Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
Высшего профессионального образования

Московский государственный технический университет имени Н.Э. Баумана

Расчетно-пояснительная записка
к дипломному проекту
НА ТЕМУ:

Проверка корректности расстановки отступов исходных текстов программ на языке C.

Студент группы ИУ9-82
_____ Разборщикова Анастасия Викторовна
«___» _____ 2019 г.
Преподаватель
_____ Скоробогатов Сергей Юрьевич
«___» _____ 2019 г.

Москва
2019

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

При разработке программ необходимо уделять внимание не только логическому проектированию программы и ее коррекности, но и стилю оформления кода (coding style). Этой проблеме посвящаются книги и научные статьи. Наименование переменых, длина строк, использование шаблонов (patterns) проектирования, — все это напрямую влияет на понимание программы, возможность распространять ее и избежать проблем с дальнейшей поддержкой, ведь большую часть вренеми при разработке занимает именно чтение кода [1]. Были проведены исследования, доказывающие, что форматирование (стиль расстановки скобок, пробелы и т.п.) влияет на восприятие программы [2].

На сегодняшний день разработано множество стандартов оформления кода, для многих крупных проектов разрабатываются собственные соглашения (например [3], [4]). Однако студенты, только начавшие изучать программирование, часто не знакомы с ними, либо пренебрегают этими правилами и не стремятся к аккуратному оформлению кода.

На кафедре ИУ9 «Теоретическая информатика и компьютерные технологии» факультета «Информатики и систем управления» МГТУ им. Н. Э. Баумана для проверки работ студентов используется автоматическая система тестирования. После проверки программы на корректность и заимствования, она попадает на проверку преподавателю. На этом этапе встает вопрос о читаемости кода, адекватности его оформления. В связи с этим возникла необходимость разработки инструмента, который бы также проверял код на соответствие некоторым правилам форматирования и выдавал студентам сообщения об ошибках, фильтруя поток так, чтобы преподаватель мог проверять только корректно отформатированный код. Сложность задачи состоит в том, чтобы не делать условия корректности слишком жесткими (многие среды разработки программ автоматически форматируют код в соответствии со своими стандартами, которые могут отличаться, при этом являясь корректными) и поддерживать все возможные расширения компилятора GCC (the GNU Compiler Collection), установленного на сервере тестирования.

1 Постановка задачи

В рамках выпускной квалификационной работы ставится задача реализовать программу для проверки корректности расстановки отступов в программах на языке C.

Приложение должно выявлять *плохо отформатированные* программы и выдавать описание найденных ошибок.

***Определение* 1.** *В рамках данной задачи исходный код программы считается плохо отформатированным, если:*

1. *На одном уровне вложенности строки имеют разный отступ (за исключением строк, которые являются продолжением выражения, а также меток, включая метки* `case` *и* `default` *в теле оператора* `switch`*).*
2. *Строки кода на бóльшем уровне вложенности располагаются левее кода на меньшем уровне вложенности. Это правило не относится к меткам (за исключением меток* `case` *и* `default`*, которые должны быть не левее объявления блока* `switch`*) и директивам препроцессора, так как они могут не иметь отступа.*
3. *Строка, являющаяся продолжением выражения (перенос), располагается не левее предыдущей строки.*

Основные требования к программе:

1. Программа должна быть реализована в виде консольного приложения, которой подается на вход исходный код программы на языке C.
2. Программа должна выдавать сообщения об обнаруженных ошибках расстановки отступов.
3. Должна быть реализована возможность управлять тем, насколько подробно выводятся сообщения (т.е. поддерживать различные режимы диагностики).
4. Программа не должна реализовывать слишком жесткие требования к оформлению кода.
5. Программа должна запускаться на платформе GNU/Linux.
6. Должна быть реализована поддержка использования любых расширений компилятора GCC, которые могут использоваться в исходных кодах программ.

Пример, иллюстрирующий различия между кодом с допустимым форматированием и плохо отформатированным, приведен на рисунке 1.

```
1  void ok(void) {
2      int x = 1,
3      n = 5, i;
4      for(i = 0;
5          i < n; i++) {
6        x += x;
7      }
8      if (x % 2 == 0
9           || x > 0) {
10       printf("Ok!\n");
11     }
12 }
13
```

а) Допустимое форматирование.

```
1  void bad(void) {
2      int x = 1,
3    n = 5, i;
4      for(i = 0;
5          i < n; i++) {
6        x += x;
7      }
8        if (x % 2 == 0
9             || x > 0) {
10    printf("Bad!\n");
11     }
12 }
13
```

б) *Плохое* форматирование.

Рисунок 1 — Примеры кода с допустим и плохим форматированием.

2 Обзор предметной области

Введем некоторые определения.

***Определение* 2.** *Компиляция (compilation) — процесс перевода (translation) кода с исходного языка программирования в язык, пригодный для выполнения на ЭВМ [**?**, с. 4].*

Программа, выполняющая компиляцию, называется компилятором.

***Определение* 3.** *Статический анализ (англ. static analisis) — любой процесс проверки исходного кода без его выполнения [**?**, с. 3].*

Статический анализ входных данных может быть разделен на две части: *контроль синтаксиса* (syntax checks), во время которого выполняется проверка форматирования кода и *контроль семантики* (semantics checks), определяющий соответствие ввода логике и функциональности приложения [**?**, с. 120].

В дальнейшем будут рассматриваться статические синтаксические анализаторы — программы и инструменты, выполняющие статическую проверку синтаксиса.

***Определение* 4.** *Линтер (lint, linter), анализатор типа lint (lint-like tool, lint-like checker) — статический анализатор кода. Изначально lint — приложение для статического анализа программ, написанных на языке C, созданное еще в 1988 году [**?**]. Сегодня же это название часто используется в отношении анализаторов, выполняющих только проверку стиля кода [**?**, с. 120].*

На сегодняшний день существует много программ для форматирования исходного кода программ на языке C:

1. Системы автоформатирования и статические анализаторы, встроенные в интегрированные среды разработки (IDE, Integrated development environment), такие как CLion [**?**], QtCreator [**?**], Eclipse [**?**] и проч.
2. Автоформатирование в текстовых редакторах, например Sublime Text [**?**], Notepad++ [**?**], Kate [**?**]. Редактор Vim [**?**] имеет как встроенные решения для автоматической расстановки отступов [**?**], так и сторонние скрипты для автоформатирования, например vim-autoformat [**?**] и vim-clang-format [**?**].

   Инструменты, встроенные в текстовые редакторы и среды разработки, позволяют форматировать текст уже в процессе набора кода.

3. Самостоятельные системы автоформатирования кода (с поддержкой языка C): clang-format [**?**], Artistic Styler [**?**], Uncrustify [**?**], BCPP [**?**], GC GreatCode [**?**] и инструмент для преобразования отступов GNU Indent [**?**]. Эти программы получают на вход файл с исходным кодом и возвращают файл с текстом, преобразованным в соответствии с заданным стилем.
4. Статические синтаксические анализаторы(в этом списке только анализаторы, доступные для запуска на ОС GNU/Linux и поддерживающие языки C/C++): clang-tidy [**?**] и Clang Static Analyzer [**?**], cpplint [**?**], Cppcheck [**?**], Splint [**?**]. Эти инструменты анализируют исходный код программы и выдают сообщения об ошибках (некоторые, в том числе, об ошибках форматирования).

Несмотря на обширный список программ с похожей функциональностью, на момент написания данной работы программы, решающей поставленную задачу, найдено не было.

Напомним, что в данной работе ставится задача проверки файла исходного текста программы на единообразие оформления строк в соответствии с некоторыми критериями: приложение должно получать на вход файл исходного кода программы на языке C и выдавать сообщения об ошибках расстановки отступов. То есть приложение не изменяет полученный файл и не требует соответствия определенному стилю оформления кода.

Очевидно, для этой задачи не подходят системы автоформатирования, так как они преобразуют файл в соответствии с некоторым стилем, не выдавая сообщений об ошибках форматирования. Тем не менее, организация таких систем будет рассмотрена в дальнейшем и использована в разработке программы проверки корректности расстановки отступов, так как эта задача относится к области форматирования кода.

Однако на данном этапе наибольший интерес для нас представляют статические синтаксические анализаторы, так как они могут предоставить требуемую функциональность. Проведем тестирование существующих программ для выявления инструментов, обладающих необходимой функциональностью.

Рассматриваемые анализаторы поддерживают языки C/C++. Создадим небольшую тестовую программу `dummy.c`, содержащую явные ошибки форматирования (листинг 1) и проверим, выявят ли их указанные анализаторы.

Листинг 1 — Текст программы `dummy.c`, содержащий ошибку форматирования.

```
1  int main(){
2
3  int a = 1;      // Ошибка: отступ отличается на одном и том же
4      if (a ==2)  // уровне вложенности
5      {
6   return 2;      // Ошибка: вложенное выражение расположено левее
7      }           // объемлющего
8      if (1)
9          if (2) return 2;  // Ошибка: ветвь else относится
10     else                 // к вложенному оператору if
11         return 3;
12     return 0;
13 }
```

Данная программа содержит 3 ошибки первого типа (см. стр. 21):

1. Строка 4 расположена на том же уровне вложенности, что и строка 3, но имеет больший отступ.
2. Строка 6 расположена по отношению к строке 5 на большем уровне вложенности, однако имеет меньший отступ.
3. В строке 10 оператор `else` находится на одном уровне вложенности с оператором `if` в строке 9, а не в строке 8, следовательно, должен иметь от же отступ. Пример с такой ошибкой добавлен потому, что в данном случае неправильное восприятие кода может привести к неожиданным результатам выполнения такой программы, и поэтому многие статистические анализаторы содержат инструменты для выявления подобных случаев (потенциальных ошибок).

2.1 clang-tidy и Clang Static Analyzer

Многие из перечисленных инструментов (clang-format, clang-tidy и Clang Static Analyzer, vim-clang-format) основаны на Clang — проекте, предоставляющем фронтенд[1] и инструменты для обработки языков семейства C (включая C/C++)[**?**].

[1] Фронтенд (front-end) — стадия компиляции, производящая синтаксический анализ исходного кода: разбиение текста на значимые единицы и построение по некоторым правилам (грамматике языка) внутреннего промежуточного представления программы (intermediate representation) [**?**, с. 4].

2.1.1. clang-tidy

Clang-tidy — анализатор типа lint для диагностики и исправления таких ошибок в программах, как нарушение стиля и некорректное использование интерфейса, а так же ошибок, которые могут быть обнаружены с помощью статического нализа [**?**]. Это консольное приложение, которое принимает на вход файл исходного текста на языке C++ и возвращает сообщения об обнаруженных ошибках. Полученные в результате анализа исправления можно сохранить в некотором файле (если указан ключ `-export-fixes=<filename>` при запуске), а потом автоматически применить предлагаемые изменения к входному файлу с помощью инструмента `clang-apply-replacements`; либо применить правки сразу, используя опции `fix` и `fix-errors`.

Запустим clang-tidy с максимальным набором проверяемых параметров, задав параметр `-checks=*`. Результат запуска приведен в листинге 2.

Как видно из вывода приложения, из трех ошибок форматирования анализатор обнаружил только одну: разный отступ в ветвях оператора `if-else` в строке 10. Этой ошибке посвящены два сообщения. Приведем их описание:

– «`/home/nastasia/Diploma/dummy.c:10:5: предупреждение: различные отступы для 'if' и соответствующего ему 'else'`»;
– «`/home/nastasia/Diploma/dummy.c:10:9: предупреждение: утверждение должно быть обрамлено скобками`»;

Открыв список проверяемых условий (ckecks) [**?**], обнаруживаем, что среди них только одно правило `readability-misleading-indentation` относится к проверке отступов, и оно задает ограничения только на оформление оператора `if-else`: ветви одного условия должны иметь одинаковый отступ, и тела блоков должны выделяться фигурными скобками. Сообщения об этих ошибках мы видим в выводе приложения, другие же ошибки отступов были проигнорированы.

Clang-tidy имеет очень гибкие настройки стиля форматирования. Эта утилита использует те же настройки форматирования, что и clang-format. Настройки можно передавать, например, используя файл `./clang-format` (программа будет выполнять поиск файла с таким именем в ближайшей родительской директории) или в виде строки в формате JSON. Подробное описание ключей для задания стиля форматирования приведено в документации clang-format [**?**].

Листинг 2 — Результат запуска программы clang-tidy.

```
 > clang-tidy -checks=* dummy.c
Error while trying to load a compilation database:
Could not auto-detect compilation database for file "dummy.c"
No compilation database found in /home/nastasia/Diploma or any parent directory
fixed-compilation-database: Error while opening fixed database: No such file or directory
json-compilation-database: Error while opening JSON database: No such file or directory
Running without flags.
15 warnings generated.
/home/nastasia/Diploma/dummy.c:8:9: warning: converting integer literal to bool, use bool literal instead [modernize-use-bool-literals]
    if (1)
        ^
note: this fix will not be applied because it overlaps with another fix
/home/nastasia/Diploma/dummy.c:8:9: warning: implicit conversion 'int' -> bool [readability-implicit-bool-conversion]
note: this fix will not be applied because it overlaps with another fix
/home/nastasia/Diploma/dummy.c:8:11: warning: statement should be inside braces [google-readability-braces-around-statements]
    if (1)
          ^
           {
/home/nastasia/Diploma/dummy.c:9:13: warning: converting integer literal to bool, use bool literal instead [modernize-use-bool-literals]
        if (2) return 2;  // Ошибка: ветвь else относится
            ^
note: this fix will not be applied because it overlaps with another fix
/home/nastasia/Diploma/dummy.c:9:13: warning: implicit conversion 'int' -> bool [readability-implicit-bool-conversion]
note: this fix will not be applied because it overlaps with another fix
/home/nastasia/Diploma/dummy.c:9:15: warning: statement should be inside braces [google-readability-braces-around-statements]
        if (2) return 2;  // Ошибка: ветвь else относится
              ^
               {
/home/nastasia/Diploma/dummy.c:10:5: warning: add explicit braces to avoid dangling else [clang-diagnostic-dangling-else]
    else                  // к вложенному оператору if
    ^
/home/nastasia/Diploma/dummy.c:10:5: warning: different indentation for 'if' and corresponding 'else' [readability-misleading-indentation]
/home/nastasia/Diploma/dummy.c:10:9: warning: statement should be inside braces [google-readability-braces-around-statements]
    else                  // к вложенному оператору if
        ^
         {
```

Каждый ключ из описания стиля форматирования кода, относящийся к отступам, задает определенную ширину отступа для каждого случая (отступ содержания блока в теле условных операторов, оступ метки `case` относительно оператора `switch` и т. д.). Однако в данной работе необходимо не осуществить

проверку исходного кода программы на соответствие определенному стилю, а проверить единообразие расстановки отступов для *любого* стиля (который может отличаться в различных файлах и заранее не известен).

2.1.2. Clang Static Analyzer

Clang Static Analyzer анализирует исходный код на этапе компиляции. Согласно информации на сайте проекта [**?**], статический анализатор должен расширять возможности компиляторов и «идти дальше», сообщая о потенциальных ошибках времени выполнения. Также сообщается, что из-за глубины анализа и сложности алгоритмов статический анализ может занимать намного больше времени, чем компиляция программы, поэтому для ускорения работы Clang Static Analyzer имеет ограничения на количество проверок и, в связи с этим, может выявлять не все ошибки. Также возможны случаи ложных срабатываний.

В отличие от clang-tidy, который является линтером и проверяет стиль оформления кода, Clang Static Analyzer не имеет соответствующих флагов в настройке. Этот инструмент больше подходит для анализа всего проекта, что становится возможным благодаря «встраиванию» анализатора в процесс сборки.

Попробуем запустить этот анализатор и посмотреть, сможет ли он найти по крайней мере ошибку в строке 10, которая может привести к неверным результатам выполнения программы.

Анализ запускается командой `scan-build`, поставленной перед командами для сборки и компиляции проекта. Результат запуска приведен в листинге 3. Ни одной ошибки найдено не было (`no bugs found`).

Листинг 3 — Clang Static Analyzer.

```
1 > scan-build gcc dummy.c
2 scan-build: Using '/usr/lib/llvm-6.0/bin/clang' for static analysis
3 scan-build: Removing directory '/tmp/scan-build
    -2019-06-09-203732-25240-1' because it contains no reports.
4 scan-build: No bugs found.
```

2.2 cpplint

Согласно информации, приведенной на странице проекта на сайте GitHub [**?**], инструмент осуществляет проверку, что файл исходного текста на C++ соответствует стандарту форматирования Google [3]. Относительно отступов в стандарте указывается только то, что размер отступа по умолчанию равняется двум пробелам.

В описании к программе также указано, что для проверки используются регулярные выражения[1] в связи с чем не все ошибки могут быть выявлены, а также случаются ложно-положительные срабатывания.

Анализатор cpplint принимает на вход только файлы исходного кода на языке C++, поэтому изменим расширение тестового файла, переименовав `dummy.c` в `dummy.cc` (возможность анализа файлов, содержащих тексты программ на языке C++ в данном случае возможно и даже избыточно, так как язык C++, с некоторыми исключениями, является надмножеством языка C [**?**]).

Программа распространяется в виде Python-скрипта. Запустим проверку. Согласно документации, создаем в том же каталоге, где находится файл `dummy.cc`, конфигурационный файл `CPPLINT.cfg` и добавим в вывод все фильтры, в названиях которых присутствует `readability` и `whitespace`. Содержание конфигурационного файла приведем в листинге 4.

Листинг 4 — Содержание файла `CPPLINT.cfg`.

```
1 filter=+whitespace/indent,+build/class,+build/c++11,+readability/
    ↪ alt_tokens,+readability/braces,+readability/casting,+readability/
    ↪ check,+readability/constructors,+readability/fn_size,+readability/
    ↪ inheritance,+readability/multiline_comment,+readability/
    ↪ multiline_string,+readability/namespace,+readability/nolint,+
    ↪ readability/nul,+readability/strings,+readability/todo,+
    ↪ readability/utf8,+whitespace/blank_line,+whitespace/braces,+
    ↪ whitespace/comma,+whitespace/comments,+whitespace/
    ↪ empty_conditional_body,+whitespace/empty_if_body,+whitespace/
    ↪ empty_loop_body,+whitespace/end_of_line,+whitespace/ending_newline
    ↪ ,+whitespace/forcolon,+whitespace/indent,+whitespace/line_length,+
    ↪ whitespace/newline,+whitespace/operators,+whitespace/parens,+
    ↪ whitespace/semicolon,+whitespace/tab,+whitespace/todo
```

[1] Регулярное выражение (англ. regular expression, regexp) — вид текстового шаблона (text pattern), который может использоваться во многих современных приложениях и языках программирования, например для проверки, что входной текст соответствует шаблону, для выполнения поиска частей текста, соответствующих шаблону, в большем блоке текста, для замены текста, соответствующего шаблону, на другой текст, разбивать текст на блоки в соответствии с шаблоном и т.д. [**?**]

Запустим анализ с максимальным уровнем вывода (`--verbose=0`). Результат запуска приведен в листинге 5.

Листинг 5 — Результат запуска программы cpplint.

```
 > python cpplint.py --verbose=0  dummy.cc
dummy.cc:0:  No copyright message found.  You should have a line:
   "Copyright [year] <Copyright Owner>"  [legal/copyright] [5]
dummy.cc:1:  Missing space before {  [whitespace/braces] [5]
dummy.cc:2:  Line ends in whitespace.  Consider deleting these extra
   spaces.  [whitespace/end_of_line] [4]
dummy.cc:2:  Redundant blank line at the start of a code block should be
    deleted.  [whitespace/blank_line] [2]
dummy.cc:5:  { should almost always be at the end of the previous line
   [whitespace/braces] [4]
dummy.cc:6:  Weird number of spaces at line-start.  Are you using a 2-
   space indent?  [whitespace/indent] [3]
dummy.cc:9:  Line ends in whitespace.  Consider deleting these extra
   spaces.  [whitespace/end_of_line] [4]
dummy.cc:9:  Else clause should be indented at the same level as if.
   Ambiguous nested if/else chains require braces.  [readability/
   braces] [4]
Done processing dummy.cc
Total errors found: 8
```

Приведем перевод сообщений об интересующих нас ошибках:

- «`dummy.cc:6: Странное количество пробелов в начале строки. Вы используете отступы величиной в два пробела?`»;
- «`dummy.cc:9: Ветвь else должна иметь отступ, равный отступу if. Двусмысленные вложенные последовательности if/else требуют обрамления скобками.`»;

Как мы видим, из трех ошибок в тестовом файле (см. стр. 8) были найдены две (в строках 6 и 9), причем из описания ошибки в строке 6 неясно, в чем она заключается.

Таким образом, данный инструмент не может использоваться для решения поставленной задачи, несмотря на то, что среди приложений, рассмотренных в данной работе, cpplint обнаружил наибольшее количество ошибок.

2.3 Cppcheck

Согласно информации на странице проекта [**?**], Cppcheck — статический анализатор кода на языке C/C++, предоставляющий возможность обнаружения ошибок и нацелен на выявление неопределенного поведения (undefined behaviour) и *опасные конструкции кода*. Цель проекта - обнаруживать только

настоящие ошибки в коде, иными словами, уменьшить количество ложных срабатываний. [**?**]

Программа получает на вход файл исходного текста программы и возвращает сообщения об ошибках, записывая при этом результат анализа в виде XML-файла с расширением `.dump`. Эти данные содержат промежуточное представление программы, которое можно в дальнейшем анализировать дополнительными скриптами (addons). На сайте проекта не было найдено скрипта, который относился бы к решаемой задаче.

В справочной информации (на сайте проекта либо в командной строке в результате запуска команды `cppcheck --doc`) не было найдено флагов, запускающих проверки, относящихся к стилю оформления кода или расстановке отступов

Проверим, обнаружит ли анализатор хотя бы потенциальную ошибку в строке 10 тестового файла (см. листинг 1 на с. 8), которую обнаружили анализаторы Cpplint и clang-tidy.

Результат запуска приводится в листинге 6.

Листинг 6 — Результат запуска программы Cppcheck.

```
1 > cppcheck dummy.c
2  Checking dummy.c ...
```

В процессе анализа не было найдено ни одной ошибки. Таким образом, статический анализатор Cppcheck не подходит для решения поставленной задачи.

2.4 Splint

Splint (Secure Programming Lint) — инструмент для статического анализа программ, написанных на языке C, на предмет уязвимостей безопасности и ошибках в коде. При необходимой настройке может быть использован как лучшая версия упомянутой ранее утилиты lint. На сайте программы также утверждается, что если в программу добавлены аннотации[1], Splint может выполнять

[1]Аннотации в языке C — расширение языка, которое позволяет явно описывать поведение функции [**?**]. Аннотации не входят в стандарт языка C, являются платформо-зависимым решением и могут отличаться для разных компиляторов (ср. аннотации Microsoft VC++ [**?**] и Clang [**?**]).

более глубокую проверку кода, нежели любой стандартный линтер [**?**] (при этом не указывается, какие линтеры считаются стандартными).

Проверим, какие ошибки в тестовом файле обнаружит данный анализатор. В документации к проекту перечислены флаги и режимы запуска. Среди флагов к отступам и форматированию относятся либо проверки форматных строк в функциях типа `printf`, либо к форматированию и расстановке отступов в выводе самой утилиты. Среди режимов запуска выберем режим `strict`, выполняющий максимальное количество проверок кода.

Результат запуска приложения приведен в листинге 7.

Листинг 7 — Результат запуска программы Splint.

```
splint +strict dummy.c
Splint 3.1.2 --- 20 Feb 2018

dummy.c:1:5: Function main declared without parameter list
  A function declaration does not have a parameter list. (Use -noparams to
  inhibit warning)
dummy.c: (in function main)
dummy.c:8:9: Test expression for if not boolean, type int: 1
  Test expression type is not boolean or int. (Use -predboolint to inhibit
  warning)
dummy.c:9:13: Test expression for if not boolean, type int: 2
dummy.c:9:23: Body of if clause of if statement is not a block: return 2
  If body is a single statement, not a compound block. (Use -ifblock to inhibit
  warning)
dummy.c:11:16: Body of else clause of if statement is not a block: return 3
dummy.c:11:16: Body of if statement is not a block:
                 if (2) return 2 else return 3

Finished checking --- 6 code warnings
```

Как видно из листинга, ни одна из требуемых ошибок (см. стр. 2) не была обнаружена. Хотя для строки 9 было выведено предупреждение об отсутствии фигурных скобок в теле оператора `if`: «`dummy.c:9:23: Тело ветви else конструкции if не является блоком: return 2. Тело if состоит из единственного утверждения, не составной блок`».

Исходя из полученных результатов, делаем вывод, что утилита Splint не пригодна для решения поставленной задачи поиска ошибок в расстановке отступов.

2.5 Выводы обзорной части

Для удобства анализа полученных данных сведем результаты в таблицу. Напомним, что каждому из рассмотренных статических синтаксических анализаторов на вход был подан файл `dummy.c` (см. листинг 1 на стр. 8). Файл содержит в себе три ошибки расстановки отступов, которые влияют на качество восприятия кода и, согласно заданию, должны быть выявлены, а пользователю должны быть показаны сообщения, описывающие суть ошибки (описание ошибок см. на стр. 8).

Сводка результатов приведена в таблице 1.

Анализатор	Разный отступ в блоке	Отступ противоречит уровню вложенности	Разный отступ ветвей оператора `if-else`
clang-tidy	—	—	Предупреждение об отсутствии скобок. Указание на двусмысленность выражения.
Clang Static Analyzer	—	—	—
cpplint	—	Предупреждение о «странном» отступе. Причина неясна.	Сообщение об ошибке отступа. Напоминание о скобках.
Cppcheck	—	—	—
Splint	—	—	Предупреждение об отсутствии скобок.

Таблица 1 — Сравнение рассмотренных статических синтаксических анализаторов на предмет выявления ошибок в тестовом файле `dummy.c`

Проанализировав полученные результаты, можно сделать следующий вывод: несмотря на то, что статические анализаторы могут предоставить требуемую функциональность (а именно проверять корректность расстановки отступов в зависимости от уровня вложенности, см. раздел «Постановка задачи») и некоторые действительно частично ее предоставляют (clang-tidy, cpplint), они не могут быть использованы для решения по следующим причинам:

1. Статические анализаторы разрабатываются для определения потенциальных ошибок времени выполнения программы, и задача проверки форматирования не является приоритетной, поэтому анализ отступов

либо не удовлетворяет требованиям данного задания, либо отсутствуют вовсе.

2. Если проверка стиля оформления кода выполняется, она производится в соответствии с некоторым установленным стандартом. Такое решение удобно при встроенной функции автодополнения в среде разработки и целесообразно для больших проектов, требующих непрерывной поддержки. Однако в среде тестирования учебных программ, которой пользуются начинающие программисты, требование жестко соблюдать стиль форматирования исходного текста программы будет выставлять слишком высокий порог прохождения тестирования, так как будут отклоняться даже хорошо отформатированные программы, но написанные в другом стиле.

Таким образом возникает необходимость разработки нового инструмента для проверки корректности расстановки отступов с учетом того, что эта программа будет использоваться в обучающих целях.

2.6 Программная архитектура существующих статических анализаторов и утилит форматирования

Рассмотрим принципы построения статического анализатора cpplint, который показали наилучший результат среди рассмотренных. Также интерес представляет Clang, на котором основан статический анализатор clang-tidy. В семействе утилит, основанных на Clang, изучим устройство программы, которая, в отличие от clang-tidy, предназначенной для автоматического форматирования исходных текстов программ. Как было указано выше, сама эта программа не может быть применена для решения поставленной задачи, так как в результате работы выдает отформатированный по заданным правилам код, а не сообщения об ошибках. Однако изучение ее внутренней организации может быть полезно при проектировании приложения, проверяющего корректность форматирования кода.

Все из перечисленных утилит являются свободными программами[1], поэтому конфликта с условиями использования программ при анализе и модификации их исходного кода не возникает.

<TODO: Тут вкратце описать cpplint и clang, что разбор кода происходит следующим образом: лексичсекий анализ, доработанный для сохранения пробелов в токенах, синтаксический анализ и построение AST, анализ AST; На основании этого в основной части потом описать, что в реализованной программе: лексический анализ, синтаксический анализ по упрощенной грамматике, построение не дерева, а таблицы, анализ этой таблицы, оценить сложность алгоритма>

[1] Термин «свободная программа» означает, что у пользователей есть четыре существенных свободы: 1) выполнять программу, 2) изучать и править программу в виде исходного текста, 3) перераспространять точные копии и 4) распространять измененные версии .

3 Руководство пользователя

3.1 Системные требования

Программа распространяется в виде исходного кода на платформе GitHub [5]. Для запуска необходимо иметь следующее ПО:

1. ОС GNU/Linux (компиляция и запуск приложения тестировались на платформах Gentoo Linux и Linux Mint, а также Windows 8, Windows 10 и MacOS High Sierra).
2. Компилятор GCC [**?**] версии не ниже 7 (либо другой компилятор языка C++, поддерживающий стандарт C++17),
3. CMake [**?**] версии от 3.10.2.

3.2 Установка и запуск

В папке с исходным кодом проекта `c-format-checker` находится скрипт `compile.bash`. (Данный скрипт написан для исполнения в командной оболочке Bash ОС GNU/Linux и может потребовать модификации для корректной работы на другой платформе. Также команды будут отличаться для разных компиляторов. Дальнейшие команды приведены для ОС GNU/Linux с установленными утилитами Cmake, make, компилятором GCC 7) Из папки `c-format-checker` в терминале необходимо выполнить следующие команды (листинг 8):

Листинг 8 — Сборка проекта.

```
$ chmod +x compile.bash  # Сделать файл исполняемым.
$ ./compile.bash         # Запустить сборку.
```

В результате выполнения скрипта в папке появятся директории `build` (содержит временные файлы `CMake`) и `bin`. В папке `bin` находится исполняемый файл `c-format-checker` (для упрощения вызова приложения можно переместить исполняемый файл в нужную директорию, либо добавить путь к папке `bin` в системную переменную `PATH` и вызывать программу как системную утилиту).

Программа является консольным приложением. Оно имеет следующие параметры вызова (листинг 9):

Листинг 9 — Формат вызова приложения.

```
$ ./c-format-checker [-f] ФАЙЛ [-q | -v | -d] [-l ФАЙЛ]
```

Программе на вход подаются имена файлов, разделенные пробелом, а также необязательный параметр (:

-f, --file Файл исходного текста программы на языке C (явное указание ключа `-f` не обязательно).

-q, --quiet Выводить кратко описание ошибок (по умолчанию).

-v, --verbose Выводить развернутые описания ошибок.

-d, --debug Выводить отладочную информацию.

--help Помощь.

Каждый файл должен содержать исходный код программы на языке C. Список файлов для анализа подается подряд, не разбивается другими ключами. Благодаря возможностям языка командной оболочки Bash, можно запускать обработку файлов, выполнив поиск по сложному запросу (см. листинг 10):

Листинг 10 — Пример вызова приложения.

```
# Обработка всех файлов в папке "011 Подсчёт слов в строке"
# в режиме verbose:
$ c-format-checker "011 Подсчёт слов в строке"/* -v
...
# Обработка всех файлов в папке "src" с расширением .c в режиме debug:
$ c-format-checker -d \$(find src -name *.c)}
```

Ключи -q, -v, -d, указывающие, насколько подробным должен быть вывод приложения, не могут встречаться в списке аргументов более одного, в противном случае будет выведено сообщение с подсказкой, и программа завершится с кодом ошибки 1. Если ни один из ключей не передан, вывод производится в режиме `verbose`.

Каждый файл из переданного списка обрабатывается программой. Вывод сообщений по умолчанию производится в стандартный поток (консоль). В результате выполнения скрипта в директории, из который производился запуск приложения, появится файл `cfc-statistics.txt`, содержащий список файлов, в которых были обнаружены некорректные отступы. Если в файле встречается хотя бы одна *непустая*[1] строка кода с отступом, в котором содержится символ

[1] Здесь под непустой строкой подразумевается строка, содержащая текста помимо пробельных символов, комментариев и директив препроцессора. Указанные элементы игнорируются.

табуляции, напротив имени этого файла в документе `cfc-statistics.txt` ставится пометка «(tab.)».

3.3 Вывод приложения и сообщения об ошибках

Программа выделяет два вида ошибок при расстановке отступов:

1. Строка имеет иной отступ, чем строки выше на том же уровне вложенности, либо находится левее относительно выражения, расположенного на верхнем уровне вложенности по отношению к данной.
2. Использование табуляций (так как нет единого стандарта о ширине символа при отображении).

Если программа обнаружила только ошибки первого вида, сообщения о них выводятся в соответствующем режиме в стандартный поток вывода и в файл `cfc-statistics.txt` записывается имя файла.

Если в процессе анализа найдена хотя бы одна ошибка второго вида, выдается соответствующее сообщение, в файл `cfc-statistics.txt` записывается имя файла с пометкой «(tab.)».

В режиме `quiet` (является форматом вывода по умолчанию) для каждого файла в стандартный поток выводится краткая сводка об обнаруженных некорректных отступах, а также сообщения об ошибках, возникших в процессе выполнения программы. Об ошибках отступов сообщается в следующем формате: «`<имя файла> (строка <номер>): <сообщение>.`»

«Сообщение» — предложение из следующего списка:

– Использование табуляции.
– Различные отступы на одном уровне вложенности.
– Ошибка отступа при переносе выражения.
– Вложенное выражение левее объемлющего.

Пример вывода приложения приведен в листинге 11.

Если ошибок не обнаружено, программа ничего не выводит и завершается с кодом 0.

В режиме `verbose` в стандартный поток выводятся развернутые сообщения о некорректных отступах и сообщения об ошибках. Сообщения имеют следующий вид: «`<имя файла> (строка <номер>): ошибка: отступ ширины <число>: <образ>. <Подсказка>`».

Листинг 11 — Запуск приложения в режиме `quiet`.

```
> c-format-checker 'Test 1'/* -q
...
Test 1/X.c (строка 20): Различные отступы на одном уровне вложенности.
Test 1/X.c (строка 21): Использование табуляции.

Test 1/Y.c (строка 10): Использование табуляции.

...
```

«Образ» — печатное представление символов отступа: «␣» для пробела и «\t» для табуляции. При вычислении величины отступа ширина табуляции в программе берется равной четырем пробелам.

«Подсказка» — это сообщение, описывающее ошибку. Если в отступе в начале строки встретился символ табуляции, выдается сообщение «Использование пробелов и табуляций в одном отступе» или «Использование табуляций (ранее использовались пробелы)».

Если ошибка связана с тем, что какая-то строка имеет отступ, не равный таковому для строки выше на том же уровне вложенности, подсказка имеет вид: «Ранее на том же уровне вложенности в строке <номер сроки выше> отступ ширины <число>: <образ отступа>.»

Если при переносе выражения на следующую строку продолжение оказалось левее начала выражения, дополнительно выводится сообщение «Продолжение выражения левее предыдущей строки.» Пример вывода приложения в режиме `verbose` приведен в листинге 12.

Листинг 12 — Запуск приложения в режиме `verbose`.

```
> c-format-checker 'Test 1'/* -v
Test 1/X.c (строка 20): ошибка: отступ ширины 8: <␣␣␣␣␣␣␣␣>. Ранее на
  том же уровне вложенности в строке 19 отступ ширины 4: <␣␣␣␣>.
Test 1/X.c (строка 21): ошибка: отступ ширины 4: <\t>. Использование
  табуляций (ранее использовались пробелы).

Test 1/Y.c (строка 10): ошибка: отступ ширины 8: <\t␣␣␣␣>.
  Использование пробелов и табуляций в одном отступе..
```

Соответствие выводимых сообщений в режимах `quiet` и `verbose` при различных ошибках приведено в таблице 2.

В режиме `debug` в стандартный поток вывода выводится вся отладочная информация.

Режим `quiet`	Режим `verbose`
”Вложенное выражение левее объемлющего.”	”Вложенное выражение левее объемлющего.”
”Ошибка отступа при переносе выражения.”	”Продолжение выражения левее предыдущей строки.”
”Использование табуляции.”	”Использование пробелов и табуляций в одном отступе.”
	”Использование табуляций (ранее использовались пробелы).”
”Различные отступы на одном уровне вложенности.”	Ранее на том же уровне вложенности в строке <номер сроки выше> отступ ширины <число>: <образ отступа>.

Таблица 2 — Сопоставление сообщений об ошибках в режимах `quiet` и `verbose`.

Эти данные содержат список токенов и вывод правил грамматики, полученные в результате лексического и синтаксического разбора, а также промежуточные таблицы анализатора, содержащие уровень вложенности, на котором находится каждая строка файла. Сообщения о найденных ошибках отступов выводятся после отладочной информации в таком же формате, как в режиме `verbose`. Если программе передан аргумент `–help`, выводится справочная информация (см. листинг 13) и программа завершается с кодом 0. В случае, если параметры были переданы некорректно, программа завершается с кодом 1 («Некорректные аргументы командной строки.»), показав пользователю сообщение с подсказкой: «`Использование: c-format-checker [-f] ФАЙЛЫ [-q | -v | -d]. Запустите «c-format-checker –help» для более подробного описания.`»

Если в процессе работы программы была обнаружена лексическая или синтаксическая ошибка, в стандартный поток вывода будет отправлено сообщение с именем файла, координатами ошибки в файле, ее описанием и функцией, в которой возникло исключение (см. пример в листинге 14). Далее программа переходит к анализу следующего файла. (Однако не следует целенаправленно использовать данную программу для проверки корректности синтаксиса, так как она производит разбор программы по упрощенной грамматике, следовательно, не все ошибки могут быть обнаружены).

Листинг 13 — Справочное сообщение (запуск приложения с ключом «--help»).

```
НАЗВАНИЕ
      c-format-checker - проверка файла исходных текстов программ на
  языке C на корректность расстановки отступов.

СИНТАКСИС
      c-format-checker [-f] ФАЙЛЫ [-q | -v | -d]

ОПИСАНИЕ
      Программа сканирует файл и проверяет корректность расстановки
  отступов
ОПЦИИ
      -f, --file    Файлы исходного текста программы на языке C
  ключ( -f не требуется, если этот аргумент идет первым).
      -q, --quiet   Выводить кратко описание ошибок по( умолчанию).
      -v, --verbose Выводить развернутые описания ошибок.
      -d, --debug   Выводить отладочную информацию.
          --help    Помощь.

СТАТУС ВЫХОДА
      0   Нормальный выход.
      1   Некорректные аргументы командной строки.
      2   Ошибка чтениязаписи/ в файл.
```

Листинг 14 — Пример сообщения об ошибке на этапе синтаксического анализа.

```
Test 2/SE.c: Ошибка на этапе синтаксического анализа: [c-format-checker/
   src/frontend/parser/parse_rules.cpp, line 327] Error at (9, 78)-
   (9, 78) - expected SEMICOLON, got END_OF_FILE.
```

3.4 Критерии корректности форматирования кода

Программа считается корректной, если на одном уровне вложенности все строки имеют одинаковый отступ, вложенные выражения располагаются не левее строк с объемлющими выражениями, при переносе строки продолжение выражения располагается не левее его начала. Также программа не должна содержать символов табуляции в начале строки (при этом не накладывается никаких ограничений на их использование в середине или конце строки).

Следующим уровнем вложенности считаются:

1. Блок.
2. Тело функции.
3. Тело конструкций условных переходов `if`, `else`, `switch`.
4. Тело цикла `for`, `while`, `do`.
5. Пользовательская метка, метки `case` и `default`.

6. Блок кода, следующий за метками `case` и `default` внутри тела `switch`.
7. Тело объявления структуры `struct`, объединения `union` и перечисления `enum`.

Не считаются следующим уровнем вложенности:

1. Список инициализации.
2. Перенос выражения на следующую строку.
3. Выражения в скобках.
4. Условные выражения в конструкциях `if`, `switch`, `for`, `while`.

Случаи, в которых ослабляется требование равенства всех отступов на одном уровне вложенности:

1. Метки могут располагаться левее или правее основного блока кода.
2. При переносе выражения на другую строку ослабляется требование равенства отступов. Перенос считается корректным, если новая строка начинается не левее предыдущей.
3. Внутри блока `switch` метки `case` и `default` могут располагаться левее прочего кода в том же блоке, но на одном уровне друг с другом и не левее самого оператора `switch`.
4. Внутри блока `switch` код, который предшествует самой первой метке `case`, имеет собственный уровень вложенности и не обязан совпадать по ширине отступа с метками или последовательностью кода после них.

Примеры корректого и некорректного форматирования тела оператора `switch` приведены на рисунке 2.

```
switch(x)
{
    int i=4;
    f(i);
case 0:
    i=17;
default:
    i=2;
}
```

а) Корректно (стиль из стандарта ISO/IEC9899:2017 [**?**, с. 109])

```
switch(x)
{
    int i=4;
    f(i);
    case 0:
        i=17;
    default:
        i=2;
}
```

б) Корректно (форматирование среды разработки CLion)

```
switch(x)
{
    int i=4;
f(i);
    case 0:
          i=17;
    default:
        i=2;
}
```

в) Некорректно

Рисунок 2 — Пример корректного (а, б) и некорректного (в) форматирования тела оператора `switch`.

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Lee T., Lee J. B., In H. P. Impacts of Coding Practices on Readability // International Journal of Software Engineering and Its Applications. Gothenburg, Sweden, 2013. Vol. 7, no. 5. P. 413–422. Access: https://pdfs.semanticscholar.org/1727/4fef400424fe4876cd23edb8318e4944b203.pdf.

[2] Dos Santos R. M., Gerosa M. A. Impacts of Coding Practices on Readability // ICPC'18: 26th IEEE/ACM International Conference on Program Comprehension. Gothenburg, Sweden, 2018. 9 p. Access: https://doi.org/10.1145/3196321.3196342. URL: https://www.ime.usp.br/ gerosa/papers/ICPC2018-Legibility.pdf.

[3] Google Style Guides. URL: http://google.github.io/styleguide/ (дата обращения: 30.05.2019).

[4] Qt Coding Style. URL: https://wiki.qt.io/Qt_Coding_Style (дата обращения: 30.05.2019).

[5] Исходный код приложения c-format-checker. URL: https://github.com/nastusha-merry/c-format-checker (дата обращения: 30.05.2019).