE_PTREE_TRAITS_FOR_BASE` в нашей кодовой базе.

# Применение шаблонов

В проекте привествуется использование шаблонов. Но надо помнить о главном правиле - не надо писать код так, чтобы показать коллегам "смотри, как я могу". Код шаблонов должен быть читабельным. В этом вам помогут *концепты* (*concepts*) из C++20. Не пишите `std::enable_if_t` и `std::void_t` при написании шаблонов, если у вас есть возможность не использовать их. Реализуйте свой концепт и используйте его.

# Формы проверок

Мы используем сокращенную форму проверки только для выражений, которые могут контекстно конвертироваться в тип `bool`. Во всех остальных случаях надо использовать полную форму сравнения с константами.

Итак, работа с `bool`:

```
bool GetBool();

void foo(bool b)
{
  if (b)               // ok
  if (!b && GetBool()) // ok
  if (b == false)      // not ok
}
```

Работа с *nullable*-типами:

```
void foo(int *ptr, const std::optional<float> &optF)
{
  if (!ptr && !optF)        // good
  if (ptr != nullptr)       // bad
  if (ptr)                  // good
  if (ptr == nullptr)       // bad
  if (!ptr)                 // good
```

```
  if (optF != std::nullopt) // bad
  if (optF)                 // good
  if (optF.has_value())     // good, but previous is preferrable
  if (optF == std::nullopt) // bad
  if (!optF)                // good
  if (!optF.has_value())    // good, but previous is preferrable
}
```

При работе с интегральными типами не полагайтесь на неявное преобразование к `bool`, сравните с нужным значением:

```
void foo(const char *str, size_t n)
{
  if (!*str)         // bad
  if (*str == '\0') // good
  if (*str)          // bad
  if (*str != '\0') // good
  if (!n)            // bad
  if (n == 0)        // good
  if (n != 0)        // good
}
```

# Использование тернарного оператора

Рассмотрим проблему оператора на примере кода с ошибкой:

```
bool ancestorsVisible = ....;
return (showLevel - (ancestorsVisible) ? 0 : 1) <= 0;
```

У тернарного оператора `?:` самый низкий приоритет. В нашей ситуации он ниже приоритета `-`. В результате программа ведёт себя не так, как хотел программист.

Используя тернарный оператор, очень легко ошибиться, а использовать его в сложных условиях - это вообще вредительство. Мало того, что можно не заметить ошибку, так ещё и читать такие выражения бывает очень сложно.

Давайте придерживаться следующих правил:

**Первое**. Не писать сложные выражения слева от `?`. Лучший вариант - в условии всегда должна быть переменная или вызов функции/лямбды:

```
const bool isWMemset = funcName == "wmemset";
fName = isWMemset ? G512_isWMemSet : G512_isMemSet;

constexpr auto isWMemset = [](std::string_view str) { return funcName ==
```

```
"wmemset"; };
fName = isWMemset(funcName) ? G512_isWMemSet : G512_isMemSet;
```

**Второе**. Не используем результат оператора `?:` в составе более сложного условия. Результат должен записываться в какую-то переменную или быть фактическим аргументом.

Плохой пример:

```
a = a ? b : c ? d : e;
ErrorType rule =
  (opName == BIN_OP_LOGIC_AND ? AndTable : OrTable)[lType][rType];
```

Хороший пример:

```
WarningLevel level = IsAnyCharType(literalType) ? Level_2 : Level_1;
G512_IsException25(walker,
                   isErrorInFirstArg ? *pArg1 : *pArg2,
                   isErrorInFirstArg ? firstBufSize : secondBufSize,
                   fName, arg_3))
```

# Глобальные переменные

Следует всячески избегать существования глобальных неконстантных объектов. Это, конечно, правило Капитана Очевидность, но всё равно лучше напомнить. В коде C++ проектов они присутствуют, но давайте не плодить их количество. Если вам очень потребовалась глобальная переменная, вам придётся доказать её значимость ревьюверу. При написании таких переменных помните про неуточнённый порядок их инициализации, так что оборачивайте статические переменные в функции и возвращайте такой объект по ссылке (синглтон Мейерса).

# Исключения для исключительных ситуаций

Мы не используем исключения для возврата каких-то состояний или прекращения каких-то алгоритмов.

Единственное нормальное применение исключений - это необходимость сигнализировать о фатальной беде и прекратить работу анализатора. Например, это нехватка памяти. В такой ситуации из оператора `new` бросится кастомное исключение типа `VivaBadAllocException`.

В остальных случаях уместнее воспользоваться `tl::expected` или `std::expected` (C++23):

```
enum class YourErrorType
{
  InvalidArgument
```

```
};

auto GetArgNumbesAsString(size_t argNum) -> std::expected<std::string,
YourErrorType>
{
  switch (argNum)
  {
    case 0:  VivaAssert(false); return
std::unexpected(YourErrorType::InvalidArgument);
    case 1:  return VivaErrors::first();
    case 2:  return VivaErrors::second();
    case 3:  return VivaErrors::third();
    case 4:  return VivaErrors::fourth();
    case 5:  return VivaErrors::fifth();
    case 6:  return VivaErrors::sixth();
    case 7:  return VivaErrors::seventh();
    case 8:  return VivaErrors::eighth();
    case 9:  return VivaErrors::ninth();
    case 10: return VivaErrors::tenth();
    default: return AsShortOrdinal(argNum);
  }
}
```

# Оптимизации

Работая над проектом PVS-Studio, вспомните знаменитую фразу:

> Преждевременная оптимизация — корень всех зол.

В большинстве случаев тормозит неэффективный алгоритм. Начните с него. Если ваш алгоритм делает что-то квадратично/экспоненциально, а это можно было сделать за линейное время - знайте, вам туда.

В качестве примера можно привести нашу вечную проблему вызова семейства функций `UpTo*` в диагностических правилах. Это абсолютное зло. Благодаря таким вызовам, а также природе рекурсивного обхода синтаксического дерева вы можете получить алгоритм экспоненциальной сложности, даже не задумываясь об этом. Доказательством этого являются куча задач в Issue Tracker о том, что у нас что-то тормозит. В большинстве случае оказывается "`UpTo` в космос".

Общие советы:

- Если вам надо понять, что вы ранее посетили какую-то ноду, сохраните состояние об этом при обходе дерева и сбросьте, когда покинете нужную ноду.
- При написании диагностических правил старайтесь "легковесные" проверки помещать вверх (например, ноды деревьев на `nullptr`; строкове содержимое терминалов), а "тяжеловесные" - вниз (например, поиск строки исходного файла на основе ноды дерева; определение, что нода дерева внутри макроса).
- Количество диагностических правил будет всё расти и расти. Поэтому надо стараться писать их достаточно эффективными. Например, чтобы ранние проверки могли отсечь 90% ненужных для правила кейсов. Иначе "тормоза" будут равномерно распределены по сотням диагностик.

> Допустим, вы написали не очень эффективную диагностику X. Прирост времени работы анализатора 0.5%. Вроде ничего страшного, ведь диагностика полезная. Потом ваш коллега написал не очень эффективную диагностику Y, и анализатор замедлился, скажем на 1%. И опять особенно не заметно. И вот, по прошествии 50 новых правил вдруг выясняется, что анализатор стал работать в полтора раза медленнее. И сделать что-то с этим будет сложно.

# 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. Пример:

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

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

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

Неправильный комментарий "капитана 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;
}
```

Комментарий выше ничего не объясняет. Почему останавливаемся? Что за число 10? Правильный 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;
}
```

При оформлении *TODO* в коде пользуйтесь следующим паттерном:

```
// TODO: [xxxx] описание
```

где `[xxxx]` - служебный тег для быстрого поиска в будущем проблемы. Например, тег `c++23` будет намекать, что при обновлении компиляторов и стандартных библиотек можно будет переписать код более эффективно с применением нового стандарта.

# Оформления файлов с диагностическими правилами и их тестами

При работе с диагностическими правилами общий принцип - не делать всё руками, а использовать скрипт `add_rule.py`:

```
python3 add_rule.py $NUMBER_OF_DIAGNOSTIC_RULE
```

Он формирует следующий набор файлов:

- `Vxxxx.cpp` - диагностическое правило
- `Vxxxx_doc.cpp` - тест документации
- `VxxxxArtifacts.cpp` - набор функциональных тестов для диагностического правила

и вносит модификации в систему контроля версий.

Рассмотрим подробнее эти файлы и что вам с ними делать

## Vxxxx.cpp

Это владения диагностического правила, именно здесь вы и будете реализовывать всю логику. Нас интересует сгенерированное содержимое файла:

```
(DEL)

/*
Vxxxx:

ОПИСАНИЕ

Пример:

Исключения:
*/
void ApplyRuleG_xxxx(VivaCore::VivaWalker &walker)
```

```
{
}
```

Остановимся поподробнее на каждой сущности в этих комментариях.

### Vxxxx

Здесь указывается номер диагностического правила, а также некоторые специальные строчки. Они будут парситься генератором документации, и из них будут автоматически сформированы некоторые таблички. Полный список описан здесь: (DEL).

## ОПИСАНИЕ

Общее правило - оформляйте описание так, чтобы другой разработчик мог понять из него, что правило делает. Считайте других разработчиков склонными к насилию психопатами, который знают, где вы живете (C) Джон Ф. Вудс. Ну или в крайнем случае лид им подскажет 😃

▶ Плохой пример N1

```
/*
Vxxxx: ....

Есть проблема, которую (DEL) хотят отлавливать.

char *array = (char *)malloc(10); // the only change
....
memset (&array, 0, sizeof(array)); // зануляем указатель вместо массива  //
зануление указателя, а не области памяти выделенной массиву.

*/
```

Что здесь плохого:

- Строка "Есть проблема, которую (DEL) хотят отлавливать" ничего не говорит читателю. Разве что правило делалось для клиента. Но даже от клиентов мы запрашиваем конкретное описание того, что она должна делать. Даже если и хочется её добавить, то ей место в конце.
- Суть диагностического правила спрятана в комментарии к коду. В добавок ещё и криво отформатированный.

▶ Исправленный пример N1

```
Потенциальная утечка памяти. Указатель на аллоцированный ресурс зануляют с помощью
`memset` вместо зануления самого ресурса.

Рассмотрим пример:

// char array[10];
char *array = (char *)malloc(10);
```

```
....
memset (&array, 0, sizeof(array));

Раннее разработчик использовал массив на стеке, и код функционировал корректно.
Однако потом он решил экономить стек и увёл создание массива в кучу.
При этом вызов `memset` он не изменил. В таком случае теперь зануляется указатель
на динамически аллоцированный буфер вместо зануления массива. В худшем случае
получаем утечку памяти, если указатель никуда не сохранили.

Правило появилось в результате общения с (DEL) (клиент).
```

Часто правила реализуются через другие. Давайте рассмотрим пример комментария, как не стоит оформлять описание.

▶ Плохой пример N2

```
X-диагностика для ГОСТ Р 71207-2024. Ищет использование индекса после проверки в
том же скоупе. Аналог V1004, только для индексов.
```

Что плохого в этом описании:

- Ребус на ровном месте. Вообще непонятно, что делает правило. Таким образом вы заставляете читателя кода переходить в другой файл, вникать в суть, а потом ещё сопоставить её с текущим диагностическим правилом.
- Нет примера, что же она ищет.

▶ Исправленный пример N2

```
X-диагностика для ГОСТ Р 71207-2024. Ищет потенциальный выход за границу массива в
следующей ситуации:
- Выражение, используемое в качестве индекса, сначала проверялось
- В той же области видимости ниже по коду происходит доступ к массиву с помощью
того же выражения. Только теперь без проверки.

Пример:

#define SIZE 10
int buf[SIZE];

int do_something(int);

int some_bad_function(int idx)
{
  int res;

  if (idx < SIZE)
  {
    res = do_something(buf[idx]);
  }
```

```
  // ....
  res = do_something(buf[idx]);   // <=
  return res;
}

Правило схоже с V1004 (разыменование указателя без проверки, хотя ранее оно
производилось под проверкой), только для индексов массивов.
```

И последний совет - старайтесь не показывать сложные примеры в описании (или из реальных проектов). Это лишь усложняет понимание сути.

▶ Плохой пример N3

```
void Metazone_Get_Flag(u16* pFlag, int index)
{
  u8* temp = 0;
  u8 flag = 0;
  if (index < 8)
  {
    temp = (u8*)pFlag + 1;
    flag = (*temp >> index) & 0x01;
  }
  else if (index < 32)
  {
    temp = (u8*)pFlag;
    flag = (*temp >> index) & 0x01;  // <=
  }
}
```

Надо прочитать простыню кода до заветной строчки, хотя здесь можно упростить проверки. Суть правила в том, правый операнд сдвига больше или равен числу бит, которое было в типе до *integral promotion* (это не UB).

▶ Исправленный пример N3

```
bool foo(u8* pFlag, int index)
{
  if (index < 8 || index >= 32) return false;

  return (*pFlag >> index) & 0x01;        // <=
}
```

## Исключения

Любое реализованное в лоб диагностическое правило даёт вагон и маленькую тележку ложноположительных срабатываний. Наша философия - генерировать их как можно меньше. Поэтому

мы реализуем в диагностических правилах исключения, когда диагностическое правило не должно отрабатывать.

Смысл в том, что **крайне рекомендуется** прописывать их в разделе "Исключения" в комментарии к диагностическому правилу.

▶ Пример исключений

```
(DEL)
```

После того, как исключения прописаны, надо также указать точку, где оно реализовано в правиле. Обычно мы делаем это при помощи комментария следующего вида:

```
// Исключение Nx
```

где `x` - номер вашего исключения из описания к правилу.

**Примечание**. Мы не пишем символ `№` для указания порядкового номера. Мы используем именно символ `N`. Давайте стараться делать это стандартизированно.

### ApplyRule

Это точка входа в диагностическое правило. К сожалению, на текущий момент нет общего интерфейса для таких точек входа. Но давайте оговорим основные принципы:

- Функция должна содержать номер диагностического правила. Если правило реализует сразу несколько правил, то разделяем их символом подчёркивания (например, `ApplyRule_501_2501`).
- Функция принимает `VivaWalker` для регистрации предупреждения.
- Если функция принимает ноду дерева, то она должна принимать наиболее специфичную ноду. Параметр `const Ptree *` - это плохой паттерн. Да, старые правила написаны именно так, но мы теперь хотим делать лучше.
- Наиболее частым местом внедрения являются функции `VivaWalker::Translate*`. Перед вызовом `ApplyRule` надо проверить, что правило активировано. Для этого позовите функцию-член `VivaWalker::IsErrorActive`.

## `VxxxxArtifacts.cpp`

Этот файл содержит набор набор функциональных тестов для диагностического правила. Чем больше паттернов вы сможете написать, тем лучше. Нужно писать как и те примеры, где мы должны срабатывать (позитивные), так и те, где мы не должны (негативные). Мы не разделяем позитивные и негативные тесты по разным файлам, но никто не будет против, если вы так сделаете.

Для пометки, что в конкретной строке должно быть срабатывание диагностического правила N, используйте комментарий `//+V`:

```
```

```
void foo()
{
  1 / 0;   //+V609
}
```

Если вы хотите также проверить, какого уровня выдаваемое предупреждение, используйте комментарий `//+Vxxx:N`:

```
void foo()
{
  1 / 0;   //+V609:1
}
```

**Примечание**. Код в файле должен проходить компиляцию. Не пишите некомпилируемые примеры!!!

### Vxxxx_doc.cpp

Этот файл содержит набор примеров, которые мы указали в документации к диагностическому правилу. Суть - мы хотим ловить регрессию, если анализатор перестал выдавать предупреждение на позитивном примере из документации. И наоборот - стал выдавать предупреждение на негативном примере. Поэтому как только написана документация, выпишите все примеры - и невалидные, и валидные, в этот файле.

Правила разметки строк с предупреждениями тот же, что и в функциональных тестах диагностического правила.

**Примечание**. Код в файле должен проходить компиляцию. Не пишите некомпилируемые примеры!!!

## Разное

1. Используйте для итераторов префиксный оператор инкремента (`++i`) вместо постфиксного (`i++`).
2. Не используйте конструкцию `goto`.
3. Используйте `nullptr` вместо макроса `NULL`.
4. Там, где возможно, вместо `enum` используем `enum class`.
5. Исключения не должны покидать деструктор. Если в деструкторе делается что-то, что может сгенерировать исключение, то следует подавить такие предупреждения:

```
try
{
  ....
}
catch (...)
{
  VivaAssert(false);
}
```

Есть один сценарий, когда надо учесть, что деструктор может бросить исключение - это *scope_fail*-guard. Но таких ситуаций - по пальцам пересчитать. В общем случае можно считать, что писать `noexcept(false)` для деструктора - это антипаттерн, и исключения не должны вылетать за его пределы.

1. Все функции, которые используются только в одном модуле, должны быть объявлены как `static`. Будет проще линкеру, а на LTO (*Link Time Optimizations*) можно больше оптимизаций сделать.
2. Для подавления предупреждений о неиспользуемых параметрах используется функция `PVS_UNREFERENCED_PARAMETER` или атрибут `[[maybe_unused]]`.
3. Для подавления предупреждений о неиспользуемых переменных используется функция `PVS_UNREFERENCED_VAR` или атрибут `[[maybe_unused]]`.
4. Не реализуем в классе в заголовочном файле `inline`-функции. Причину можно изучить здесь. Реализацию функций делаем в `*.cpp` файлах.
5. Используйте `using` вместо `typedef`. Более того, псевдоним шаблона через `typedef` реализовать нельзя.
6. Используйте `std::string_view` вместо `const std::string &` или `(const char *, size_t)`.
7. **НЕ ПЕРЕДАВАЙТЕ** в C-функции для манипуляции со строками (например, `strlen`) выражение `std::basic_string_view<T>::data`, если вы также не можете передать размер. Причина - в `std::basic_string_view<T>` лежит 2 указателя (начало и конец буфера), и буфер может быть даже не нуль-терминированным. Но даже если нуль-терминированный, то функции начнут оперировать данными, которые туда не должны были прийти. Также помните, что некоторые классы делают это неявно - например, конструктор `std::basic_string<T>::basic_string(const char *)`.
8. DRY (don't repeat yourself). Не надо писать `foo("abc", std::size("abc") - 1)` даже ради производительности. Современные оптимизаторы умнее вас.
9. Избегайте паттерна функций, которая возвращает bool в случае успеха и модифицирует объект через мутабельную ссылку/указатель. Возвращайте `std::optional<T>` или cv-квалифицированную версию `T *`.
10. Используйте `std::variant` вместо `union`.
11. Используйте `auto` там, где это уместно.
12. Не используйте `auto` там, где это неуместно.

# Заключение

Описать все возможные сценарии и правила невозможно, поэтому, если непонятно, как лучше оформить ту или иную конструкцию, то можно либо посмотреть, как подобные конструкции записаны в других частях программы, либо обратиться за советом к тимлиду или техлидам. Они подскажут, как поступить лучше, а тимлид заодно поправит стандарт, чтобы исключить такие вопросы в дальнейшем.

Существующий код в проекте во многом не соответствует данному стандарту. Однако, нет смысла просто рефакторить код ради приведения его к красивому виду. Это отнимет время, не принеся заметной пользы, и усложнит изучение диффов. Поэтому есть смысл менять оформление только тех функций, которые вы модифицируете, решая задачи.