

CCGLImperator

C++ Core Guidelines Rules Checker and Quick
Fixes

Bachelor Thesis

Department of Computer Science
University of Applied Science Rapperswil

Spring Term 2017

Authors:	Rolf Bislin, Kilian Diener
Advisors:	Thomas Corbat, Felix Morgner
Project Partner:	IFS Institute for Software
Duration:	20.02.2017 – 16.06.2017

I Abstract

C++ Core Guidelines [SS15] are a set of rules to encourage the use of modern C++, which is a simpler and safer subset of the language.

In this bachelor thesis the already existing Eclipse CDT plug-ins CCGLator and GslAtrPtr have been improved and extended. CCGLator supports programmers using C++ Core Guidelines in everyday programming. In the scope of this project new rules were added, performance enhancements were implemented and the code quality was improved.

GslAtrPtr focuses solely on the rules concerning the correct use of pointers. It was improved to handle more dynamic function interfaces and support more specific types in the quick fixes. Additionally, the support to set attributes or comments to ignore specific rules was added, resulting in a coherent handling of ignoring rules in CCGLator and GslAtrPtr.

II Management Summary

II.1 Introduction

C++ Core Guidelines [SS15] are a set of rules to encourage the use of modern C++, which is a simpler and safer subset of the language.

In this bachelor thesis the already existing Eclipse CDT plug-ins CCGLator and GslAtorPtr have been improved and extended. CCGLator supports programmers using C++ Core Guidelines in everyday programming. It was developed in two previous projects. GslAtorPtr was also created in a previous project and focuses on the rules concerning the correct use of pointers.

II.2 Approach

First, the plug-in CCGLator has been tested on a real-world project to see how it performs on a large codebase. Based on this assessment several improvements have been implemented and errors have been fixed.

Then the work split into two parts. One was the addition of new rules to CCGLator and the other was the improvement of GslAtorPtr. Because the two plug-ins are very similar, the architectures of both could be compared during the implementation, and flaws in either have been eliminated, resulting in a cleaner and more alike structure. Additionally, new insights were gained which resulted in improvements to both plug-ins.

II.3 Results

CCGLator now has better performance, reports fewer false positives and supports five additional rules. GslAtorPtr handles a wider range of function interfaces and supports more specific types in the quick fixes.

Also, the added possibility to set attributes or comments to ignore specific rules provides convenient and coherent handling of ignoring rules in both plug-ins. Additionally, a lot of work was put into the infrastructure of both plug-ins to clean up the code and make the plug-ins perform better in general.

Furthermore, a contribution to the Eclipse CDT project was made, which has already been merged into the project and is bound to be published in the next release.

II.4 Outlook

The C++ Core Guidelines still hold a lot of unimplemented rules and new ones are added frequently allowing for a lot of future work for CCGLator. Besides, some false positives are possible to still be present which could be detected and fixed. GslAtrPtr still holds some points in the future work chapter which could be implemented in the future.

Additionally, a lot of features are now implemented twice in CCGLator as well as in GslAtrPtr. This code could be extracted into a "base plug-in" to avoid duplication.

III Declaration of Authorship

We hereby declare,

- that this project was done without external assistance except the ones declared in the documentation or discussed with the advisor.
- that all the used sources are cited according to the usual scientific citation rules.
- that no resources protected under copyright law (e.g. images) are illegitimately used in this project.

Place and date

Rolf Bislin

Place and date

Kilian Diener

Contents

I	Abstract	i
II	Management Summary	ii
II.1	Introduction	ii
II.2	Approach	ii
II.3	Results	ii
II.4	Outlook	iii
III	Declaration of Authorship	iv
1	Introduction	6
1.1	Previous Work	6
1.1.1	CCGLator	6
1.1.2	CharWars and GslAtrPtr	6
1.2	Eclipse CDT	6
1.2.1	Cevelop	7
1.2.2	Codan	7
1.3	Software Stack	7
1.4	Scope Definition	8
1.4.1	Minimal Scope	8
1.4.2	Optimal Scope	8
1.4.3	Maximum Scope	9
1.5	Results	9
2	Analysis CCGLator	10
2.1	C++ Core Guidelines	10
2.1.1	GSL: Guideline Support Library	11
2.2	Testing on Real World Application	11
2.2.1	C.83: For value-like types, consider providing a noexcept swap function	12
2.2.2	C.84: A swap function may not fail	12
2.2.3	C.85: If a user defined swap member function is used, namespace- level swap(a, b) should be overwritten	13
2.2.4	C.164: Avoid conversion operators	13
2.2.5	ES.26: Don't use a variable for two unrelated purposes	13
2.2.6	ES.46: Avoid lossy (narrowing, truncating) arithmetic con- versions	14
2.2.7	ES.49: If you must use a cast, use a named cast	16

2.2.8	ES.74: Prefer to declare a loop variable in the initializer part of a for-statement	16
2.2.9	C.20: If you can avoid defining default operations, do	16
2.2.10	Quick Fixes	17
2.3	ES.75: Avoid do-statements	18
2.3.1	Enforcement	18
2.3.2	Pre fix Code	18
2.3.3	Post fix Code	18
2.4	ES.76: Avoid goto	19
2.4.1	Enforcement	20
	Skipping a Codepart on Some Condition	20
	Loop Back to an Earlier Code Part on Some Condition	21
	Conclusion	23
2.4.2	Pre fix Code	24
2.4.3	Post fix Code	24
2.5	ES.78: Always end a non-empty case with a break	25
2.5.1	Enforcement	25
2.5.2	Pre fix Code	26
2.5.3	Post fix Code	26
2.6	ES.9: Avoid ALL_CAPS names	27
2.6.1	Enforcement	27
2.6.2	Problematic Code	27
2.7	ES.50: Don't cast away const	28
2.7.1	Enforcement	28
2.7.2	Pre fix Code	29
2.7.3	Post fix Code	29
3	Analysis GslAtorPtr	30
3.1	string_span	30
3.1.1	Difference Between span and string_span	30
3.1.2	String_span Types	31
3.1.3	Trampoline Function for string_span	31
3.1.4	String_span Rewrite Quick Fix	32
3.1.5	Fixing C Strings with string_span	33
3.2	string_view	34
3.2.1	What is a string_view?	34
3.2.2	string_view Types	34
3.2.3	string_span or string_view	34
3.3	Improvement of the span<T>Refactoring	35
3.3.1	Multiple Pointer and Size Parameter Combinations	35
3.3.2	Multiple Pointer and One Size Parameter	36

3.3.3	Compatibility with <code>string_span</code> and <code>string_view</code>	36
3.4	Review	36
4	Implementation CCGLator	37
4.1	ES.75: Avoid <code>do</code> -statements	38
4.1.1	Checker	38
4.1.2	Quick Fix	38
4.2	ES.76: Avoid <code>goto</code>	40
4.2.1	Checker	40
4.2.2	Goto Usage Pattern Analyser (ES76GotoUsagePattern) . . .	40
	If and Loop Behaviour	40
	Break Behaviour	42
	Multi-Break Behaviour	43
4.2.3	Quick Fix	43
	Use Normal If-Statement	44
	Use While Loop	45
	Use Simple Break	47
	Use Surrounding Lambda and Return	47
4.3	ES.78: Always end a non-empty case with a break	49
4.3.1	Checker	49
4.3.2	Quick Fix	49
	Is Applicable?	49
	Modifying the AST	50
4.3.3	JUnit Tests	50
4.3.4	CDT Bug 514684 - ASTWriter's StatementWriter does not write Attributes for some Nodes like IASTForStatement . .	50
4.3.5	StandardAttributes Class	51
4.4	ES.9: Avoid ALL_CAPS names	52
4.4.1	Checker	52
4.4.2	Quick Fix	52
4.4.3	Ignore Attributes Issue (Matcher & Quick Fix)	52
4.5	ES.50: Don't cast away <code>const</code>	54
4.5.1	Checker	54
4.5.2	Quick Fix	56
	Is Applicable?	56
	Label	57
	Modifying the AST	57
4.6	ES.49: If must use a cast, use a named cast	60
4.6.1	<code>const_cast</code> in Checker & Quick Fix	60
4.6.2	<code>dynamic_cast</code> Quick Fix	60
4.6.3	<code>reinterpret_cast</code> Quick Fix	60

4.7	ASTHelper	61
4.7.1	findNames, getFunctionDeclaratorFromName, getFunctionDeclaratorFromNameInSameTU and getFunctionDeclaratorFromNameViaIndex	61
4.7.2	isInMacro	61
4.7.3	namesEqual	61
4.7.4	Other Added or Modified Helper Functions	62
4.8	Other Changes	63
4.8.1	Handling Markers at Nodes in Macros	63
4.8.2	Not Expanding Macros in Reused Nodes	63
4.8.3	Changing the Ignore Attribute	64
4.8.4	Defining a Marker Type	64
4.8.5	Lock Index	64
4.8.6	Merging Problem-IDs	64
	Problem Preferences	65
	isApplicable	65
	Future Work	65
4.8.7	Making the Ignore Attribute Matcher & Quick Fix more Generic	65
4.9	Testing	66
4.9.1	Checker	66
4.9.2	Quick Fix	67
4.9.3	To Do's	67
	Testing with Different Problem Preferences	67
	Support for Testing "IsApplicable"	68
	Testing a Quick Fix Where There Are Always Multiple Mark- ers	68
5	Implementation GslAtrPtr	69
5.1	Pointer and Size Parameter	69
5.1.1	Checker	69
5.1.2	Quick Fix	69
5.1.3	Checking C++ Version	71
5.1.4	Testing Framework Compatibility	71
5.2	Suppression of Warnings via Attribute	72
5.2.1	Ignore Attribute	72
5.2.2	Ignore Comment	73
5.2.3	The Problem with Static Generators	73
5.3	Improve Span Refactoring	74
5.3.1	Additions to the Checker	74
5.3.2	Additions to the Quick Fixes	74

6 Conclusion	76
6.1 CCGLator Results	76
6.2 GslAtrPtr Results	76
6.3 Future Work	76
6.3.1 Merge Problem-ID's	76
6.3.2 Cross File Changes	77
6.3.3 ES.46 Runtime	77
A Project organisation	I
A.1 Approach	I
A.2 Project Plan	I
A.3 Project Management Environment	II
A.4 This Document	II
B User Manual	III
B.1 Installation	III
B.2 Configuration	IV
B.3 Usage	VI
C Developer Manual	VII
C.1 Prerequisite	VII
C.2 Setting up the Eclipse Workspace	VIII
C.3 Coding	IX
C.3.1 plugin.xml	IX
C.3.2 Checkers, Visitors and Quick Fixes	IX
C.3.3 ASTHelper, ASTFactory	IX
C.3.4 Testing	IX
C.4 Maven	IX
C.5 Continuous Integration Server	X
Glossary	XI
Bibliography	XIII

1 Introduction

Bjarne Stroustrup and Herb Sutter released the C++ Core Guidelines document [SS15] at cppcon 2015. In it they describe several sets of rules which enforce the use of modern C++, improve the quality of code and avoid resource leaks resulting in a simpler and safer subset of the language.

In this project plug-ins are extended and improved which check and fix these rules allowing the developers to integrate these guidelines into their own infrastructure with minimal effort.

1.1 Previous Work

This bachelor thesis uses two previous projects as a foundation to further extend the support for the C++ Core Guidelines.

1.1.1 CCGLator

In the spring of 2016 Özhan Kaya and Kevin Schmidiger developed as their bachelor thesis [zKS16] a plug-in for the Integrated Development Environment Cevelop named CCGLator, which enforces some of these rules and offers quick fixes to adhere to these rules. In the fall of 2016 we enhanced the plug-in [BD16] with new rules and improvements to the code.

1.1.2 CharWars and GslAtorPtr

CharWars is a plug-in for Cevelop as well, developed as a bachelor thesis in 2014 [SG14] which handles C-Strings and their calls with `std::string` operations.

In the spring of 2016 the bachelor thesis GslAtorPtr [GM16] from Elias Geisseler and Philipp Meier added new features to CharWars to handle pointers in code which are based on the C++ Core Guidelines.

1.2 Eclipse CDT

Eclipse CDT [Fou17c] is a fully functional IDE for C/C++ based on the Eclipse framework. As is usual with the Eclipse environment, plug-ins can be easily programmed to extend the functionality and support new features.

1.2.1 Cevalop

Cevalop [fSR16b] is an enhanced version of the Eclipse CDT, released by the Institute for Software (IFS) [fSR16c]. It implements a variety of new plug-ins supplementing the IDE.

1.2.2 Codan

Codan is a part of Eclipse CDT which handles the code analysis part of CDT. It offers checks for usual C/C++ problems such as: "no return value" or "statement has no effect". Apart from it's own checks, it also offers an API to extend the default problems with new, self defined ones. From this interface the plug-ins extend and build up their own checkers and quick fixes.

1.3 Software Stack

The software stack in figure 1 displays the dependencies for this bachelor thesis.

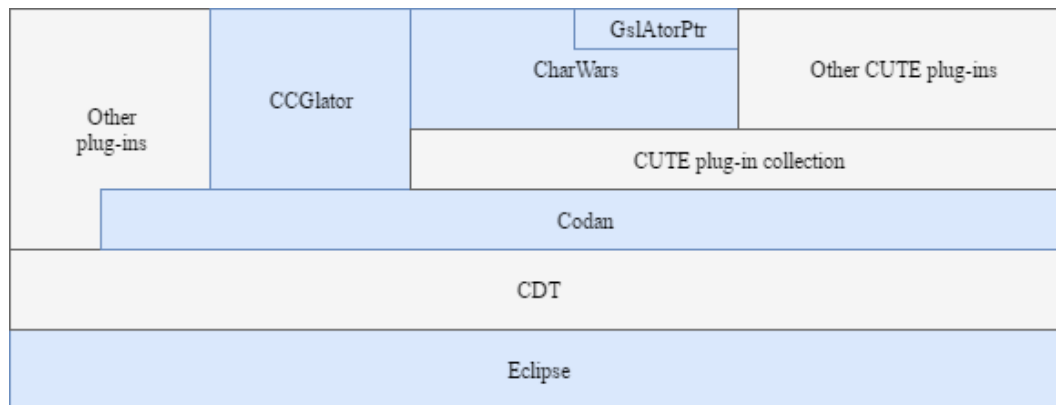


Figure 1: Software stack for this project.

1.4 Scope Definition

For this bachelor thesis we plan to improve the existing plug-ins CCGLator, which we already worked on for our term project, as well as CharWars and GslA-torPtr.

1.4.1 Minimal Scope

For the Minimal scope requirements the following should be done:

- CCGLator
 - Reduce the amount of false positives by testing the plug-in with real world projects.
 - Implement new rules:
 - * ES.75: Avoid do-statements
 - * ES.76: Avoid goto
 - * ES.78: Always end a non-empty case with a break
- GslA-torPtr
 - Checks and refactorings for `string_span<>`

1.4.2 Optimal Scope

For the optimal scope these additional tasks should be done:

- CCGLator
 - Use `isApplicable` to avoid multiple problems for the same rule
 - Improve ES.49:
Try to automatically recognise the appropriate cast function
 - Implement new rules:
 - * ES.9: Avoid ALL_CAPS names
 - * ES.50: Don't cast away const
- GslA-torPtr
 - Suppression of warnings via attributes

1.4.3 Maximum Scope

If the project goes better than expected the following additional tasks can be attempted:

- CCGLator
 - Refactor away the big Switch Case of the ES.46 checker
 - Implement new rules:
 - * Enum.3: Prefer class enums over "plain" enums
 - * Enum.5: Don't use ALL_CAPS for enumerators
- GslAtrPtr
 - Improvement of the `span<T>` refactoring so it can handle more diverse function interfaces.
 - Quick assist for `span<T>` in addition to the quick fix.

1.5 Results

Following items have to be delivered at the end of this project:

- Documentation of the changes to the two plug-ins.
- Plug-in CCGLator with the newly added features and changes.
- Plug-in CharWars with the newly added features and changes.
- Video which gives an overview of the plug-ins.
- Poster with a visualisation of the project.

2 Analysis CCGLator

In this chapter the analysis for the CCGLator plug-in is done. For this an overview of the C++ Core Guideline Rules [SS15] and then a detailed analysis for each implemented rule in the scope of this project is given. The analysis consists of an explanation of the rule, a way to enforce the rule and a part where an example code is shown before and after the quick fix is applied.

2.1 C++ Core Guidelines

The C++ Core Guidelines are split into several sections, each covering different parts of C++. The following sections are available:

- P: Philosophy
- I: Interfaces
- F: Functions
- C: Classes and class hierarchies
- Enum: Enumerations
- R: Resource management
- ES: Expressions and statements
- E: Error handling
- Con: Constants and immutability
- T: Templates and generic programming
- CP: Concurrency
- SL: The Standard library
- SF: Source files
- CPL: C-style programming

All the rules in this project are either part of "C: Classes and class hierarchies" or "ES: Expressions and statements". Each rule in the guidelines document then is divided into several sub chapters detailing the reason behind the rule, examples of code which violate the rule, and an enforcement discussing the approach for a static analysis tool. Furthermore, if the rule is still the object of discussions a chapter for discussion is present. For this project the enforcement part was especially helpful because it gives an introduction on how to implement the rule.

2.1.1 GSL: Guideline Support Library

The Guideline Support Library (GSL) is a C++ library which is needed to properly comply to some of the Guidelines.

The Guideline Support Library (GSL) contains functions and types that are suggested for use by the C++ Core Guidelines maintained by the Standard C++ Foundation. [...]

The library includes types like `span<T>`, `string_span<>`, `owner<>` and others.

Microsoft, *GSL: Guideline Support Library* [Mic16]

2.2 Testing on Real World Application

During our work on the term project, we planned to test the plug-in on a open source software to see how it reacts with bigger codebases. However, not enough time was left to complete this task in a sensible manner. So we decided to do it in our bachelor thesis. For testing purposes the Fish-Shell project was used. The github project [FS17] contains around 40'000 lines of code in C++, in which our plug-in found 1311 violations.

Rule	Found Issues
C.83: For value-like types, consider providing a <code>noexcept</code> swap function	93
C.84: A swap function may not fail	0
C.85: If a user defined swap member function is used, namespace-level <code>swap(a, b)</code> should be overwritten	0
C.164: Avoid conversion operators	0
ES.26: Don't use a variable for two unrelated purposes	180
ES.46: Avoid integer to Char conversions	1
ES.46: Avoid narrowing integer/char conversions	17
ES.46: Avoid narrowing integer/char Function argument conversions	17
ES.46: Avoid signed to unsigned conversions	6
ES.46: Avoid signed to unsigned Function Argument conversion	19
ES.49: If you must use a cast, use a named cast	887
ES.74: Prefer to declare a loop variable in the initializer part of a for-statement	91
Total	1311

During the execution of the code analysis it became apparent that some performance issues were present in the plug-in. We analysed the time consumed per checker and came to the conclusion that C.20 and ES.46 are taking the major parts of the runtime. A detailed overview of the runtimes before and after the improvements to the checker, which are mentioned in the following chapters, can be seen in figure 2.

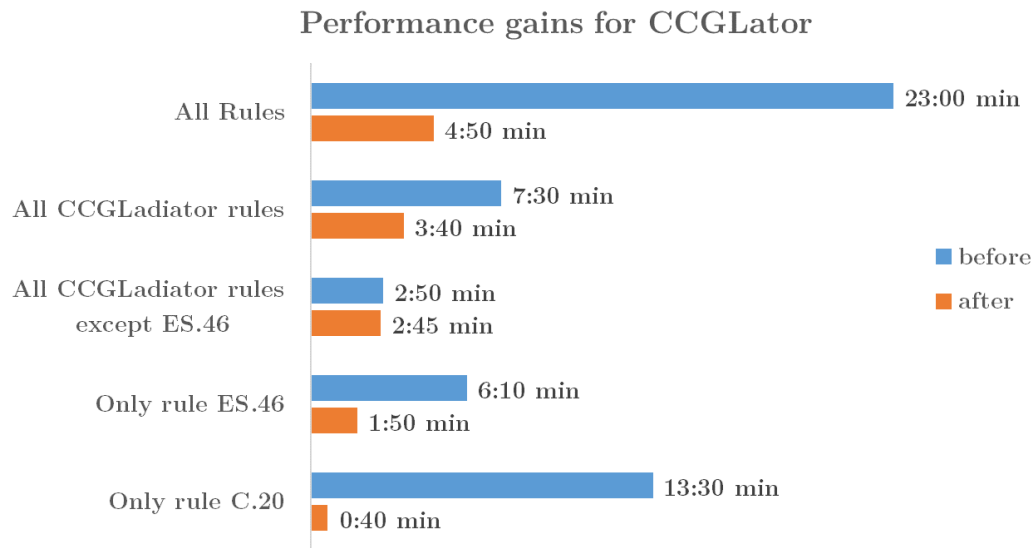


Figure 2: Runtime before and after the performance improvements

2.2.1 C.83: For value-like types, consider providing a noexcept swap function

A lot of value-like classes got marked without swap functions. No false positives were found.

2.2.2 C.84: A swap function may not fail

No violations found in the project because no swap functions are defined.

2.2.3 C.85: If a user defined swap member function is used, namespace-level swap(a, b) should be overwritten

No violations found in the project because no swap functions are defined.

2.2.4 C.164: Avoid conversion operators

No violations found in the project because no conversion operators are defined.

2.2.5 ES.26: Don't use a variable for two unrelated purposes

During testing an apparent problem of ES.26 came to light. A code snippet like the one in listing 1 was marked with a warning:

```
1 void function() {  
2     bool inserted = true;  
3     if (!inserted) { //ES.26 Error showed up  
4     }  
5 }
```

Listing 1: ES.26: Not-operator problem

The problem was that `!inserted` is represented as an `IASTUnaryExpression` in the AST. However, increment and decrement operations are represented by the same node and the check only looked if an `IASTUnaryExpression` is present and increased the usage counter of the variable. For a fix the operator type from these nodes are checked and only if they are either `++` or `--` the usage is highlighted.

Another issue was the handling of variables inside a switch statement. It was not possible to assign a new value to a variable in multiple switch-cases as in listing 2.

```
1 void function() {  
2     int i;  
3     switch(1) {  
4         case 1 : i = 1; //ES.26 marked variable i;  
5         break;  
6         case 2 : i = 2; //ES.26 marked variable i;  
7         break;  
8     }  
9 }
```

Listing 2: ES.26: Switch problem

Luckily the check if the variable is reassigned inside an if-statement is based on the same logic and could be reused for the switch.

Another issue encountered during testing were loops implemented like in the example in listing 3.

```
1 void function() {  
2     int i = 0;  
3     while(i < 10) {  
4         i++; //ES.26 Error showed up  
5     }  
6     int a = i;  
7 }
```

Listing 3: ES.26: While-loop problem

The marker on line 4 should not be present as this is a valid form of a loop. To counteract this behaviour the counter for allowed usages, in this case the initialization and incrementation of the variable, had to be increased to 2.

Additionally, the performance of the checker was improved because as a first statement it checks if the node is from the type `IASTSimpleDeclaration`. If not the program continues.

2.2.6 ES.46: Avoid lossy (narrowing, truncating) arithmetic conversions

The checks for this rule took about twenty times longer to finish than most other checks. When analysing which code part takes how long, two code parts were using most of the time.

The problematic code was finding the function declarator based on a name. This has two parts which both could be improved.

One part was finding the declarator if it is inside the same translation unit (the same file). This was done via a recursive search through the whole AST to get all function declarators, which were then each checked if it has the name of the one we search. This process was responsible for two thirds of the checker's runtime.

The translation unit (the root of the AST) however already provides a way to get a list of names in declarations based on a binding which is way faster. Compare listings 4 and 5.

```

1 public <T extends IASTNode> List<T> findNodeTypes(IASTNode node, Class<T> type) {
2     List<T> list = new ArrayList<T>(); // simplified: created only if needed
3     for (IASTNode child : node.getChildren()) // simplified: additional checks
4         list.addAll(findNodeTypes(child, type));
5     if (type.isInstance(node))
6         list.add(type.cast(node));
7     return list;
8 }
9 // ...
10 private IASTFunctionDeclarator getFunctionDeclaratorFromNameInSameTU(IASTName name
11     ) {
12     List<IASTFunctionDeclarator> funcdecllist =
13         findNodeTypes(name.getTranslationUnit(), IASTFunctionDeclarator.class);
14     for (IASTFunctionDeclarator fdecl : funcdecllist) {
15         if (fdecl.getName().equals(name))
16             return fdecl;
17     }
18 }

```

Listing 4: Old slow way of getting the Function Declarator (simplified code)

```

1 private IASTFunctionDeclarator getFunctionDeclaratorFromNameInSameTU(IASTName name
2     ) {
3     IName[] declarations = name.getTranslationUnit()
4         .getDeclarations(name.resolveBinding()); // method provided by CDT
5     if (declarations.length==1) // simplified: additional checks
6         return (IASTFunctionDeclarator)((IASTName)declarations[0]).getParent();
7 }

```

Listing 5: Optimized way of getting the Function Declarator (simplified code)

The other part was finding the declarator if it is in another file. This used the index to find the names in other files and parsed found files.

This checker has often to check function declarations to get the parameter types which are often in other files. But for every lookup the file was parsed anew which is a slow process. Figure 3 visualises this process.

Therefore, we introduced the option of a cache, which this checker now uses. The cache is stored in the checker's visitor object which gets re-instantiated for each run and file, so there should not be any issues with outdated caches.

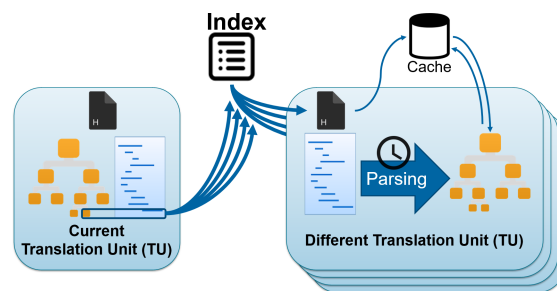


Figure 3: Using the Index and reparsing repeatedly is slow without the cache

2.2.7 ES.49: If you must use a cast, use a named cast

At first, we thought there were lots of unexpected false positives, because there were lots of code parts highlighted where there were seemingly no casts used.

When looking at the AST of the highlighted code there were way more nodes including the found casts. The reason for this was the following. Macros get expanded before parsing and the code parts and nodes they stand for get inserted in their place. Because of this, used casts inside macros were not marked at the define statement but everywhere where the macro was used.

This is a general issue that affects all checkers and through this rule it got apparent that it can be confusing. Therefore we modified our code to not mark such nodes inside macro expansions. (See also chapters 4.7.2 `isInMacro` and 4.8.1 Handling Markers at Nodes in Macros)

2.2.8 ES.74: Prefer to declare a loop variable in the initializer part of a for-statement

In the test project the rule was highlighted several times which was either because the initializer part was empty or the loop variable was initialised outside of the loop. No false positives were found.

2.2.9 C.20: If you can avoid defining default operations, do

During testing our checkers we noticed that the checker for the rule C.20, implemented by a previous group, took more than 13 minutes to finish, which is way too long. We quickly found out that the code to get the definition from a declared function takes most of the time.

Said code part uses the index to find other occurrences of the name and parses files where it is found for further checking. This is an expensive process but should not result in such an extreme long runtime. What was confusing was that they used the `findNames`-flag `FIND_ALL_OCCURRENCES` instead of the `FIND_DEFINITIONS` flag and that it stopped working when changed.

That is until we realized that they searched for the class name and not for the function name. Removing the detour over the class-name reduced the completion time to less than a minute.

In addition, while analysing the checker's source code, we found a slight oversight which resulted in some false positives. The function `"isFunctionDefinitionNecessary"`

ignored any member initializer lists which resulted in some constructors being flagged as not necessary (for example the one in listing 6).

```
1 struct options {  
2     bool a, b, c;  
3     options(): a(true), b(true), c(true) {}  
4 };
```

Listing 6: C.20: False Positive with initializer list

2.2.10 Quick Fixes

Whilst the checkers are easily automated to execute for the whole project, all the found markers and their corresponding quick fixes can only be executed by hand. For every rule several occurrences were manually fixed and the resulting code was analysed for breaking changes. The code was still runnable and the test, if present, were successful.

2.3 ES.75: Avoid do-statements

For better readability, and to avoid overlooking the while-statement, do-statements should be avoided. For while-statements which should always run at least once, make this explicit with accordingly modified while-statements.

2.3.1 Enforcement

The checker for this rule is straight forward. Any IASTDoStatement node can be flagged.

The quick fix can not just change it to a normal while because it would not make a default first run anymore. But a quick fix should behave like a refactoring and not change behaviour. Therefore, we need to retain that behaviour. To achieve that we create a new boolean variable set to true, add that to the beginning of the while condition joined to the original condition with a logical or, and directly set it to false once inside the while body.

The order of the conditions and the statement inside the while-body is important. If the condition has a side effect we should not start executing that before the first run, to preserve the behaviour of the do-while loop. Additionally, the value has to be set to false as early as possible. The following example (listings 7 & 8) shows that things break if the ordering is wrong:

```
1 int condition = 10;
2
3 do {
4     if(!enableDebugOutput) continue;
5     std::cout << condition << std::endl;
6
7 } while(condition-- > 0);
```

Listing 7: ES.75 pre wrong quick fix

```
1 int condition = 10;
2 bool firstRun = true;
3 while ((condition-- > 0) || firstRun) {
4     if (!enableDebugOutput) continue;
5     std::cout << condition << std::endl;
6     firstRun = false;
7 }
```

Listing 8: ES.75 post wrong quick fix

This "fixed" code decrements the condition value too early compared to the original code and if enableDebugOutput is false it results in an endless loop, because firstRun is never set to false.

2.3.2 Pre fix Code

```
1
2 do {
3
4     doSomething();
5 } while(someCheck());
```

Listing 9: ES.75 pre fix

2.3.3 Post fix Code

```
1 bool firstRun = true;
2 while(firstRun || (someCheck())) {
3     firstRun = false;
4     doSomething();
5 }
```

Listing 10: ES.75 post fix

2.4 ES.76: Avoid goto

[...] our intellectual powers are rather geared to master static relations and [...] our powers to visualize[sic] processes evolving in time are relatively poorly developed. For that reason we should do [...] our utmost best to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence [...] as trivial as possible.

Edsger W. Dijkstra, *Go-to statement considered harmful* [Dij68]

Jumping around the code with goto-statements makes the code hard to understand and might result in errors. Especially if used as a substitute for while loops and if-then-else conditionals as stated by Ken Bloom [Blo10]

Breaking out of multiple convoluted loops is often referenced as one "still valid" use case for goto as seen in bta's [bta10] or shsteimer's [s⁺09] comments to questions on this topic. See listing 11 for an example.

```
1 while(someCondition) {
2     while(someOtherCondition) {
3         // some code
4         if(exitCondition) goto exit;
5     }
6 }
7 exit:
8 //more code
```

Listing 11: Valid usage of goto

But even this should be avoided if possible. For example with a lambda expression and a return statement. If used, the goto label should be directly after an outer loop.

Another potentially valid use of goto might be using it "as an alternative to exception handling" as stated by Rob Walker [Wal08] (see listing 12).

```
1 if(condition1) goto error_cleanup;
2 if(condition2) goto error_cleanup;
3 return;
4 error_cleanup:
5 //more code
```

Listing 12: Using Goto as an alternative to exception handling

But this is not really a special pattern of using goto, and if a developer really wants to use such code he can just use ignore attributes to ignore the checker in such code.

2.4.1 Enforcement

Finding goto-statements (IASTGotoStatement) is easy. But if we want to provide quick fixes we have to analyse the surrounding code a bit more.

There are 4 simple usage patterns of goto which we can recognise.

- Skipping a code part on some condition
→ use normal if.
- Loop back to an earlier code part on some condition
→ refactor like do-while in ES.75 (see chapter 2.3.1)
- Breaking out of one level of a loop
→ use break instead.
- Breaking out of multiple levels of loops
→ use a surrounding lambda and a return statement.

But in any of these cases we can only remove the label if it is not used by another goto.

In the first two cases we do not only replace the goto-statement, we also modify the surrounding if-statement and move the statements between the if-statement and the label around. Therefore, we have to look at this in a bit more detail.

Skipping a Codepart on Some Condition

In the simplest case the goto-statement is directly inside an if-statement, is the only statement executed, and there is no "else"-clause: `if(condition) goto label;`

In such cases the quick fix should look like this: (Listings 13 & 14)

```
1 if(condition) goto end;
2 // optional code
3 end:
4 // ...
```

Listing 13: ES.76 "goto if" (pre fix)

```
1 if(!condition) {
2   // optional code
3 }
4 // ...
```

Listing 14: ES.76 "goto if" (post fix)

But we should know how to handle other forms of if-statement and goto combinations. If we look at the following code samples (listings 15 & 16) we notice that any statement after the goto would be unreachable. (Except if there was another label but in such cases we do not provide a quick fix.) Statement `C()`; can just be moved inside the clause in which the goto is not located, and the goto-statement can be dropped.

```

1 if(condition) {
2   A();
3   goto label;
4   // code here would be unreachable
5 } else {
6   B();
7   // move C(); here
8 }
9 C();
10 label:
11 D();

```

Listing 15: ES.76 if then goto analysis

```

1 if(condition) {
2   B();
3   // move C(); here
4 } else {
5   A();
6   goto label;
7   // code here would be unreachable
8 }
9 C();
10 label:
11 D();

```

Listing 16: ES.76 else goto analysis

Of course, `A();`, `B();` and `C();` could be more than one statement or no statement at all. In case the "then"-clause would get empty we can negate the condition and use the "else"-clause as a replacement for the "then"-clause to simplify the resulting code. To negate the condition we have to put it in an `IASTUnaryExpression` of type `op_bracketedPrimary` and that in an `IASTUnaryExpression` of type `op_not`. The bracketed expression is not necessary if it is only a single `IASTIdExpression`.

What about `else if`? If we look at the code from listing 15, the code in the "else"-clause (`B();`) does not have to be inside a compound-statement (curly brackets) but could come directly after the `else`. That is exactly how the `if` part of an `else if` is located in the AST. Therefore we can handle it like any other statement. However, if we would try providing a quick fix for goto-statements inside an `else if`, it would get complicated really quickly. Such gotos we just mark but we do not provide a quick fix for them.

Loop Back to an Earlier Code Part on Some Condition

Here, the simplest cases are again if-statements that only have a goto-statement in the "then"-clause like this: `if(condition) goto label;`

So if we look at some simple example code we could provide a quick fix like in listings 17, 18 & 19:

```

1
2 loop:
3
4 // repeating code
5 if(condition) goto loop;

```

Listing 17: ES.76 "goto loop" (pre fix)

```

1
2 do {
3
4   // repeating code
5 } while(condition);

```

Listing 18: ES.76 "goto loop" (intermediate step)

```

1 bool firstRun = true;
2 while (firstRun ||
3   condition){
4   firstRun = false;
5   // repeating code
6 }

```

Listing 19: ES.76 "goto loop" (post fix)

As mentioned above, the code from the first step of refactoring (see listing 18) conflicts with rule ES.75 (see chapter 2.3) which is why we end up with the code from listing 19.

Now, what if we have more complicated if-statements? If we have an additional "else"-clause (see `C()`; in listing 20), that is simply executed once we stop looping, and can be put after the loop in refactored code.

But additional code in the "then"-clause quickly gets complicated. Like before, we consider code in the "then"-clause after the `goto` unreachable and do not provide a quick fix. But lets look at `B()`; in listing 20. That code gets executed after checking the condition but before looping. Or alternatively it gets executed before `A()`; but not the first time.

This results in code like in listings 21 & 22. The code in listing 21, if even possible, is ugly and gets unreadable really quickly. (Consider more than one statement instead of `B()`;) Likewise, the code from listing 22 would complicate the quick fix quite a bit and would require more analysis. An open question is for example: "Would the code still do the same in combination with side effects?"

```

1 loop:
2 A();
3 if(condition) {
4     B();
5     goto loop;
6     // code here would be unreachable
7 } else {
8     C();
9 }

```

Listing 20: ES.76 if goto loop analysis (pre fix)

```

1
2 do {
3
4
5     A();
6 } while(condition && B());
7 C();

```

Listing 21: ES.76 if goto loop analysis (post fix candidate 1)

```

1 firstRun = true;
2 while(firstRun || condition) {
3     if(!firstRun) B();
4     firstRun = false;
5     A();
6 }
7 C();

```

Listing 22: ES.76 if goto loop analysis (post fix candidate 2)

A quick look at `goto` loops where the `goto` is in the else statement, results in the following potential quick fix (see listings 23 & 24).

```
1 loop:
2 A();
3 if(condition) {
4   B();
5 } else {
6   C();
7   goto loop;
8   // code here would be unreachable
9 }
```

Listing 23: ES.76 else goto loop analysis (pre fix)

```
1 do {
2   A();
3   if(condition) {
4     B();
5   } else {
6     C();
7   }
8 } while(!condition);
```

Listing 24: ES.76 else goto loop analysis (post fix candidate)

This quick fix candidate is more likely to be added in the future than the previous, but for now we do not think it is worth the time to support this quick fix.

Conclusion

In conclusion, we have the following additional constraints for supporting a quick fix, for the following usage patterns from the beginning of this chapter:

- Skipping a codepart on some condition
→ Support quick fix only if `goto` is the last statement in the "then"- or "else"-clause
- Loop back to an earlier code part on some condition
→ Support quick fix only if `goto` is the only statement in the "then"-clause

2.4.2 Pre fix Code

```

1 // "Skipping code on some condition"
2 if(someCondition) goto end;
3 // optional code
4 end:
5 // ...
6 // Additionally, see listings 15 & 16
7
8
9 // "Loop back ... on some condition"
10
11 loop:
12 // repeating code
13 if(someCondition) goto loop;
14
15
16
17 // "Breaking out of one level of a loop"
18 while(someCondition) {
19 // ...
20 if(someOtherCondition) goto exit;
21 // ...
22 }
23 exit:
24 // ...
25
26 // "Breaking out of multiple levels of
  loops"
27
28 while(condition1) {
29 while(condition2) {
30 // ...
31 if(condition3) goto exit2;
32 // ...
33 }
34 }
35 exit2:
36 // ...

```

Listing 25: ES.76 pre fix

2.4.3 Post fix Code

```

1 // "Skipping code on some condition"
2 if(!(someCondition)) {
3 // optional code
4 }
5 // ...
6 // Additionally, see listings 15 & 16
7
8
9 // "Loop back ... on some condition"
10 bool firstRun = true;
11 while(firstRun || (someCondition)) {
12 firstRun = false;
13 // repeating code
14 }
15
16
17 // "Breaking out of one level of a loop"
18 while(someCondition) {
19 // ...
20 if(someOtherCondition) break;
21 // ...
22 }
23
24 // ...
25
26 // "Breaking out of multiple levels of
  loops"
27 [&] {
28 while(condition1) {
29 while(condition2) {
30 // ...
31 if(condition3) return;
32 // ...
33 }
34 }
35 }();
36 // ...

```

Listing 26: ES.76 post fix

2.5 ES.78: Always end a non-empty case with a break

During the analysis of this rule we quickly noticed that such a checker is already implemented in CDT's [Fou17c] static analysis tool "codan" [Fou17b]. Codan provides quick fixes to add a break-statement or to add a suppressing comment to ignore an occurrence.

But we wanted support for explicit "[[fallthrough]];" statements introduced in C++17 [Fou17a, "Fallthrough attribute" [dcl.attr.fallthrough]].

We considered implementing the rule completely new and disabling the corresponding codan rule with our plug-in. But we decided to go for the much cleaner option of changing codan's checker directly as well as adding a quick fix for "[[fallthrough]];" statements to codan.

2.5.1 Enforcement

To recognise explicit "[[fallthrough]];" statements, we have to check if the last statement before another "case"- or "default"-label is an IASTNullStatement, and if that "null"-statement has the attribute "fallthrough". See figure 4 for an example AST and the corresponding code in listing 27.

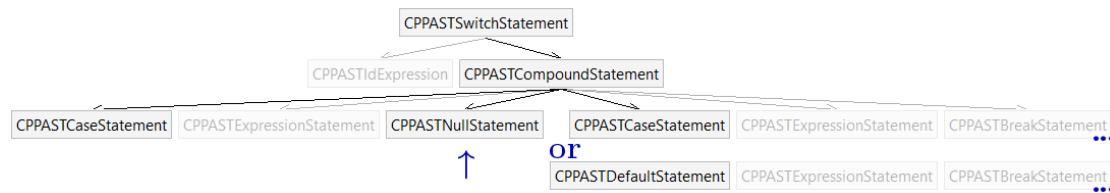


Figure 4: AST Nodes used to recognise [[fallthrough]] in ES.78

```

1 switch (i) {
2 case 1:
3     doSomething();
4     [[fallthrough]]; // ←
5 case 1: // OR default:
6     doSomething();
7     break;
8 }

```

Listing 27: Code for AST in figure 4

It may be the last statement inside a compound-statement (which itself is the last statement before another "case"- or "default"-label), because then the following requirement by the standard is still true:

The next statement that would be executed after a fallthrough statement shall be a labeled statement whose label is a case label or default label for the same switch statement.

Working Draft, Standard for Programming Language C++ [Fou17a, "Fallthrough attribute" [dcl.attr.fallthrough]]

2.5.2 Pre fix Code

```
1 void doSomething() {}
2 void foo(int i) {
3     switch(i) {
4         case 1:
5             doSomething();
6
7         case 2:
8             doSomething();
9             break;
10    default:
11        doSomething();
12    }
13 }
```

Listing 28: ES.78 pre fix

2.5.3 Post fix Code

```
1 void doSomething() {}
2 void foo(int i) {
3     switch(i) {
4         case 1:
5             doSomething();
6             [[fallthrough]];
7         case 2:
8             doSomething();
9             break;
10    default:
11        doSomething();
12    }
13 }
```

Listing 29: ES.78 post fix

2.6 ES.9: Avoid ALL_CAPS names

As the title of the rule says, one should not use any names without lower-case letters.

Reason Such names are commonly used for macros. Thus, ALL_CAPS name[s] are vulnerable to unintended macro substitution.

C++ Core Guidelines [SS15, ES.9]

Optionally, for code still using macros, we could mark non-ALL-CAPS macro names.

2.6.1 Enforcement

We can mark every IASTName, if it is a Declaration, all in CAPS, and not a single letter template parameter.

If enabled in the preferences we can loop through the macros of every IASTTranslationUnit, and mark any macros which have a name not in all CAPS.

A special quick fix is not needed. The option to rename is already provided on any name by default.

2.6.2 Problematic Code

```
1 #define NE !=
2 // ...
3 enum Direction { N, NE, NW, S, SE, SW, E, W }; // Syntax ERROR
4 void foo(Direction dir) {}
5 void bar() {
6     foo(Direction::NE);
7 }
```

Listing 30: ES.9 Example Code

2.7 ES.50: Don't cast away const

The reason mentioned by the guideline puts it quite simple:

Reason It makes a lie out of const. If the variable is actually declared const, the result of “casting away const” is undefined behavior.

C++ Core Guidelines [SS15, ES.50]

When using `const_cast` we either have a variable declared as `const` which is used in a non-`const` way and the `const` is a lie. Or else we use `const_cast` where it is not needed and the cast is a lie.

2.7.1 Enforcement

We can just mark any `IASTCastExpression` with the operator type `op_const_cast`.

But if we stop here we overlook quite a lot of `const` casts. Additionally, we have to mark `IASTCastExpressions` if it is a C-style cast (`op_cast`) and if it casts away `const`. And finally we have to mark C-style casts if the surrounding function is `const` and the operand is a member variable of the functions class.

As quick fixes we can suggest removing the cast and the `const` property from the variable. Or if the cast is in a `const` function we can provide a quick fix to either remove `const` from the function or set the (member-)variable to mutable and, in both cases, remove the `const` cast.

2.7.2 Pre fix Code

```
1 void foo() {  
2     int i = 20;  
3     int const * ic = &i;  
4     int * inew = const_cast<int *>(ic);  
5     *inew = 10;  
6 }  
7  
8 class ValWithCache {  
9 public:  
10     int getVal() const {  
11         const_cast<ValWithCache*>(cache)  
12             .set(val);  
13         return val;  
14     }  
15     void set(int x) {  
16         val = x;  
17     }  
18 private:  
19     int val = 0;  
20     ValWithCache cache;  
21 };
```

Listing 31: ES.50 pre fix

2.7.3 Post fix Code

```
1 void foo() {  
2     int i = 20;  
3     int * ic = &i;  
4     int * inew = ic;  
5     *inew = 10;  
6 }  
7  
8 class ValWithCache {  
9 public:  
10     int getVal() const {  
11         cache  
12             .set(val);  
13         return val;  
14     }  
15     void set(int x) {  
16         val = x;  
17     }  
18 private:  
19     int val = 0;  
20     mutable ValWithCache cache;  
21 };
```

Listing 32: ES.50 post fix

3 Analysis GslAtorPtr

In this section the analysis for the GslAtorPtr plug-in is done. A lot of understanding was gained by reading the documentation for CharWars [SG14] and GslAtorPtr [GM16]. The plug-in's architecture is very similar to CCGLator helping a lot to understand it.

3.1 string_span

In the future work chapter of GslAtorPtr [GM16] they discuss the implementation of the `string_span<>` type from the Guideline Support Library [Mic16]. In this project we decided to add this into GslAtorPtr. A `string_span<>` can be used as seen in listing 33 which is a simple hello world application.

```

1 #include <iostream>
2 #include "gsl.h"
3
4 void printout(gsl::cstring_span<> string) {
5     std::cout << gsl::to_string(string) << std::endl;
6 }
7
8 int main() {
9     gsl::cstring_span<> string = gsl::ensure_z("Hello world");
10    printout(string);
11 }

```

Listing 33: Example of a `string_span`

This example shows one of the main benefits of a `string_span<>`. The function only takes one argument and does not need to receive an additional parameter for the size of the string. Information like this is wrapped inside the `string_span<>` type.

3.1.1 Difference Between `span` and `string_span`

The GSL provides a type `span<T>` which should be used instead of a pointer and a size counter. It represents a contiguous range of memory which already implements bound safety. Whilst this type is also applicable for strings, the GSL additionally provides the `string_span` type. This type is based on a one dimensional `span<T>` with added templates and functions helping to handle strings, like the `to_string()` function.

3.1.2 String_span Types

Following types are provided by the GSL:

string_span type	null terminated string_span type	string type
string_span	zstring_span	char*
cstring_span	czstring_span	const char*
wstring_span	wzstring_span	wchar_t*
cwstring_span	cwzstring_span	const wchar_t*

As can be seen in the paper from Neil MacIntosh regarding `string_span<>` [Mac16], a `string_span<>` can represent both a normal string and a null-terminated one whilst the type `zstring_span<>` always ensures that a null-terminated is stored. Normally the standard `span<T>` types are sufficient enough for the usual use cases. Null terminated ones should only be used in the context of converting a null terminated `span<T>` into a legacy string.

3.1.3 Trampoline Function for string_span

As already documented in the `GslAtorPtr` documentation [GM16] in chapter 2.1.3, a trampoline function can be used to refactor a function with parameter combination of pointer and size. In the current state, this always results in a `span` type (listing 34) even if a `string_span` as in listing 35 would be applicable. For this, the plug-in has to be extended to check for the type of the pointer.

```
1 void printNumbers(int* numbers, int length) {
2     std::cout << numbers << std::endl;
3 }
```

Listing 34: `span` should be used

```
1 void printNames(char* string, int length) {
2     std::cout << string << std::endl;
3 }
```

Listing 35: `string_span` should be used

After the type of the pointer is evaluated, an according `string_span` (listing 36) or `span` (listing 37) trampoline function should be implemented.

```

1 void printNames(char* string, int length) {
2     std::cout << string << std::endl;
3 }
4
5 //added trampoline function
6 void printNames(gsl::string_span<> string) {
7     return printNames(string.data(), string.size());
8 }

```

Listing 36: `string_span` trampoline function

```

1 void printNumbers(int* numbers, int length) {
2     std::cout << numbers << std::endl;
3 }
4
5 //added trampoline function
6 void printNumbers(gsl::span<> numbers) {
7     return printNumbers(numbers.data(), numbers.size());
8 }

```

Listing 37: `span` trampoline function

Additionally, the before mentioned `string_span` types have to be considered in the fix for a trampoline function. For example for a `const wchar_t` pointer, the type `cwstring_span` needs to be applied.

3.1.4 String_span Rewrite Quick Fix

`GslAtrPtr` offers an additional quick fix called a rewrite. This fix changes the found function directly to support a `string_span` as well as all the found call sites, as can be seen in listings 38 and 39.

```

1 void printOut(char* string, int size) {
2     for(int i = 0; i < size; i++) {
3         std::cout << string[i];
4     }
5     std::cout << std::endl;
6 }
7
8 int main(){
9     int len { 4 };
10    char string[len] {"test"};
11    printOut(string, len);
12 }

```

Listing 38: Pre rewrite quick fix

```

1 #include "gslrefactor.h"
2 void printOut(gsl::string_span<> string) {
3     for (int i = 0; i < string.size(); i++) {
4         std::cout << string[i];
5     }
6     std::cout << std::endl;
7 }
8
9 int main(){
10     int len { 4 };
11     char string[len] {"test"};
12     printOut(gsl::string_span<> { string, len });
13 }

```

Listing 39: Post rewrite quick fix

The problem with this quick fix however is that the remaining function body still uses the features of a legacy string such as an index based for-loop in this example. With the new `string_span` type this could be refactored to a for-each loop as in listing 40. This would increase the code quality significantly but has to be implemented by the programmer himself because the plug-in can not handle all the possible cases.

```

1 #include "gslrefactor.h"
2 void printOut(gsl::string_span<> string) {
3     for (auto const & element : string) {
4         std::cout << element;
5     }
6     std::cout << std::endl;
7 }
8
9 int main(){
10     int len { 4 };
11     char string[len] {"test"};
12     printOut(gsl::string_span<> { string, len });
13 }

```

Listing 40: Manual improvement to for-each

3.1.5 Fixing C Strings with `string_span`

CharWars already implements the substitution of C-strings with the `string` class from the C++ standard library. Another possible substitution would be the use of a `string_span`. However, a lot of use cases from the CharWars plug-in rely on functions applicable on `std::string` which are either not available, or have a different syntax, for `string_span`. This would result in a substantial change of the already working CharWars plug-in. So we decided against this change.

3.2 string_view

During the research for `string_span<>` the type `string_view` appeared multiple times promising similar benefits as a `string_span<>`. This type already exists for some time in the boost library [Lib17] and is, as of C++17 [Fou17a, `string.view`] , added to the standard library as the type `std::string_view`. Because this plugin already provides `span<T>` and `string_span<>` checkers and fixes it would be beneficial if it also supported the `string_view` type.

3.2.1 What is a string_view?

A `string_view` is comparable to the already analysed `string_span<>` type (see chapter 3.1). Similarly, it holds a pointer and size parameter. However, the biggest difference is that the pointee is constant making `string_view` a read-only access to a string. Additionally, a lot of member functions are provided to use this type efficiently. Amongst them are, iterators, a swap function, capacity functions and a substring function.

3.2.2 string_view Types

Based on the provided string type, different `string_view` types have to be used. Which ones exist can be read out of the table below:

string_view type	string type
<code>string_view</code>	<code>char*</code>
<code>wstring_view</code>	<code>wchar_t*</code>
<code>u16string_view</code>	<code>char16_t*</code>
<code>u32string_view</code>	<code>char32_t*</code>

3.2.3 string_span or string_view

To safely decide if a `string_view` can be used when a pointer and size parameter are found, two requirements have to be met. First the C++ version has to be C++17 or higher. Second the found pointer has to be used in a read-only manner within the function. This can be checked with whether the pointee is constant or not. If it is, it's safe to assume so.

If both of these conditions are met, the same quick fixes as a `string_span<>` can be offered with the minor difference of the used parameter type. For code examples of these fixes look into chapter 3.1.3 and 3.1.4.

3.3 Improvement of the span<T>Refactoring

Another point in the further work chapter of GslAtorPtr [GM16] is the improvement of the span<T> refactorings to handle more diverse function interfaces. In its current state it highlights a function interface as soon as a pointer + size parameter combination is found. Following improvements can be made to the refactoring.

3.3.1 Multiple Pointer and Size Parameter Combinations

One improvement is the handling of multiple occurrences of pointer + size parameters to be refactored in one step. Currently following code (see listing 41) is generated by the plug-in.

```

1 #include "gslrefactor.h"
2 void function(int* a, int size1, int* b, int size2) { //original function
3 }
4
5 void function(gsl::span<int> a, int* b, int size2) { //first quick fix
6     return bimbambozle(a.data(), a.size(), b, size2);
7 }
8
9 void function(gsl::span<int> a, gsl::span<int> b) { //second quick fix
10    return bimbambozle(a, b.data(), b.size());
11 }

```

Listing 41: span refactoring with multiple executions

While the quick fix does not break the code, it would be better if only one function was added which takes two span<T> and has the according call to the original function (see listing 42).

```

1 #include "gslrefactor.h"
2 void function(int* a, int size1, int* b, int size2) {
3 }
4
5 void function(gsl::span<int> a, gsl::span<int> b) { //quick fix
6     return function(a.data(), a.size(), b.data(), b.size());
7 }

```

Listing 42: Desired refactoring for multiple pointer + size combinations

Another way to order the parameters would be, several pointers followed up by an equal amount of size parameters, resulting in a interface like this: (Listing 43)

```

1 void function(int* a, int* b, int size1, int size2) {
2 }

```

Listing 43: Possible ordering of multiple pointer + size combinations

However, no such variant was found in multiple C libraries and code snippets displaying the usage of pointer + size parameters. Because of this, the implementation will not support such an interface, however the marker will still appear for the innermost combination.

3.3.2 Multiple Pointer and One Size Parameter

Another possible improvement is a interface where multiple pointers are followed up by one size parameter representing the size of all the preceding pointers. In the current state only the last pointer would be refactored and all the other ones would be ignored. The following improvement could be added to the quick fix (see listing 44).

```
1 #include "gslrefactor.h"
2 void function(int* a, int* b, int size2) {
3 }
4
5 void function(gsl::span<int> a, gsl::span<int> b) { //quick fix
6     return function(a.data(), b.data(), b.size());
7 }
```

Listing 44: Desired refactoring for multiple pointer + one size combinations

3.3.3 Compatibility with string_span and string_view

These improvements are compatible with the already mentioned addition of `string_span<>` and `string_view`. The quick fix is independent of the used type. However the checker has to report the correct problem when for example a char and an int pointer are used, a `span<T>` quick fix problem should be reported.

3.4 Review

Both plug-ins are improved with new features or better architecture which results in a better user experience and better base for future additions to them. The new features round out the plug-ins but are in no way final. There is still a lot of work to do to support more rules.

4 Implementation CCGLator

As a starting point, the CCGLator plug-in from the CCGLadiator term project [BD16] was used. For an in-depth overview over the plug-in, please refer to its documentation. In this project the following changes and additions to CCGLator were necessary and are explained in this chapter:

- Additions and refactorings to ASTHelper, ASTFactory and more changes
- Checkers with visitors for the newly added rules
- Quick fixes for the newly added rules
- Tests for checkers and quick fixes

In the whole implementation process the go-to resource was the Eclipse CDT API Documentation [Fou11] providing us with the needed information about the CDT environment.

For an overview of the most important classes and files, refer to the developer manual (appendix C).

4.1 ES.75: Avoid do-statements

Implementing this rule was quite straight forward. However, after the first implementation we noticed some minor issues which required us to reevaluate our analysis and make some minor adjustments to end up with the current version.

4.1.1 Checker

The checker visits the IASTStatements and flags them if they are of type IASTDoStatement.

4.1.2 Quick Fix

Providing a quick fix is not too difficult either. We just have to generate a new IASTWhileStatement with slightly modified while-statement and body. To ensure a guaranteed first run of the loop, we make a boolean variable "firstRun", with an appended number if needed. Its declaration gets put in front of the new while-statement [1] (see figure 5 and listing 45), gets added to the beginning of the while-statement condition [2] (using a logical or [3]) and it needs to get changed in the while body[4].

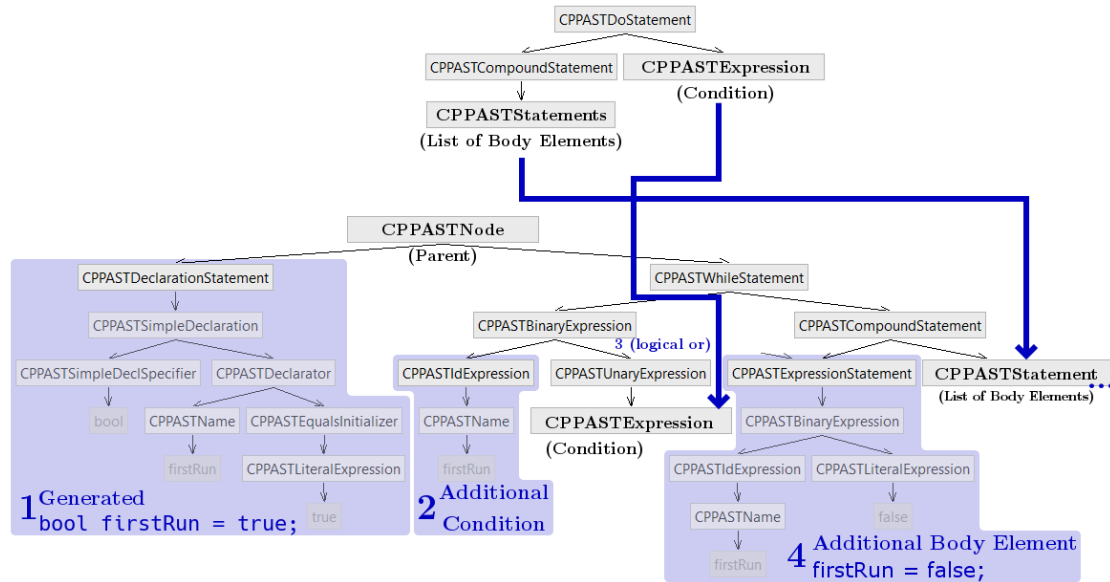


Figure 5: AST nodes used to enforce ES.75

```
1 do {  
2   doSomething();  
3 } while(someCheck());  
4  
5 bool firstRun = true; // [1] Generated  
6 while(firstRun || (someCheck())) { // [2] Additional Condition ("firstRun"),  
                                   // [3] logical or ("||")  
7   firstRun=false; // [4] Additional Body Element  
8   doSomething();  
9 }
```

Listing 45: Code for AST in figure 5

4.2 ES.76: Avoid goto

This rule needed the most iterations until we were happy with the way it works. Partly this was because code with gotos can quickly get complicated and we overlooked some special cases. Another reason was that while fixing some stuff we decided that it would be easy to provide some additional quick fixes.

4.2.1 Checker

The checker checks each IASTStatement if it is a goto-statement and marks them. By fetching the referenced label (IASTLabelStatement) it already tries to detect normal "if behaviour", "loop behaviour" or "break behaviour" and provides more detailed description to the marker accordingly.

Reporting "multi-break behaviour" can be disabled in the preferences, because this could count as the only valid use case of goto. (See chapter 2.4)

4.2.2 Goto Usage Pattern Analyser (ES76GotoUsagePattern)

Because we need to find out which usage pattern fits the goto-statement in the checker as well as the quick fix, we put this in a separate class, which returns an enum.

If and Loop Behaviour

To recognise "if or loop behaviour" the goto-statement has to be in an if-statement. Said if-statement has to have the same parent node as the label [1] (see figures 6 & 7 and listings 46 & 47). Depending on the order of appearance of these two either an "if behaviour" or "loop behaviour" could be detected. Both cases have additional constraints. For the "if behaviour", the goto-statement has to be the last statement [2] in one of the clauses [3]. For the "loop behaviour", it has to be the only statement in the "then"-clause [4].

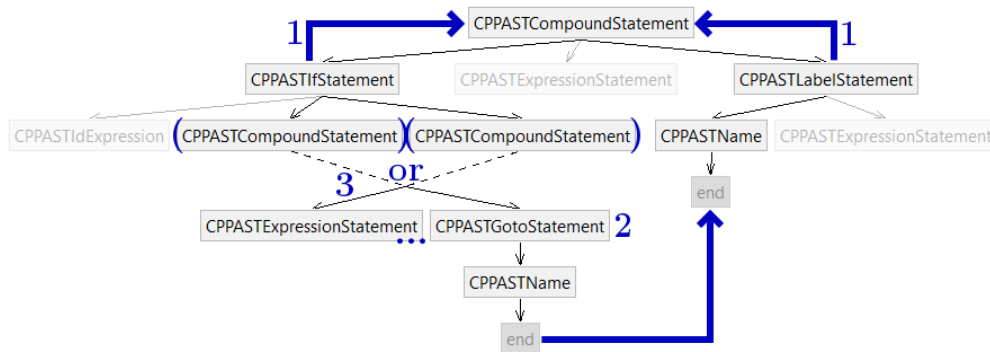


Figure 6: AST nodes used to recognise if behaviour

```

1 {
2   if (someCondition) { // [1] ("if"), [3]
3     doSomething();
4     goto end; // [2]
5   } else { // ([3])
6     doSomething();
7   }
8   doSomething();
9   end: // [1]
10  doSomething();
11 }

```

Listing 46: Code for AST in figure 6

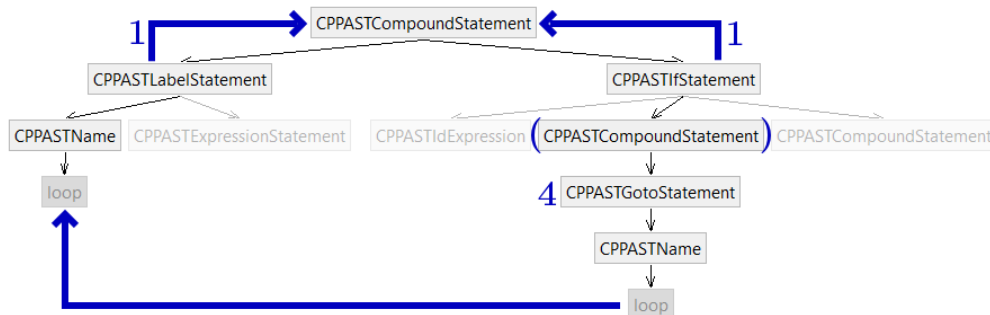


Figure 7: AST nodes used to recognise loop behaviour

```

1 {
2   loop: // [1]
3   doSomething();
4   if (someCondition) { // [1]
5     goto loop; // [4]
6   } else {
7     doSomething();
8   }
9 }

```

Listing 47: Code for AST in figure 7

Break Behaviour

To recognise "break behaviour" the goto-statement [1] (see figure 8 and listing 48) has to be inside a loop [2]. If the label is directly after the innermost loop [3] in which the goto-statement is, a break-statement could be used instead of the goto.

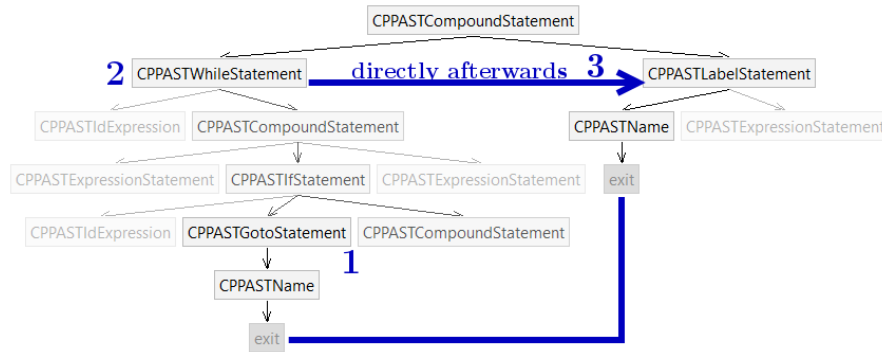


Figure 8: AST nodes used to recognise break behaviour

```

1 {
2   while(someCondition) { // [2]
3     doSomething();
4     if(someCondition)
5       goto exit; // [1]
6     doSomething();
7   }
8   exit: // [3]
9   doSomething();
10 }
```

Listing 48: Code for AST in figure 8

Multi-Break Behaviour

If the label is directly after [4] any other loop [3] further out [2] than directly above the goto-statement [1] (see listing 9 and listing 49), a break-statement would not work, but a surrounding lambda and a return statement might.

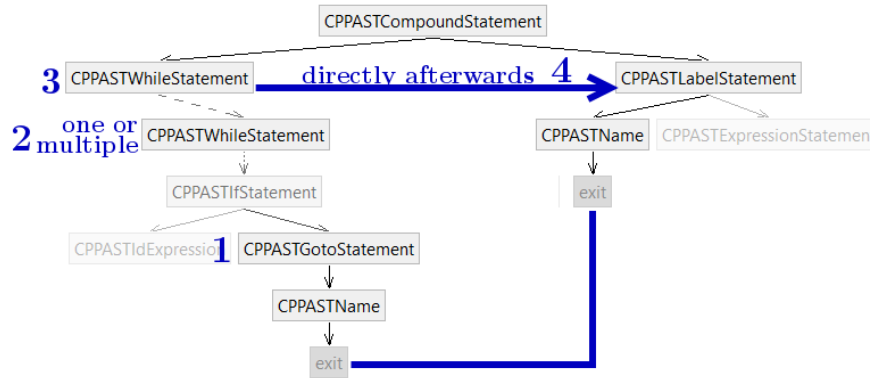


Figure 9: AST nodes used to recognise valid goto usage

```

1 {
2   while(someCondition) {
3     while (someOtherCondition) {
4       if(someCondition) goto exit;
5       doSomething();
6     }
7   }
8   exit:
9   doSomething();
10 }

```

Listing 49: Code for AST in figure 9

If none of the above behaviours are recognised we return a usage pattern of type Unknown which results in a default marker message (if called by the checker) and makes any of the quick fixes (apart from the "set ignore attribute" quick fix) report as non applicable.

4.2.3 Quick Fix

We provide 4 different quick fixes.

Every quick fix checks in the overridden "isApplicable" method if the marker is for the goto usage pattern it was made for using the methods mentioned above. In every quick fix we can remove the label only if it is not used anywhere else. This can be done by replacing it with its contained statement.

Use Normal If-Statement

This quick fix collects all nodes between [1] the if-statement [2] which contains the `goto` [3] and the label [4] (see figure 10 and listing 50).

Then the "then"-clause and the "else"-clause for the new if-statement need to be prepared. For both we copy the previous clause and in one we remove the marked goto [5] statement and in the other we add the previously collected nodes [6].

If the resulting "else"-clause is empty we delete it [7]. If the resulting "then"-clause is empty we negate the condition [8] and make the "else"-clause our "then"-clause [9]. To negate the condition we have to put it in an `IASTUnaryExpression` of type `op_bracketedPrimary` and that in an `IASTUnaryExpression` of type `op_not`. The bracketed expression is not necessary if it is only a single `IASTIdExpression`.

The old if-statement gets replaced with the new one [10] and the old nodes which are now in the `if` can be removed. If the label is not used anywhere else it can be removed as well [11].

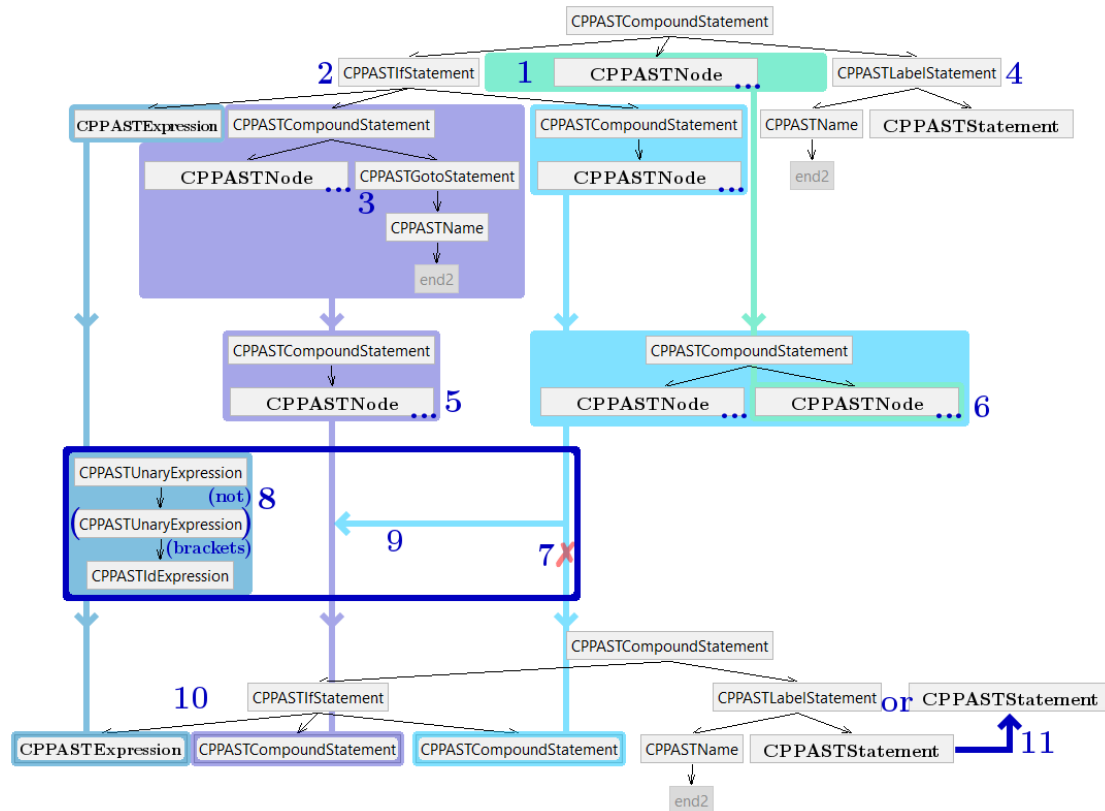


Figure 10: AST nodes used to change a goto to an if-statement

```

1  /* ... */ {
2      if (someCondition) { // [2]
3          doSomething();
4          goto end2; // [3]
5      } else {
6          doSomething();
7      }
8      doSomething(); // [1]
9      end2: // [4]
10     doSomething();
11 }
12
13 // ...
14
15 /* ... */ {
16     if (!(someCondition)) { // [8] "!(...)"
17         doSomething(); // [5]
18     } else {
19         doSomething();
20         doSomething(); // [6]
21     }
22     end2: // may get replaced with [11] below
23     doSomething(); // [11]
24 }

```

Listing 50: Code for AST in figure 10

Use While Loop

Like the previous quick fix this one collects all nodes between the if-statement which contains the `goto` and the label [1] (see figure 11 and listing 51). The node contained in the label-statement does also count to this list [2]. But at first it does not generate a normal while loop but instead a more similar do-while loop [3] using these nodes. Because of this the condition from the `if` can be reused as is [4].

The created do-while statement gets put through the ES.75 logic (see chapter 4.1.2) which returns the `firstRun` declaration statement and the while-statement [5]. The declaration statement can either be replaced or put after the label [6] (depending on, if the label is still used by another goto-statement) and the while-statement replaces the if-statement [7]. Any nodes in the "else"-clause of the if-statement can be put after the while-statement [8]. The old nodes between the label and the if-statement get removed.

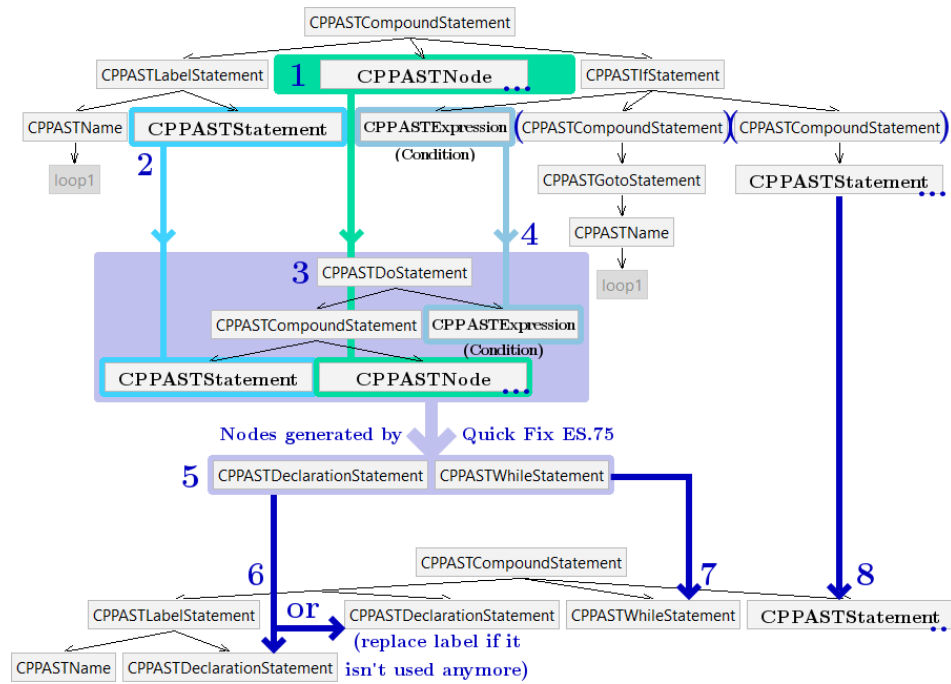


Figure 11: AST nodes used to change a goto to a while-statement

```

1  /* ... */ {
2    loop1:
3    doSomething(); // [2]
4    doSomething(); // [1]
5    if(someCondition) { // [4]
6      goto loop1;
7    } else {
8      doSomething();
9    }
10 }
11
12 /* ... */ {
13   do { // [3]
14     doSomething();
15     doSomething();
16   } while (someCondition); // [4]
17 }
18
19 /* ... */ {
20   loop1: // [6]
21   bool firstRun = true; // [5] inside label-statement
22   bool firstRun = true; // or outside label-statement
23   while (firstRun || (someCondition)) { // [7]
24     firstRun = false;
25     doSomething();
26     doSomething();
27   }
28   doSomething(); // [8]
29 }

```

Listing 51: Code for AST in figure 11

Use Simple Break

This quick fix is as simple as creating a new break-statement [1] (see figure 12 and listing 52) and replacing the goto-statement with it [2] and removing the label if it is not used anymore [3].

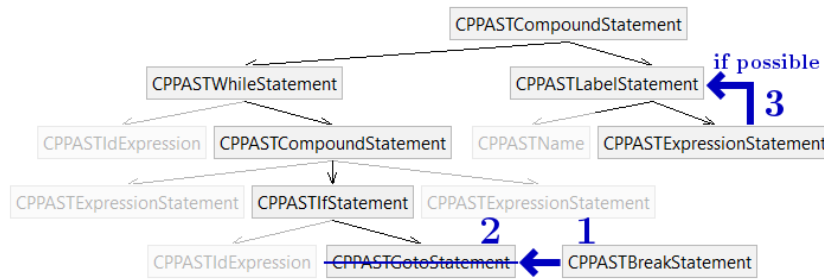


Figure 12: AST nodes used to change a goto to a break-statement

```

1  /* ... */ {
2    while(someCondition) {
3      doSomething();
4      if(someCondition)
5        goto exit; // [2] gets replaced by [1]
6        break; // [1]
7      doSomething();
8    }
9    exit: // [3] removed if possible
10   doSomething();
11 }

```

Listing 52: Code for AST in figure 12

Use Surrounding Lambda and Return

This quick fix has additional checks in the `isApplicable` function. It is only applicable if the only gotos which go outside the while loops are the ones which lead to the lambda directly after the while loops.

In the quick fix we first search which loop is the one directly in front of the label [1] (see figure 13 and listing 53). Once we have determined that, we copy the loop [2]. Then we have to find goto nodes which correspond to the label of the marked goto node [3]. We replace them with a new return statement [4]. This modified loop we can put inside a new compound-statement [5] which we put inside a new lambda expression [6] in a new function call expression [7] in a new expression statement [8]. Now we can replace the old loop with the new lambda expression statement and finally remove the label if possible [9].

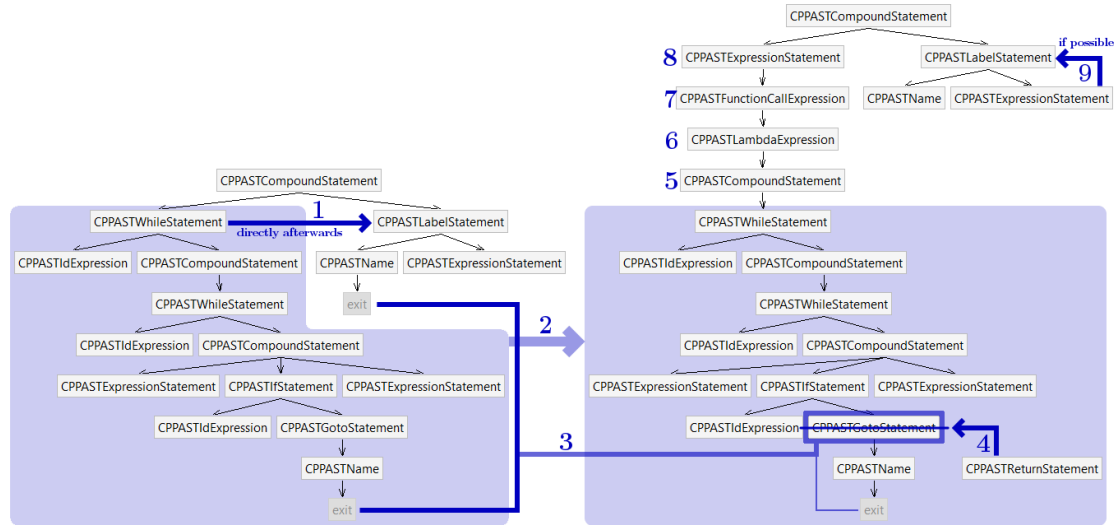


Figure 13: AST nodes used to surround loops with lambda and use return

```

1  /* ... */ {
2    while (someCondition) { // [2]
3      while (someCondition) {
4        doSomething();
5        if (someCondition) goto exit; // [3]
6        doSomething();
7      }
8    } // [1]
9    exit: // [3]
10   doSomething();
11 }
12
13 // ...
14
15 /* ... */ {
16   [&] { // [6] "&", [5] "{...}"
17     while (someCondition) {
18       while (someCondition) {
19         doSomething();
20         if (someCondition)
21           // goto exit; // [3] replaced by [4]
22           return; // [4]
23         doSomething();
24       }
25     }
26   }(); // [7] "()", [8] ";"
27   exit: // [9] removed if possible
28   doSomething();
29 }

```

Listing 53: Code for AST in figure 13

4.3 ES.78: Always end a non-empty case with a break

For this rule we modified and improved the existing checker in codan in CDT's codebase. The issue number in the eclipse bugtracker is #514685 [Bis17e] and the gerrit change id is 94361 [Bis17c].

4.3.1 Checker

Codan's case break checker already searches the last statement of each case where we can just issue a `continue`; if it is a valid fallthrough statement.

For this statement to count as a valid fallthrough it is not allowed to be the last statement of the whole switch. If this statement is a non-empty `IASTCompoundStatement`, further checks get executed on its last statement. The statement has to be an `IASTNullStatement` and have an attribute named "fallthrough". (Note the chapter "StandardAttributes Class" below.)

If the statement gets recognised as a valid fallthrough we do not flag the case.

4.3.2 Quick Fix

Now we can not provide the quick fix for all markers of the checker. Therefore, we did not remove the ignore comment quick fix and we had to override the `isApplicable` function.

Is Applicable?

Because codan is for C as well as C++ we have to check if the marker is for C++ code. Additionally, the fallthrough attribute is only valid for C++17 and above. The C++ version used is not easily checkable from the quick fix code, which is why we added an option to enable this quick fix in the settings and have it disabled by default. And finally, if the setting for checking the last cases is enabled, we have to check if a `[[fallthrough]]`; would be valid at the marker's position.

If the quick fix is enabled, the code is C++ code and the marker's position is a valid position for fallthrough then the quick fix is applicable.

Modifying the AST

The code for modifying the AST is similar to the existing quick fix for adding a break-statement. Which is why we added a new abstract parent class and pulled up nearly all functionality from that quick fix and just made the node to place definable by a parameter.

We generate the new `IASTNullStatement` with the `fallthrough` attribute and just insert it at the correct place using the previous code from the existing quick fix.

4.3.3 JUnit Tests

Of course, we updated and added some JUnit tests based on our changes. Additionally, we noticed that there were no JUnit tests for the ignore comment quick fix. For this we opened a separate bug in the bugtracker (#515814 [Bis17f]) and created a change for it (95765 [Bis17d]).

Note the related subsequent change 96102 [Cor17a] by Thomas Corbat.

4.3.4 CDT Bug 514684 - ASTWriter's StatementWriter does not write Attributes for some Nodes like IASTForStatement

Sadly we ran into the bug of vanishing attributes on rewrite again, which we already had in our term project [BD16, 3.4.2 Set an attribute on a `IASTForStatement`]. Writing nodes of type `IASTForStatement`, `IASTDoStatement`, `IASTNullStatement` and some more do not persist attributes. Using the same workaround again in codan did not make much sense because we are already making changes in the CDT codebase. Why not fix the issue directly?

We only needed to add some calls to `writeAttributes` in the appropriate functions of `ASTWriter's StatementWriter`.

This we did. The issue number in the Eclipse bugtracker is #514684 [Bis17a] and the gerrit change id is 94351 [Bis17b]. Note the related subsequent change 96567 [Cor17b] in which Thomas Corbat simplified `StatementWriters` logic a bit.

4.3.5 StandardAttributes Class

As requested by Peter Sommerlad we created a class named StandardAttributes where we put all standard attribute class names and have them in a central place. This helps with not having strings and char arrays scattered around in code multiple times. The only place in the CDT codebase where an attribute name was already used we updated accordingly.

4.4 ES.9: Avoid ALL_CAPS names

4.4.1 Checker

The checker was nearly as easy as it sounds in the analysis chapter. There were some minor issues. One issue was that the ignore attributes had to be more flexible. More about this can be found below.

Another issue was how to find and mark or ignore macro definitions. For finding and marking non-ALL-CAPS macro definitions solutions were quickly found. Ignoring such macros however is trickier. But considering that these are optional markers, disabled by default, we just won't support ignoring them.

4.4.2 Quick Fix

This rule does not need an extra quick fix because markers on IASTNames provide already a "Rename"-quick fix.

However, the "set ignore attribute" quick fix did not always put the ignore attribute at the wanted position.

4.4.3 Ignore Attributes Issue (Matcher & Quick Fix)

The main issue with ignore attributes was that names can appear at a lot of different locations in code. Previously we defined the position of such ignore attributes relatively statically based on the type of the marked (or to be marked) node.

However, to support this rule we had to change this to a more general approach. Instead of checking if we know where the ignore attribute has to be based on the marked node type, we go up the AST and search the first node which is one of the wanted IASTAttributeOwner.

But there are some node types which are of type IASTAttributeOwner but should not be. One such example is ICPPASTCatchHandler as mentioned in the comments to our gerrit change 94351 [Bis17b] (see also chapter 4.3.4). And there are some node types which are valid attribute owners but if used result in difficult to read code.

E.g. ICPPASTDeclarator:

```
void [[gsl::suppress("Rc-swap-fail")]] swap(foo& other) { /* ... */ }
```

To centralise the logic which attribute owners we want and which not, we created an `AttributeOwnerHelper` class, which now contains the list of invalid and unwanted classes and the logic to get the next parent which is a wanted (for the quick fix) or valid (for the matcher) attribute owner.

Another issue was in the attribute matcher, where we previously just used the first attribute owner we found (with attributes or not). However, here in rule ES.9, it is quite likely that we want to ignore for example all occurrences in one class, or similar.

Therefore, we had to change it to aggregate all attributes found between the marked node and the `IASTTranslationUnit` (the root of the AST). This is now the general behaviour.

The third issue was that if we mark macros which do not support attributes at all, we want to not show the "set ignore attribute" quick fix. To do this, we added an `isIgnoreApplicable` function to the `BaseChecker` which gets called on the marker's checker which now can override it.

4.5 ES.50: Don't cast away const

While implementing this rule we noticed that some changes to "ES.49: If must use a cast, use a named cast" were needed. See chapter 4.6 for more information.

4.5.1 Checker

As mentioned in the analysis chapter any CastExpression with type `op_const_cast` can be marked. For C-style casts we have to check some more things. To find out if it removes a const, we could just reuse slightly modified functions used by ES.46 "Avoid lossy Arithmetic Conversions" [BD16, chapter 3.10]. If the function [2] containing the cast [1] (see figures 14 & 15 and listings 54 & 55) is const, we have to check if the operand is a member of the functions class.

This last part was the trickiest bit. From the functions [2] class [3] we get a list of declarations [4] for which we check if its name [5] matches the name [6] of the operand [7] of the cast. If the function is not defined directly in the class declaration we need to search the class declaration via the name in the CPPASTQualified Name [8] (see figure 15) via the class name declaration from the index [9].

But we cannot just compare the strings of the bindings because that would sometimes give wrong results. For more details see chapter 4.7.3.

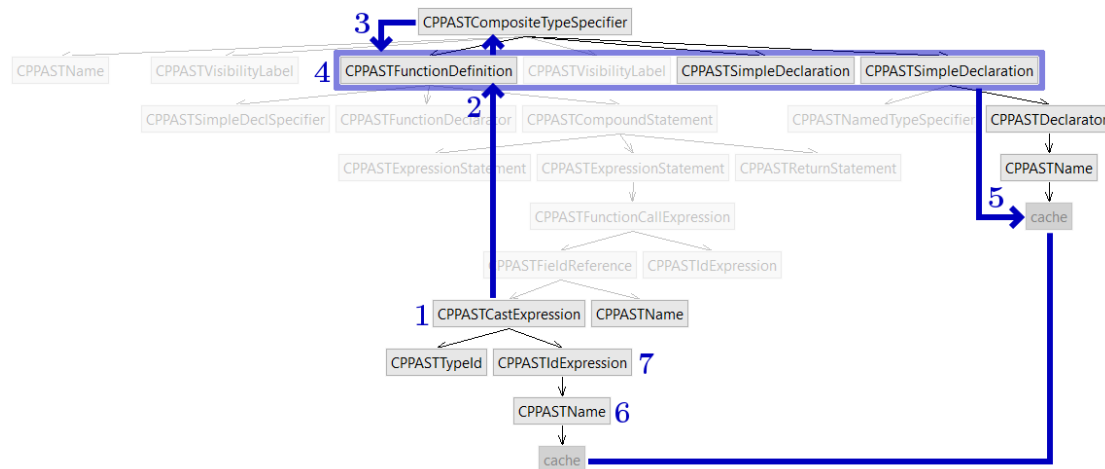


Figure 14: Is the cast operand in the same class as the function using the cast? (Single AST)

```

1 class constFuncExample { // [3]
2   public:
3     int getVal() const { // [2]
4       ((CacheClass&) cache).set(val);
5       const_cast<CacheClass&>(cache).set(val); // [1] "const_cast", [6/7] "cache"
6       return val;
7     }
8   private:
9     int val = 0;
10    CacheClass cache; // [5]
11 };

```

Listing 54: Code for AST in figure 14

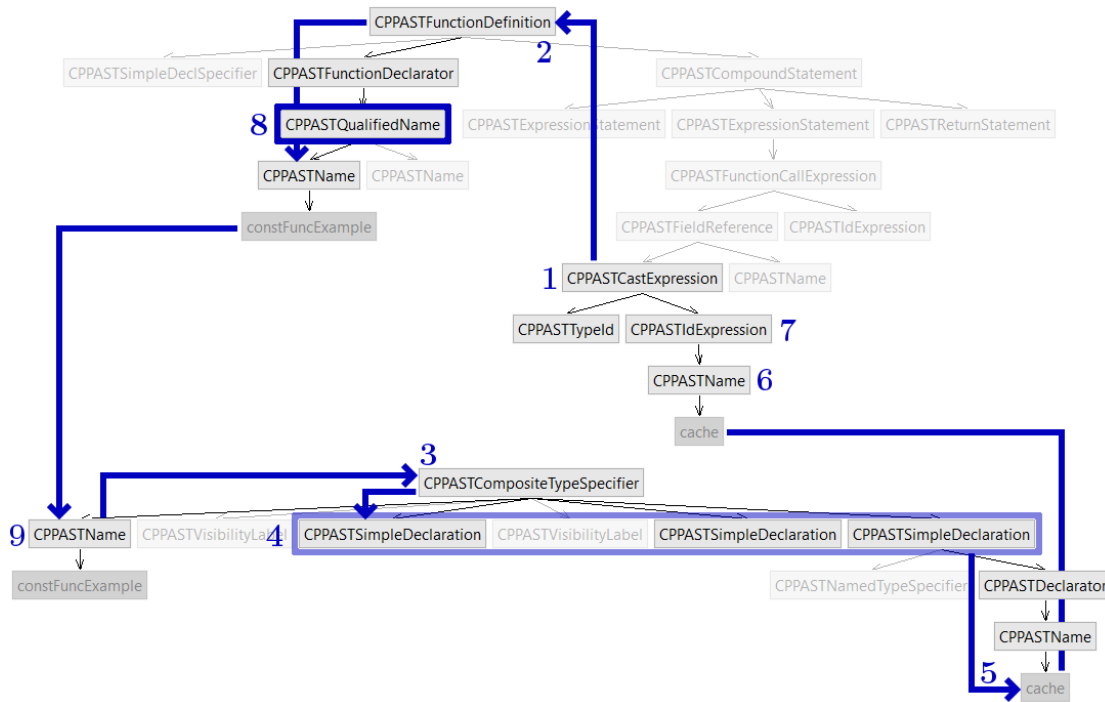


Figure 15: Is the cast operand in the same class as the function using the cast? (Multiple ASTs)

```

1 // main.cpp
2 #include "main.h"
3 int constFuncExample::getVal() const { // [2], [8] "constFuncExample::getVal"
4     ((CacheClass&)cache).set(val);
5     const_cast<CacheClass&>(cache).set(val); // [1] "const_cast", [6/7] "cache"
6     return val;
7 }
8 // main.h
9 // ...
10 class constFuncExample { // [3], [9] "constFuncExample"
11 public:
12     int getVal() const;
13 private:
14     int val = 0;
15     CacheClass cache; // [5]
16 };

```

Listing 55: Code for AST in figure 15

4.5.2 Quick Fix

Here we have multiple different quick fixes. Therefore, we have to check which quick fix is applicable. Additionally, the main quick fix has a different Label based on the AST.

Is Applicable?

We check if the type of the operand is const, the function is const and if the operand is a member of the containing class.

- If the type of the operand is const we can provide the quick fix to remove const from the variable.
- Else, if either the function is not const or the operand is not a member of the class (or both), then we have an unneeded cast and can provide the quick fix to remove it.
- Finally if the function is const and the operand is a a member of the class, then we can provide
 - one quick fix to remove const from the function and
 - one quick fix to set the member variable mutable.

Label

If the function is const and the operand a member of the class the label is

”ES.50: make function non-const and remove const cast”.

If the type of the operand is const the label is

”ES.50: make variable non-const and remove const cast”.

Else the label is

”ES.50: remove const cast”.

The label of the second quick fix class is always

”ES50: make member variable mutable and remove const cast”.

Modifying the AST

The quick fixes consist of two parts. One part where we make sure we have a non-const access to the variable and one part where we remove the const cast.

The first part can be one of three methods. Remove const from the variable declaration, remove const from the function or make the member variable mutable.

Remove Const from the Variable Declaration

With the variable name we search, first in the same AST then via the index, [1] (see figure 16 and listing 56) for the declarator [2] of that name. Now using the IASTDeclarator we get the IASTDeclSpecifier [4] via the IASTDeclaration [3]. We copy the IASTDeclSpecifier, set it to non-const and replace the old one with the non-const one [5]. We might need to get a new ASTRewrite on which to call the replace method if we are in a different AST than the marked node.

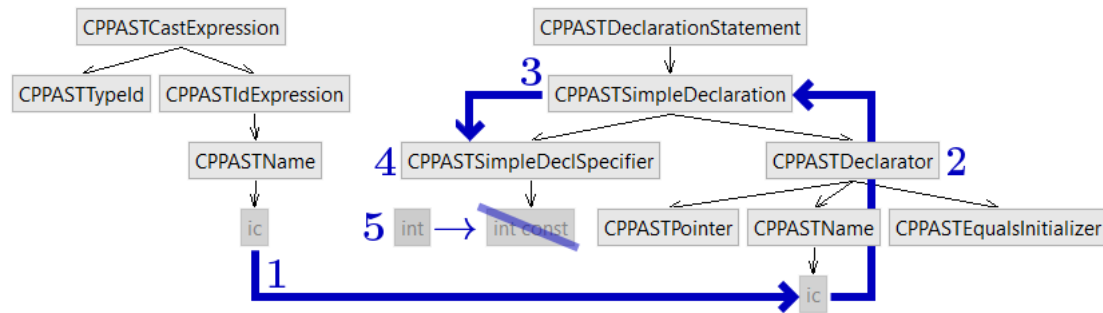


Figure 16: How to find the IASTDeclSpecifier in the AST based on its name.

```

1 int i = 20;
2 int const * ic = &i; // [2-5]
3 int * inew = const_cast<int *>(ic); // [1]

```

Listing 56: Code for AST in figure 16

Remove Const from the Function

To remove const from the function we have to remove it from the declarations as well as from the definition.

Removing it from the definition is easy because that is easily accessible via parent nodes [2] of the cast [1] (see figure 17 and listing 57). We just need to find the IASTFunctionDeclarator [3], copy it and set the copy to non-const and replace the old node with the new one.

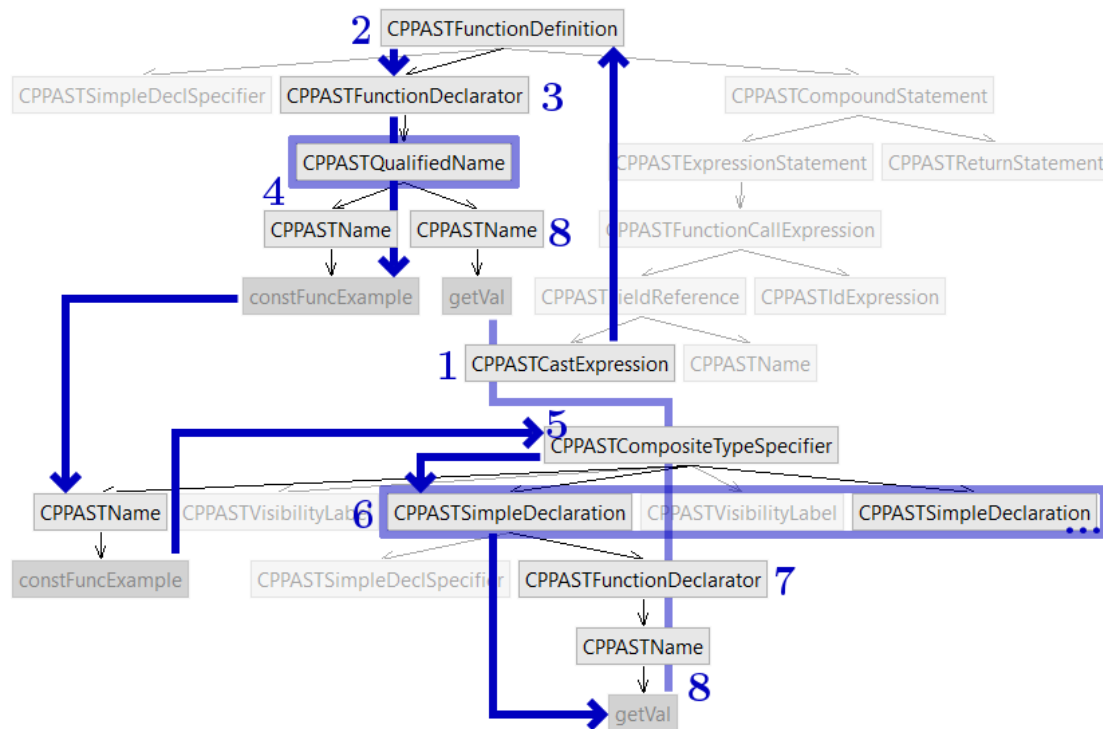


Figure 17: How to find the IASTFunctionDeclarators in the AST based on its name.

```

1 // main.cpp
2 #include "main.h"
3 int constFuncExample::getVal() const { // [2], [3],
                                     [4] "constFuncExample::getVal", [8] "getVal"
4     const_cast<CacheClass*>(cache).set(val); // [1]
5     return val;
6 }
7 // main.h
8 // ...
9 class constFuncExample { // [5]
10 public:
11     int getVal() const; // [7], [8] "getVal"
12 private:
13     int val = 0;
14     CacheClass cache;
15 };

```

Listing 57: Code for AST in figure 17

To find the declaration (if it is separate) we need to find the class [5] via the qualified name [4]. Once we have the class we can search through the declarations [6] and its declarators [7] for one with the same name as the function [8].

With that declarator we can do the same as with the `IASTFunctionDeclarator` from the definition. For that we most likely need to get a new `ASTRewrite` for the new AST to call the `replace` method.

Make the Member Variable Mutable

Using the same method like in chapter "Remove Const from the Variable Declaration" we get the `IASTDeclSpecifier`.

On a copy of the `IASTDeclSpecifier` we can set the "Storage Class" to `sc_mutable` and replace the old node with the new one.

Remove the Const Cast

For all quick fix variations we can replace the const cast with its operand as the last step.

4.6 ES.49: If must use a cast, use a named cast

While implementing "ES.50: Don't cast away const" (chapter 4.5) we noticed that ES.49 needed some modifications.

4.6.1 const_cast in Checker & Quick Fix

We now have ES.50 which says to not cast away const. The ES.50 checker marks `const_cast` as well as C-style casts which cast away const.

Therefore, we can now ignore any cast which cast away const and remove the `const_cast` quick fix.

4.6.2 dynamic_cast Quick Fix

The following description of C-style casts shows that a C-style cast can not be a dynamic cast.

C-style cast and **function-style cast** are casts using `(type)object` or `type(object)`, respectively. A C-style cast is defined as the first of the following which succeeds:

- `const_cast`
- `static_cast` (though ignoring access restrictions)
- `static_cast` (see above), then `const_cast`
- `reinterpret_cast`
- `reinterpret_cast`, then `const_cast`

stackoverflow.com community wiki, *When should static_cast, dynamic_cast, const_cast and reinterpret_cast be used?* [sta16]

Therefore, this quick fix can be removed.

4.6.3 reinterpret_cast Quick Fix

The `reinterpret_cast` is quite a dangerous cast and one should avoid using it. Because we now are using preferences for quick fixes we now also hide this quick fix behind a check box in the preferences, and is disabled by default.

4.7 ASTHelper

The ASTHelper class contains helpful functions to find nodes in the AST based on other nodes or information. As soon as some code needs to be used in multiple rules, we extract that into this class to reduce duplicated code.

This chapter shows changes and additions made to this class.

4.7.1 findNames, getFunctionDeclaratorFromName, getFunctionDeclaratorFromNameInSameTU and getFunctionDeclaratorFromNameViaIndex

These functions were modified, renamed and split while optimising the runtime of the ES.46 checker.

`getFunctionDeclaratorFromName` tries to find the declarator in the same translation unit via `getFunctionDeclaratorFromNameInSameTU`. If nothing is found it tries to find it via `getFunctionDeclaratorFromNameViaIndex` which in turn uses `findNames` using the provided `astCache`. Previously `findNames` did not support an `astCache`.

For more detailed information see chapter 2.2.6.

4.7.2 isInMacro

Nodes generated by C++ macros appear like normal nodes in the AST. But because a quick fix fails if it tries to modify nodes generated by macros, we have to be able to tell if a node is in a macro.

Finding out if a node is generated by a macro is not that difficult. We can loop through the node locations and if any of them is of type `IASTMacroExpansionLocation` the node is generated by a macro.

4.7.3 namesEqual

It seems to be quite difficult to reliably check if two `IASTName` objects name the same thing. (For the following code samples `a` and `b` are `IASTName` nodes.)

Quite quickly we noticed that just using `a.equals(b)` is not enough. But using the bindings does not always work either. `a.resolveBinding().equals(b.resolveBinding())` often returns `false` even though both name the same thing. Mainly when the two `IASTNames` are from two different files and AST.

After some testing we thought that `a.resolveBinding().getScope().getBinding(b, true).equals(a.resolveBinding())` did the trick. But after adding some additional JUnit tests to

our project we noticed some cases where this returned `true` when it should not. This happened with two equally named member variables in two different classes in the same file.

In our current solution we now get for each name a list of all occurrences via the index. Then we compare those two lists using the file name, offset and node length stored in the `IASTFilelocations` of the `IIndexNames`. If the two lists are equal, the two names are equal. This can be done without parsing the AST for the `IIndexName`. Therefore, runtime should not be a big issue.

If we get empty lists, which seems to happen sometimes, we fall back to our first version of `a.resolveBinding().equals(b.resolveBinding())`.

4.7.4 Other Added or Modified Helper Functions

Some other functions were created or needed small modifications to ease the traversal of the AST.

Name	What changed	used by
<code>getASTTranslationUnit</code>	Added index read-lock. (See chapter 4.8.5.)	multiple
<code>collectMemberFunctions</code>	Fixed cast exception.	C.83, C.85
<code>getDeclSpecifierFromDeclaration</code>	Added support for more declaration nodes. Using Reflection to check if the declaration node has a <code>getDeclSpecifier</code> method.	ES.50, C.83, ...
<code>getTypeFromExpressionElement</code>	Modified to be able to get the List of <code>IASTTypeId</code> not converted to strings.	ES.46, ES.50
<code>getNameOfFunctionFromDeclaration</code>	New – Gets the Name of a Function Declaration.	C.20
<code>getLoopVariable</code>	New – Finds the first id expression in the iteration expression of a for-loop	ES.74
<code>getChildExpression</code>	Modified to fix a bug.	ES.74
<code>isForLoopStatement</code>	New – Searches the next parent node which has a Scope.	ES.75
<code>getNextParentScope</code>	New – Searches the next parent node which is a Loop.	ES.76
<code>getNextOupterLoop</code>	New – Checks if one node is directly afther the other.	ES.76
<code>isDirectlyAfterwards</code>	New – Gets the children of the parent node and finds the node after the current one.	ES.76
<code>getNextNode</code>	New – Checks if a given Name is a member of a given Class.	ES.50
<code>isNameMemberofClass</code>	Gets the list of <code>IASTDeclarators</code> from a <code>IASTDeclaration</code> .	ES.50
<code>getDeclarators</code>	New – searches first the translation unit then the index for the declarator of a given name.	ES.50
<code>findDeclaratorToName</code>	New – searches the AST for the next outer <code>IASTDeclaration</code> .	ES.50
<code>getDeclaration</code>	New – checks if the type is <code>const</code> for different instances of <code>IType</code> .	ES.50
<code>isTypeConst</code>		

4.8 Other Changes

All changes to CCGLator not specific to a rule we describe in this chapter.

4.8.1 Handling Markers at Nodes in Macros

During testing the plug-in on a Real World Application (see chapter 2.2) as well as when implementing rule ES.75 (see chapter 4.1) we noticed issues with marked nodes generated by macros.

When analysing an AST, nodes in macros get expanded where macros get used and appear as normal nodes in the AST. Therefore in a project which uses macros, lots of markers were generated where the issue was not directly visible but inside a used macro. Additionally, for one issue in a macro you would have gotten a marker for every macro usage.

To prevent this we decided to not mark any node inside a macro. To do this we modified our `BaseChecker` and added a check for `isInMacro` (see chapter 4.7.2) in the overridden `reportProblem` functions.

4.8.2 Not Expanding Macros in Reused Nodes

Every call like `node.copy()`; expanded used macros. This modified the source more than necessary.

`ASTRewrite` already supports generating code based on nodes from macros, but needs the macro location saved on them.

To not loose this information when copying a node, we just had to call `copy` with the appropriate `CopyStyle` like this: `node.copy(CopyStyle.withLocations)`;

We searched our existing codebase for every call to `INode.copy()` and made sure to change this if needed.

4.8.3 Changing the Ignore Attribute

The C++ Core Guidelines now define the format of the ignore attribute. [SS15, In.force]

To comply with the guidelines we now use `gsl::suppress` instead of `ccglator::ignore`. Likewise, instead of using the rule numbers (like "C.83") which are subject to change, we use now the anchor names of the rules as defined by the guidelines. (See figure 18, hint: do not use the link at the rule title.)

Because we needed to add the anchor names to all the checkers, we used this to clean up the code a bit and move the ignoreStrings to the same place as the problem-IDs.

Additionally, every JUnit test had to be updated.

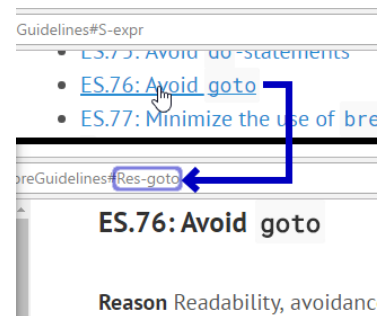


Figure 18: How to find the anchor name of a rule

4.8.4 Defining a Marker Type

As per Peter Sommerlad's request we defined the `markerType` of our markers such that they have their own group and do not get mixed with other general markers anymore.

This was just a small adjustment to the `plugin.xml`. To suppress a warning we added a stub class extending the `CodanProblemMarkerResolutionGenerator`.

4.8.5 Lock Index

In the `findNames` and the `getASTTranslationUnit` methods of the `ASTHelper` class and in the `getASTTranslationUnit` method of the `ASTRewriteStore` class we forgot to lock the index upon usage. To prevent issues we fixed this.

Without this correction we got assertion errors during JUnit tests. This happened on checks to the locked status in the index database.

4.8.6 Merging Problem-IDs

For some rules we had multiple problem-IDs. For example for rule ES.46 we had 12 problem-IDs to be able to disable different types of lossy casts independently. Or rule ES.74 had one problem-ID for markers which support a quick fix and one without. To tidy up this mess we started using problem preferences as well as the "isApplicable"-check in quick fixes.

Problem Preferences

In order to switch to using preferences, we had to report the same problem-ID everywhere in the checker but before reporting check the preferences if it is enabled for the current case.

The rules now having preferences are:

- ES.46: Avoid narrowing conversions
- ES.49: If you must use a cast, use a named cast
- ES.9: Avoid ALL_CAPS names (new rule)
- ES.76: Avoid goto (new rule)

isApplicable

In order to use `isApplicable` instead of multiple problem-IDs, any special-casing in the checker can be removed and we can always report the same problem-ID. Optionally, different problem arguments can be reported to still have different problem descriptions. The checks which decided which problem-ID to use need to be changed to report a boolean value in the overridable `isApplicable` method in the quick fixes.

The rules we updated to use this function are:

- C.45: Don't define a default constructor that only initializes data members; use in-class member initializers instead
- C.83: For value-like types, consider providing a noexcept swap function
- ES.74: Prefer to declare a loop variable in the initializer part of a for-statement

For new rules we used `isApplicable` too.

Future Work

Rules still having multiple problem-IDs are:

- C.31: All resources acquired by a class must be released by the class's destructor
- C.60: Make copy assignment non-virtual, take the parameter by `const&`, and return by `non-const&`
- C.63: Make move assignment non-virtual, take the parameter by `&&`, and return by `non-const&`

4.8.7 Making the Ignore Attribute Matcher & Quick Fix more Generic

Our changes to the ignore attributes matcher and quick fix are explained in detail in chapter 4.4.3.

4.9 Testing

Major parts of this chapter have been copied from our term project [BD16].

In this project all the rules implemented are tested using the IFS CDT Testing Tools [fSR16a]. These tools allow easy testing of checkers and quick fixes. The test cases, consisting of a code-block and a configuration, have to be written into a .rts-File. For every rule and quick fix a separate file has to be created.

4.9.1 Checker

To test a checker, the configuration supports a "markerPositions" attribute. With it, we can define on which line in the test code the marker should appear. If none is provided no marker should appear. See listing 58 for an example.

```
1  //!SwappableClassInNamespace
2  //@.config
3  markerPositions=2
4  //@main.h
5  1 namespace swap {
6  2     struct SwappableMember {
7  3     void swap(SwappableMember &other) { }
8  4     };
9  5 }
10  //!NamespaceLevelSwapInOtherFile
11  //@swap.cpp
12  1 namespace swap {
13  2     struct SwappableMember {
14  3     void swap(SwappableMember &other) { }
15  4     };
16  5 }
17  //@swap2.cpp
18  1 namespace swap {
19  2     void swap(SwappableMember &a, SwappableMember &b) {}
20  3 }
```

Listing 58: Two tests for the C.85 Checker

4.9.2 Quick Fix

A similar approach is possible for testing a quick fix. First off the code before the quick fix is written. Separated with the "//=" string, the code after the execution of the quick fix can be defined. See listing 59 for an example.

```
1 //!ClassInNamespaceWithoutSwapFunction
2 //@main.h
3 1 namespace swap {
4 2 struct SwappableMember {
5 3 void swap(SwappableMember &other) {}
6 4 };
7 5 }
8 //=
9 1 namespace swap {
10 2 struct SwappableMember {
11 3 void swap(SwappableMember &other) {}
12 4 };
13 5
14 6 void swap(SwappableMember& a, SwappableMember& b) noexcept
15 7 {
16 8 a.swap(b);
17 9 }
18 10 }
```

Listing 59: Test for the C.85 quick fix

4.9.3 To Do's

Some of the previously mentioned changes needed updates in the JUnit tests.

Testing with Different Problem Preferences

Because we merged problem-IDs some of the JUnit tests where we expected only one marker now had those of the other types too.

To be able to test them independently we needed to add a way to change the preferences of a problem before a test.

Without modifying the CDT Testing Tools the only way we found was to have an "override preference" map in the BaseChecker. This is because CDT-Testing calls `runCodan()` directly after modifying the preferences which overrides any previously modified preferences.

But the modification needed in the CDT Testing Tools was quite simple. A simple overridable `problemPreferenceSetup(RootProblemPreference preference)` which gets called before calling `runCodan()` solves this issue. For this we made a small Pull request for the IFS CDT Testing Tools repository on GitHub [Bis17g].

By overriding this new function in the test classes we now can modify the preferences of the tested problem.

Support for Testing "IsApplicable"

To be able to test the functionality of `isApplicable` we added a new test-property of the same name. If set, it will be tested if the quick fix is applicable and if that is expected before running the quick fix.

Testing a Quick Fix Where There Are Always Multiple Markers

The checker of rule C.83 had a problem-ID which was always only marked additionally to the main problem-ID. By merging the problem-IDs we now had the issue that when trying to test that quick fix the JUnit tests could not know which of the two markers should be used.

To fix this we defined a new test-property "markerNr" which tells the JUnit test which of the markers should be used. If not supplied, the old behaviour to expect exactly one marker is preserved.

To support this we needed to sort the marker list ourselves. This sorting function is now in a `MarkerHelper` class.

5 Implementation GslAtorPtr

For GslAtorPtr the most recent commit for CharWars was used and a new branch was created containing all the code of GslAtorPtr. On this branch all of the following work was done.

5.1 Pointer and Size Parameter

Whilst the GSL `span<T>` type was already implemented in the GslAtorPtr [GM16], the similarities between `span<T>`, `string_span<>` and `string_view` came to light in the analysis of the types. This resulted in the possibility of reusing a large portion of the codebase of `span<T>` for the other two types.

5.1.1 Checker

The checker for a `span<T>` already implemented a lot of the logic needed to handle `string_span<>` as well as `string_view`. Because of this, it was renamed to `PointerSizeParameterProblem-Generator` and customised to also handle `string_span<>` and `string_view`. For this the check which type of pointer it is, has to be added. When the pointer is neither an unsigned or signed char like type, either a `string_span<>` or `string_view` can be applied. To decide which one of these two is the right one, the pointer checks if it is constant and if the chosen C++ version is C++17 or higher (see 5.1.3 for more information). If both of these conditions are met, a `string_view` can be used.

5.1.2 Quick Fix

The first implementation of the quick fixes showed that with the current state of the code, a lot of duplication had to be generated to handle the three different types.

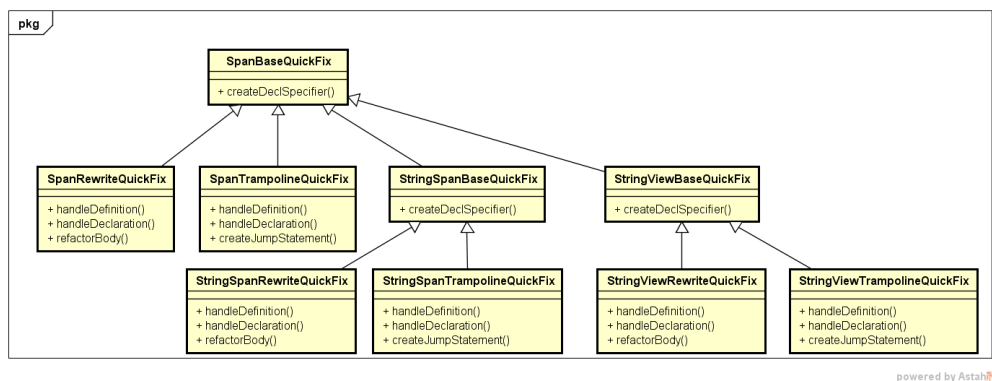


Figure 19: First implementation of `string_span` and `string_view`

As can be seen in the diagram above (figure 19), the only real difference for `span<T>`, `string_span<>` and `string_view` is the method `createDeclSpecifier` in the BaseQuickFix classes. This method handles the generation of the used type in the quick fix. All the other methods in the classes `RewriteQuickFix` and `TrampolineQuickFix` are the same. While the fixes worked as intended, the design was not satisfactory. The new design for all three quick fixes avoided a lot of code duplication (see figure 20).

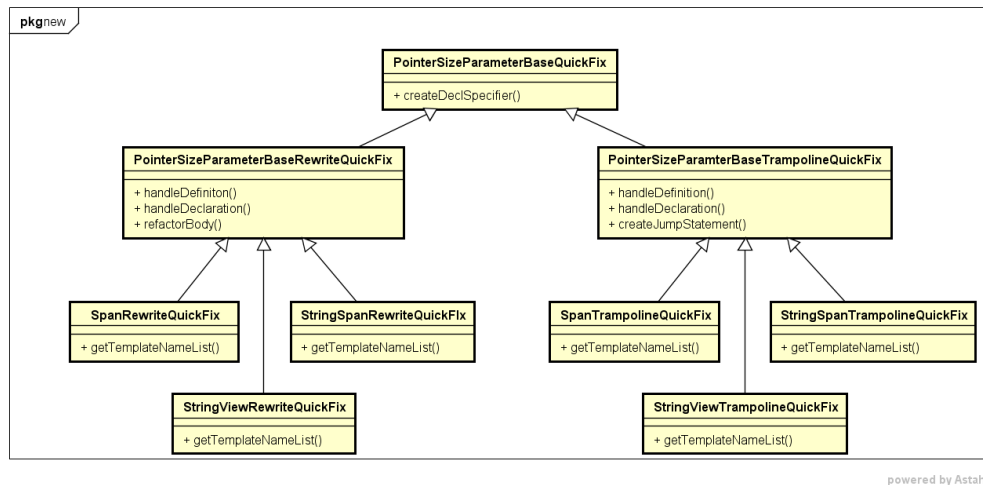


Figure 20: New design for pointer and size parameter

To achieve this, a new abstract class named `PointerSizeParameterQuickFix` was introduced. In this class the method `createDeclSpecifier` is implemented which calls the method `getTemplateNameList`. This method handles the creation of the appropriate type depending on the reported error.

The quick fix specific methods like `handleDefinition` and `handleDeclaration` are now in their own baseclasses instead of in every quick fix, resulting in less duplicated code. A downside to this design is the similarity between a `RewriteQuickFix` and a `TrampolineQuickFix`. The method `getTemplateNameList` is defined in both of them and results in the same behaviour. Because of this, the trampoline classes call the method from their respective rewrite class. Because the plugin requires a class which can be called if a quick fix is executed, this can not be avoided.

5.1.3 Checking C++ Version

For the check which C++ version is used, the plug-in Elevenator [fSR17] can be used. This plug-in allows to generalise the used C++ version for all newly created projects in a workspace, reducing the configuration overhead needed when creating a new one. Additionally, it allows to query the used C++ version from a project. This feature is used in the checker to decide if a `string_view` can be used or if the C++ version is lower than C++17.

5.1.4 Testing Framework Compatibility

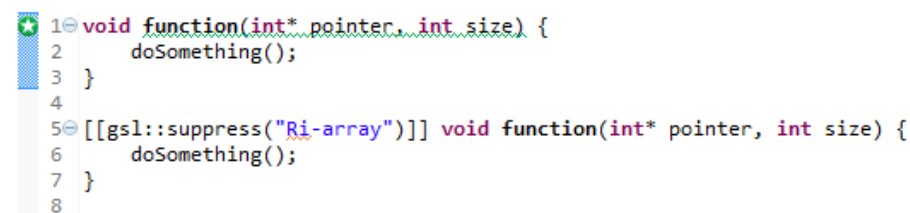
Because the checker uses the Elevenator plug-in, the tests for a `string_view` failed automatically, because the found C++ version was always set to C++11 in the testing framework. So to fix this problem, it has to be possible to set a `CPPVersion` in the checker which is used instead of the found version from Elevenator. After adding this field it has to be set to C++17 in the `setUp` method from the `string_view` tests. With this change the testing worked as intended.

5.2 Suppression of Warnings via Attribute

As described in the future work chapter of GslAtrPtr [GM16] the suppression of warnings via attributes was a feature they wished to implement but did not have enough time for. This feature is now added to the plug-in but was further divided into two problems. For the GslAtrPtr problems, the ignore attribute functionality was implemented whilst for the CharWars problems, the suppression is achieved by ignore comments.

5.2.1 Ignore Attribute

For the problems of GslAtrPtr an ignore attribute can be defined in a very similar fashion to what CCGLator already provides. Because all the features of GslAtrPtr are based on the C++ Core Guidelines, the argument of the attribute can be set to the anchor of the corresponding rule like in figure 21.



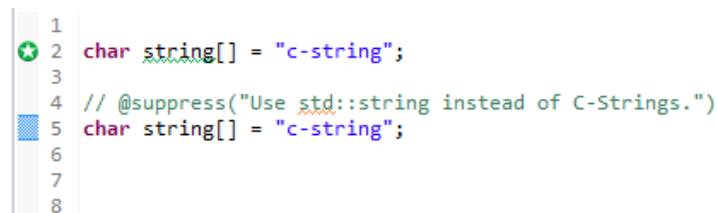
```
1 void function(int* pointer, int size) {  
2     doSomething();  
3 }  
4  
5 [[gsl::suppress("Ri-array")]] void function(int* pointer, int size) {  
6     doSomething();  
7 }  
8
```

Figure 21: Ignore Attribute used in CharWars

To implement this several classes were copied from the CCGLator plug-in. The "set ignore attribute" quick fix handles the generation and placement of the attribute when run as a quick fix and the AttributeMatcher class supports the checkers in finding an already present attribute on the respective node.

5.2.2 Ignore Comment

Codan provides per default a suppress comment per problem. So for CharWars problems, these values are extracted from the settings page of a problem and then used to either check or generate the ignore comment (see figure 22). Luckily codan provides a `CommentMap` which allows the lookup of comments for a specific node.



```
1
2 char string[] = "c-string";
3
4 // @suppress("Use std::string instead of C-Strings.")
5 char string[] = "c-string";
6
7
8
```

Figure 22: Ignore Comment used in CharWars

To decide on which node a comment has to be added, the same method as for ignore attributes can be used.

5.2.3 The Problem with Static Generators

During the implementation a problem came to light with the generator classes which handle the generation of problems. In the old state, they were implemented statically, however the method used to check for an ignore attribute or comment would be best implemented in a shared base class. For this a new class `BaseGenerator` was added from which all the other generators extend and all generators had to be changed to non static methods. With this change the two methods used for checking can be implemented in the base class, avoiding duplicated code.

5.3 Improve Span Refactoring

As discussed in the analysis chapter 3.3, the span refactoring can be improved with two additional forms of function interfaces. To support these changes the checker as well as the quick fix has to be improved.

5.3.1 Additions to the Checker

The original version reported the found pointer parameter position as an argument to the quick fix. However, if multiple pointers should be found, it has to report multiple positions. So the improved checker reports one list with all the found pointer parameter positions and one list with the according size parameter positions. So the whole analysis which pointer needs which size parameter is done in the checker.

5.3.2 Additions to the Quick Fixes

The quick fix calls three different methods. One handles the update of the callsite, one the declaration and one the definition of the function. All three methods now also receive the list with pointer and size parameter positions instead of only the single pointer parameter position. This allows to fix all the found problems in one run. For this several loops had to be added to execute the refactoring statements on multiple pointer parameters. The quick fix can now handle code like in figures 23 and 24.

```
1 void bimbambozle(int* a, int size1, int* b, int size2);
2 void bimbambozle(int* a, int size1, int* b, int size2){
3 }
4
5 int main(){
6     int len { 10001 };
7     int overEight[len] { };
8     int overNine[len] { };
9     bimbambozle(overEight, len, overNine, len);
10 }
```

Figure 23: Before execution of span refactoring

```
1 #include "gslrefactor.h"
2 void bimbambozle(gsl::span<int> a, gsl::span<int> b);
3 void bimbambozle(int* a, int size1, int* b, int size2);
4
5 void bimbambozle(int* a, int size1, int* b, int size2) {
6 }
7
8 void bimbambozle(gsl::span<int> a, gsl::span<int> b) {
9     return bimbambozle(a.data(), a.size(), b.data(), b.size());
10 }
11
12 int main(){
13     int len { 10001 };
14     int overEight[len] { };
15     int overNine[len] { };
16     bimbambozle(gsl::span<int> { overEight, len }, gsl::span<int> { overNine, len });
17 }
```

Figure 24: After execution of span refactoring

6 Conclusion

This section gives an overview of the results achieved in this project as well as pending work possible to be done in other projects.

6.1 CCGLator Results

This project extended the existing plug-in with five new rules enforcing the correct use of switch-statements, avoiding do-statements and highlighting of all uppercase names. For each rule, tests were written to ensure the correct behaviour of the plug-in. Additionally, a lot of work was invested into improving the code and making the plug-in more user-friendly and more expandable. For the rule "ES.78: Always end a non-empty case with a break" two changes to Eclipse CDT were proposed and are already merged into the CDT codebase. The changes are to be released with the next CDT version.

6.2 GslAtorPtr Results

The plug-in was improved to handle more dynamic function interfaces and support more specific types in the quick fixes. Additionally, the support to set attributes or comments to ignore specific rules was added, resulting in a coherent handling of ignoring rules in both plug-ins.

6.3 Future Work

As stated in the introduction, the C++ Core Guidelines [SS15] are an extensive set of rules and only a small part of them are already implemented in this project. For future work additional rules can be implemented. Especially the chapters "Interfaces" and "Functions" from the C++ Core Guidelines define a lot of rules which could use an implementation. For this the plug-in allows an easy extension of the already available rules to support new ones. Some future work points in GslAtorPtr and CharWars are still not implemented and could be tackled in another project.

6.3.1 Merge Problem-ID's

As already mentioned in chapter 4.8.6 there are still some rules having multiple problem-IDs and therefore multiple entries in the settings. It might make sense to also merge these and detect the cases using `isApplicable` and by using problem preferences. The rules are C.31, C.60 and C.63.

6.3.2 Cross File Changes

A lot of rules rewrite a function interface but do not handle the same interface in another file. For example if a function is declared in a header file and the definition is in a cpp file only the definition is updated. No changes are made to the header file. Such a behaviour can be observed in multiple rules and would be a point to improve. Some examples are: C.83, C.84, C.164. As a reference implementation one might have a look at ES.50.

6.3.3 ES.46 Runtime

The runtime of the checker for the rule ES.46 is still higher than all other checkers. We already improved the runtime a lot as mentioned in chapter 2.2.6. However, it might be possible to get the needed parameter types directly from the bindings (of type ICPPFunction) in the index without the need for parsing the AST of other files.

A Project organisation

In this chapter the organisation of the project is outlined. This contains a time report, used tools and a overview of our approach.

A.1 Approach

Because we worked on two different plug-ins we each focused on one of them. But in case of design or comprehension questions, we asked each other for a second opinion. Rolf Bislin focused on CCGLator, Kilian Diener focused on CharWars.

For implementing new rules in the CCGLator we dealt with one rule at a time and wrote tests, analysis, checkers and quick fixes for this rule. After the rule was done the next one was tackled.

For CharWars the existing infrastructure had to be extended or changed for the new features. This resulted in a lot of trial and error to get an understanding of the plug-in and find the right methods to write or extend.

With this approach we were mostly independent from each other allowing us to work on the project according to our own schedule.

A.2 Project Plan

This project had a time budget of 360 hours per student. This results in 21 hours and 11 minutes per student per week over the timespan of 17 weeks. The actual achieved time per student is:

- Rolf Bislin: 351 hours
- Kilian Diener: 335 hours

A report based on worked hours per week is found in the figure 25.

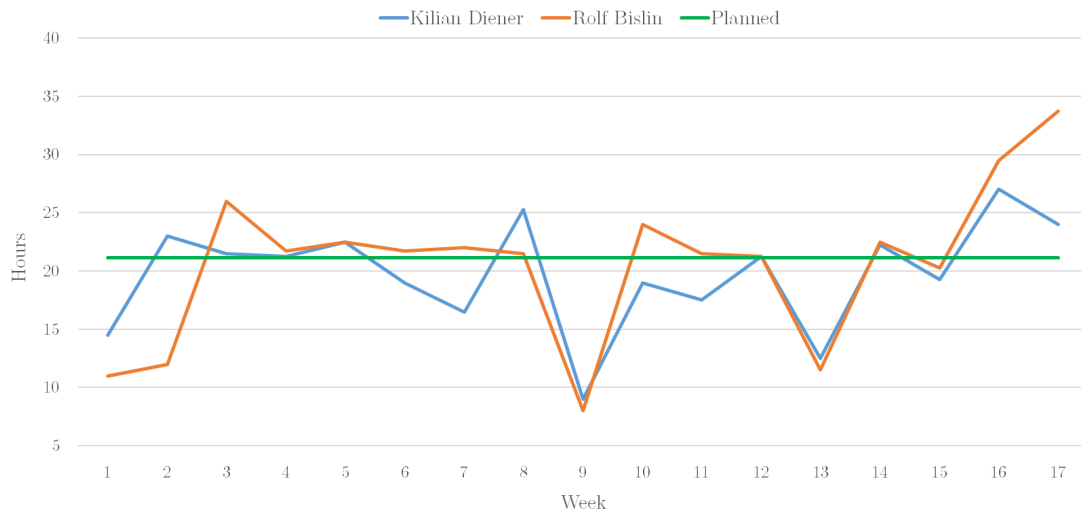


Figure 25: Planned vs. actual hours worked per week

A.3 Project Management Environment

The virtual server provided by the HSR contained a Redmine installation which we used as a project planning and time management tool as well as to store our agenda item lists for our weekly meetings. For the Redmine database we created a daily cronjob to make a backup onto an external online storage.

We also used an additional Apache docker container to provide a stable copy of the CCGLator's plug-in update site.

A slight modification to the nginx proxy installation was needed to enable transferring files bigger than 1 MB.

A.4 This Document

To generate this Document we used \LaTeX which we installed using the MiKTeX Installer [Sch] and as an editor we used TeXstudio [vdZ⁺]. To edit the images in this document we used Gimp. [Tea]

B User Manual

This user manual is an updated version of the one written in our term project [BD16]. The manual contains a quick overview on how to install, configure and use the CCGLator or CharWars plug-in for Eclipse.

B.1 Installation

To use the CCGLator or CharWars plug-in install it via "Help → Install New Software..." and use the appropriate update site from below or the path to the correct "updatesite" folder in the archive.

CCGLator <http://sinv-56012.edu.hsr.ch/updatesite-ccglator/> (figure 26)

CharWars <http://sinv-56012.edu.hsr.ch/updatesite-cute/> (figure 27)

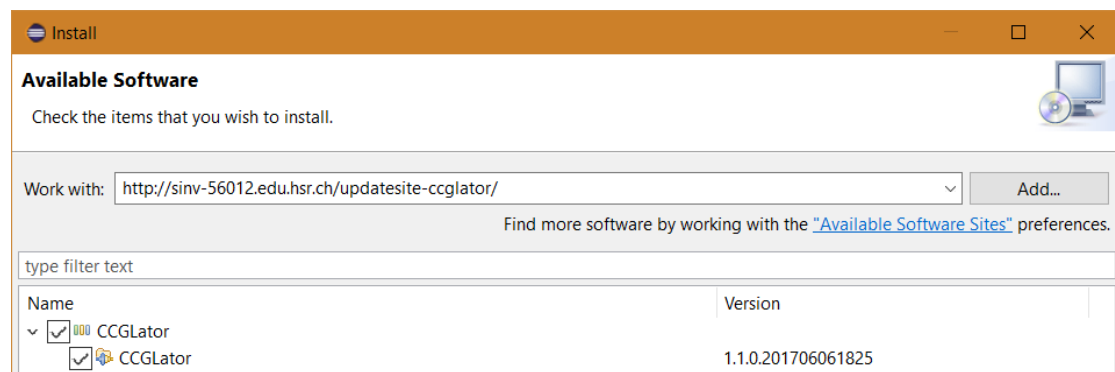


Figure 26: Installing the CCGLator plug-in via the update site

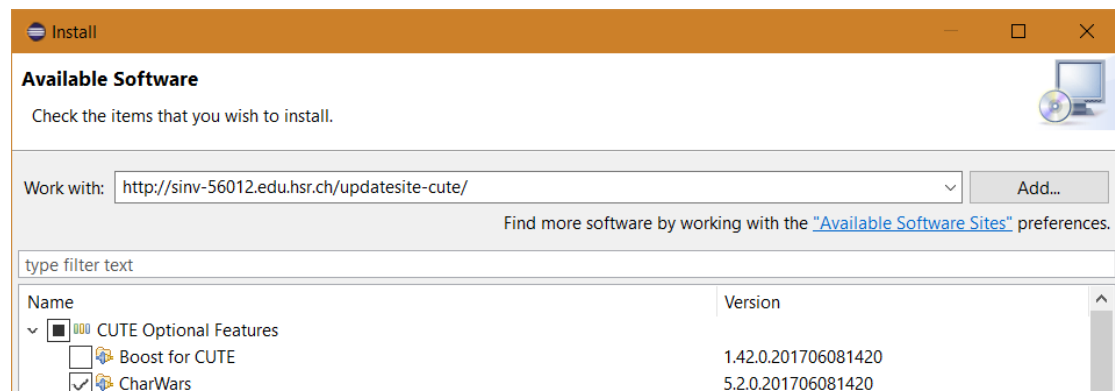


Figure 27: Installing the CharWars plug-in via the update site

B.2 Configuration

To select which rules should be active, open
”Window → Preferences → C/C++ → Code Analysis” or
”Project → Properties → C/C++ General → Code Analysis” and
select the rules from the list: (figure 28)

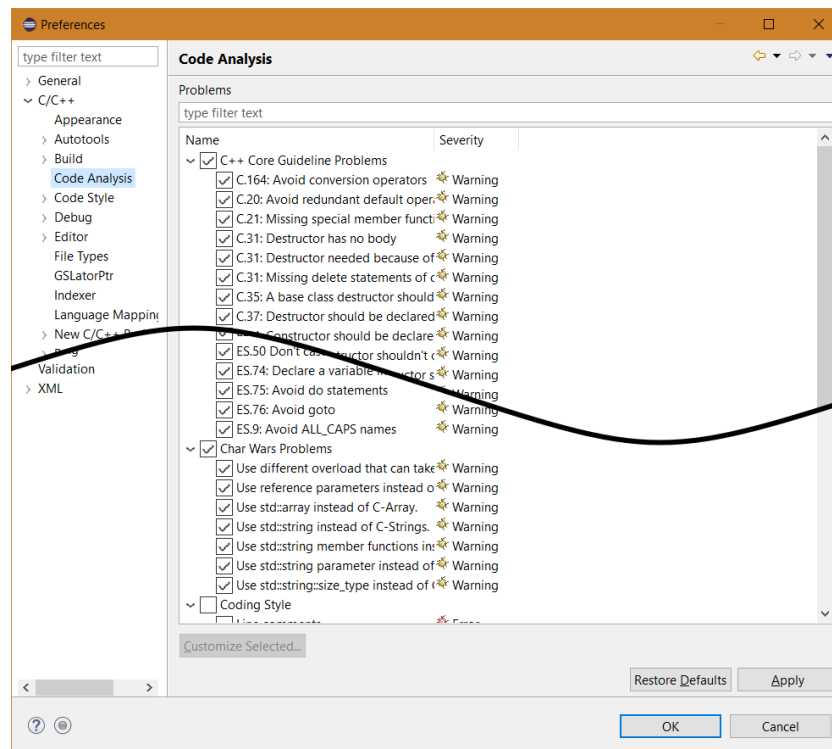


Figure 28: The Code Analysis Selection Screen

Some rules have additional configurations under "Customize Selected...". One example screen is seen in figure 29.

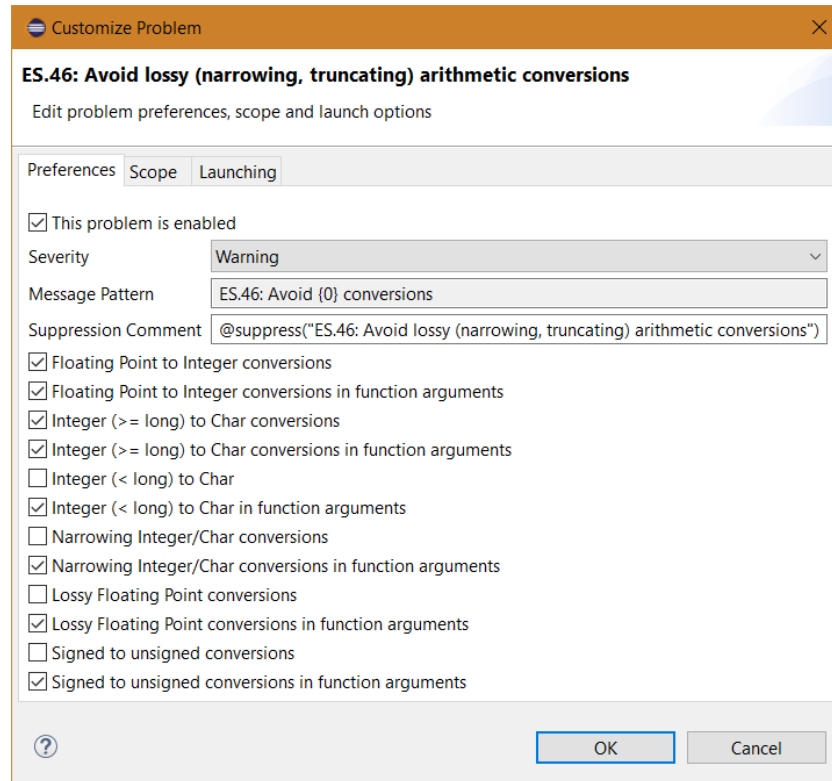


Figure 29: Customize Problem Screen

One additional configuration screen for GslAtorPtr is found under "Window → Preferences → GslAtorPtr" (see figure 30). On this page the include of the GSL headers can be configured and the name of GSL-types can be changed.

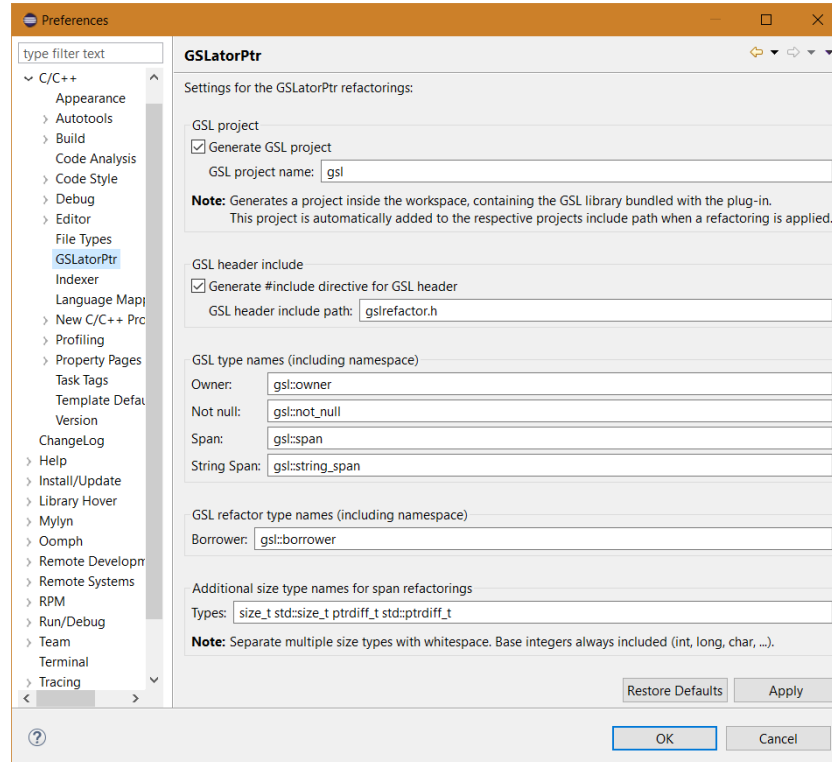


Figure 30: The Code Analysis Selection Screen

B.3 Usage

Any found issue of the enabled rules will get marked with a yellow squiggly underline in the code [1] (see figure 31). Using the icons on the left [2] the quick fix list is opened [3]. From that list a quick fix can be chosen to be applied to the code.

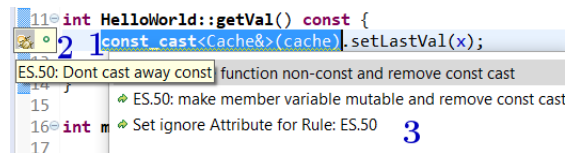


Figure 31: Using the plug-in

C Developer Manual

This developer manual is an updated version of the one written in our term project [BD16]. The manual covers how to set up and use the development environment for the code base of a codan plug-in locally as well as on a continuous integration server.

C.1 Prerequisite

The following requirements have to be met:

- Git [gs] has to be installed and known
- Java Development Kit (JDK) 8 [Ora] needs to be installed
- The Eclipse SDK, found on the Eclipse Download Page [Foua], is used as an IDE.
- A working C++ compiler must be installed and working in a normal Eclipse C++ environment.

On Windows we recommend using the MingW package from Nuwen.

Use these instructions by the IFS C++ Wiki [SCH⁺16]:

1. download MingW from Nuwen [Lav]
2. install MingW Nuwen
3. add the MingW bin directory to the system PATH
4. in the bin directory copy "cpp.exe" and name it "x86_64-w64-mingw32-gcc.exe" to help Eclipse find it.

C.2 Setting up the Eclipse Workspace

To set up the Eclipse workfolder for the plug-in follow these steps:

1. Clone the git project (or extract it from a zip file)
(`git clone https://hmuster@git.hsr.ch/git/CCGLadiator ccglator` or
`git clone https://hmuster@git.hsr.ch/git/Cute cute`)
2. Open Eclipse and choose some workspace folder
(e.g. "C:\CCGLImperator\workspace"
Depending on the Project this can be the same as the plug-in source folder.)
3. Open File → Import → General → Existing Project into Workspace
4. Select the plug-in source folder as root directory
(e.g. "C:\CCGLImperator\ccglator")
5. Optionally deselect unwanted projects from the list
6. Click Finish
7. In the Package Explorer search and open the target definition file.
(e.g. "ch.hsr.ifs.cute.ccglator.targetdefinition.target"
or "ch.hsr.ifs.cute.target.cdt921.target")
8. Click on "Set as Target Platform" in the upper right corner and wait for it to finish.
9. All errors in CCGLator/CharWars should vanish and building the project should succeed.

Note: Most additional errors in other Cute projects get resolved by setting an API baseline under Window → Preferences → Plug-in Development → API Baselines. e.g. choose the "Running Platform".

Select "Close Project" in the right-click menu on projects which still have errors.

C.3 Coding

To run or debug an Eclipse instance with the plug-in, right click on the CCGLator or CharWars project and choose "Run As → Eclipse Application" or "Debug As → Eclipse Application".

Below is a quick overview of the important parts of the project.

C.3.1 plugin.xml

In this file the newly added checkers and quick fixes have to be defined so that Eclipse can find these rules and provide them.

C.3.2 Checkers, Visitors and Quick Fixes

For every rule these three kinds of classes are needed. These have to be implemented and are the core work of an extension. In the checker class we define rule specific information like "Rule-NR", "problem-ID", "Ignore-String", "Profile-Group" and which visitor to use.

C.3.3 ASTHelper, ASTFactory

Functions which are used over different rules have to be extracted into these classes to avoid code duplication.

C.3.4 Testing

See chapter 4.9 for an overview of the testing framework used.

C.4 Maven

The plug-in can be built with the Java build management tool Maven [Foub]. The command "mvn install" builds, test and packages the plug-in resulting in a archive file. In it the update site is found which can be used by any Eclipse installation.

C.5 Continuous Integration Server

The Continuous Integration Server is using a Jenkins docker container. This was preinstalled in the virtual server which was provided by the HSR.

Additionally, we installed the following packages using the "pacman -S" command:

- **libxtst** [Fouc]
Missing package needed by the Eclipse environment
- **xorg-server-xvfb** [Foud]
A dummy X-Server which enables the server to run the JUnit tests without a real desktop environment.

In Jenkins we had to install the xvfb Jenkins plug-in, add the git project, select the (updated) Maven [Foub] pom.xml (in the ch.hsr.ifs.cute.ccglator.parent package), enable the xvfb plug-in in the project.

We defined it to build the project when a webhook containing the branch name "master" gets called. In the HSR git SCM Manager we defined the following webhook url pattern:

`http://sinv-56012.edu.hsr.ch/jenkins/job/CCGLadiator/build?token=build${first.branches}`

Correctly configured the current plug-in version can be installed by using the generated update site package: (Figure 32)

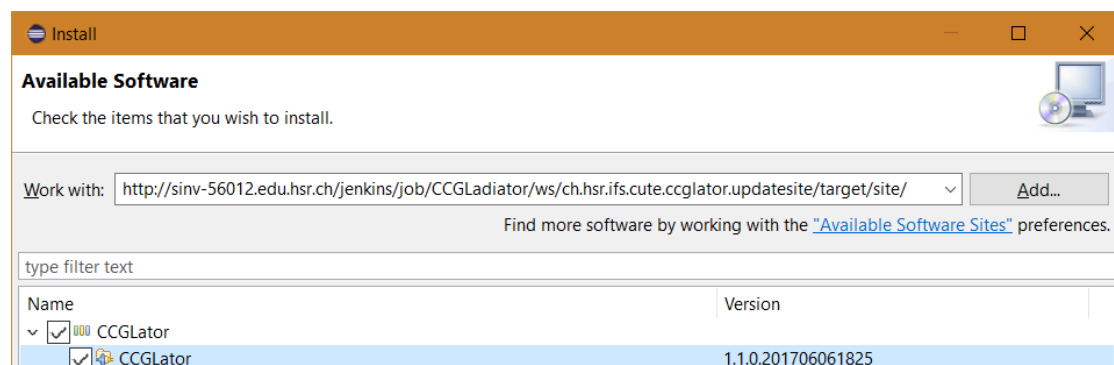


Figure 32: Using the update site generated by jenkins

Glossary

abstract syntax tree (also **AST**) The tree structure representing the written code. The contained information is normalised and supplemented by additional information like bindings. XIII, XIV, 13, 14, 16, 21, 25, 50, 52, 53, 56, 57, 59, 61–63, 77

binding A reference from one name to others. XIII, XIV, 14, 54, 61, 77

C++ Core Guidelines A set of rules for better quality of code. See chapter 2.1 and [SS15]. i–iii, XIII, 6, 10, 28, 64, 72, 76

CCGLadiator Our term project. Further development of CCGLator. See [BD16]. 37

CCGLator Eclipse plug-in developed by Kaya and Schmidiger in their bachelor thesis. See chapter 1.1.1 and [zKS16]. i–iii, I–III, VIII, IX, XIII, 6, 8–10, 30, 37, 63, 72

CCGLImperator This bachelor thesis.

CDT Testing Tools Plug-in by the IFS which provides the JUnit testing infrastructure for checkers and quick fixes. See chapter 4.9 and [fSR16a]. 66, 67

Cevelop A version of Eclipse CDT by the IFS. See chapter 1.2.1 and [fSR16b]. XIII, 6, 7

CharWars Cevalop plug-in developed by Suter and Gonzalez in their bachelor thesis. See chapter 1.1.2 and [SG14]. I, III, VIII, IX, XIII, 6, 8, 9, 30, 33, 69, 72, 73, 76

checker Analyses the AST for a specific problem. Implemented using a visitor which gets called on the necessary node types. I, IX, XIII, XIV, 7–9, 12, 14–19, 25, 34, 36–38, 40, 43, 49, 52, 53, 60, 61, 63–68, 71, 72, 74, 77

codan The part of Eclipse CDT which provides basic infrastructure for checkers and quick fixes. VII, 25, 49, 50, 67, 73

Cute The plug-in collection containing CharWars/GslAtrPtr managed by the IFS. VIII

Eclipse An IDE. III, VII–X, XIII, 6, 50

Eclipse CDT (also **CDT**, **Eclipse C/C++ Development Tooling**) Eclipse package for C++. See chapter 1.2 and [Foua]. i, ii, XIII, 6, 7, 25, 37, 49–51, 76

Elevenator Cevalop plug-in developed the IFS. See chapter 5.1.3 and [fSR17]. 71

GslAtrPtr Bachelor thesis of Geisseler and Meier. Further development of CharWars. See chapter 1.1.2 and [GM16]. i–iii, VI, XIII, 6, 8, 9, 30–32, 35, 69, 72, 76

Guideline Support Library (also **GSL**) A library used for some C++ Core Guideline rules. See chapter 2.1.1 and [Mic16]. 11, 30, 31, 69

IFS Institute for Software. XIII, 7, 66, 67

- index** Contains information about which binding corresponds to which locations in which source code files. 15, 16, 54, 57, 62, 64, 77
- integrated development environment** (also **IDE**) Software package for developing programs. VII, XIII, 6, 7
- JUnit test** unit testing suite for Java to test written code. X, 50, 61, 64, 67, 68
- macro** Some code part which is given a name and gets inserted everywhere where that name gets used by the preprocessor while compiling. 16, 27, 52, 53, 61, 63
- marker** A problem reported to a specific node in the AST. The corresponding code gets highlighted with squiggly lines and an icon on the left. VI, XIV, 12–14, 16, 17, 36, 40, 43, 44, 47, 49, 52–54, 57, 63, 64, 66–68
- plug-in** A software package supplementing an existing program. i–iii, I–III, VII–X, XIII, XIV, 6–12, 25, 30, 31, 33–37, 63, 70–72, 76
- problem** (also **codan problem**) One specific issue type. Has one ID, a settings entry with optionally additional preferences, a checker and can have one or more quick fixes. IX, XIII, XIV, 7, 8, 36, 64, 65, 67, 68, 72–74, 76
- quick fix** (also known as **resolution**) A method of automatically fixing a marked issue in source code. Most often a refactoring. Provided to the programmer by a plug-in. Implemented by modifying or replacing nodes in the AST. i, ii, I, VI, IX, XIII, XIV, 6, 7, 9, 10, 17, 18, 20–23, 25, 27, 28, 32–38, 40, 43–45, 47, 49, 50, 52, 53, 56, 57, 59–61, 64–70, 72, 74, 76
- refactoring** Changing the source code of a program without modifying the perceived behaviour in order to improve the structure of the program. XIV, 8, 9, 18, 20, 22, 31, 33, 35–37, 74
- translation unit** (also **TU**) One file in a C++ programm. As a node in the AST it is the root node of the tree. 14, 27, 53, 61, 62

References

- [BD16] Rolf Bislin and Kilian Diener. CCGLadiator - C++ Core Guidelines Rules Checker and Quick Fixes. <https://eprints.hsr.ch/551/>, 2016. [Online; accessed 7-April-2017].
- [Bis17a] Rolf Bislin. ASTWriter’s StatementWriter does not write Attributes for some Nodes like IASTForStatement. https://bugs.eclipse.org/bugs/show_bug.cgi?id=514684, 2017. [Online; accessed 9-May-2017].
- [Bis17b] Rolf Bislin. Change 94351 (for ”ASTWriter’s StatementWriter does not write Attributes for some Nodes like IASTForStatement”). <https://git.eclipse.org/r/94351>, 2017. [Online; accessed 9-May-2017].
- [Bis17c] Rolf Bislin. Change 94361 (for ”Codans Case Break Checker ignores the new C++17 fallthrough attribute”). <https://git.eclipse.org/r/94361>, 2017. [Online; accessed 9-May-2017].
- [Bis17d] Rolf Bislin. Change 95765 (for ” Codans Case Break Comment QuickFix does not have JUnit tests”). <https://git.eclipse.org/r/95765>, 2017. [Online; accessed 9-May-2017].
- [Bis17e] Rolf Bislin. Codans Case Break Checker ignores the new C++17 fallthrough attribute. https://bugs.eclipse.org/bugs/show_bug.cgi?id=514685, 2017. [Online; accessed 9-May-2017].
- [Bis17f] Rolf Bislin. Codans Case Break Comment QuickFix does not have JUnit tests. https://bugs.eclipse.org/bugs/show_bug.cgi?id=515814, 2017. [Online; accessed 9-May-2017].
- [Bis17g] Rolf Bislin. Pull request ”Support modifying ProblemPreferences” for CDT-Testing. <https://github.com/IFS-HSR/ch.hsr.ifs.cdtesting/pull/10>, 2017. [Online; accessed 25-May-2017].
- [Blo10] Ken Bloom. Answer to ”What is wrong with using goto?” (1). <http://stackoverflow.com/a/3517763>, 2010. [Online; accessed 7-April-2017].
- [bta10] bta. Answer to ”What is wrong with using goto?” (2). <http://stackoverflow.com/a/3517765>, 2010. [Online; accessed 7-April-2017].
- [Cor17a] Thomas Corbat. Change 96102 (for ” Codans Case Break Comment QuickFix does not have JUnit tests”). <https://git.eclipse.org/r/96102>, 2017. [Online; accessed 9-May-2017].
- [Cor17b] Thomas Corbat. Change 96567 (for ”ASTWriter’s StatementWriter does not write Attributes for some Nodes like IASTForStatement”). <https://git.eclipse.org/r/96567>, 2017. [Online; accessed 9-May-2017].
- [Dij68] Edsger W. Dijkstra. Go-to statement considered harmful. <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF>, 1968. [Online; accessed 7-April-2017].
- [Foua] Eclipse Foundation. The Eclipse Project Downloads. <http://download.eclipse.org/eclipse/downloads/>. [Online; accessed 15-June-2017].
- [Foub] The Apache Software Foundation. Apache maven project. <https://maven.apache.org/>. [Online; accessed 15-June-2017].

- [Fouc] X.Org Foundation. libxtst 1.2.3-1. https://www.archlinux.org/packages/extra/x86_64/libxtst/. [Online; accessed 16-June-2017].
- [Foud] X.Org Foundation. xorg-server-xvfb 1.18.4-1. https://www.archlinux.org/packages/extra/x86_64/xorg-server-xvfb/. [Online; accessed 15-June-2017].
- [Fou11] Eclipse Foundation. Api Documentation for Eclipse CDT. <http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.cdt.doc.isv%2Freference%2Fapi%2Foverview-summary.html>, 2011. [Online; accessed 07-June-2017].
- [Fou17a] Standard C++ Foundation. Standard for Programming Language C++. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf>, 2017. [Online; accessed 12-June-2017].
- [Fou17b] The Eclipse Foundation. CDT/designs/StaticAnalysis. <https://wiki.eclipse.org/CDT/designs/StaticAnalysis>, 2017. [Online; accessed 7-April-2017].
- [Fou17c] The Eclipse Foundation. Eclipse CDT (C/C++ Development Tooling). <https://eclipse.org/cdt/>, 2017. [Online; accessed 7-April-2017].
- [FS17] Fish-Shell. Fish-Shell Github Project. <https://github.com/fish-shell/fish-shell>, 2017. [Online; accessed 02-March-2017].
- [fSR16a] Institute for Software Rapperswil. Updatesite for CDT-Testing tools released by IFS. <https://www.cevelop.com/cdt-testing/neon/>, 2016. [Online; accessed 15-June-2017].
- [fSR16b] Institute for Software Rapperswil. Homepage of the Cevlop IDE. <https://cevelop.com>, 2016. [Online; accessed 15-June-2017].
- [fSR16c] Institute for Software Rapperswil. Homepage of the Institute for Software Rapperswil. <https://ifs.hsr.ch>, 2016. [Online; accessed 15-June-2017].
- [fSR17] Institute for Software Rapperswil. Website of the Elevenator plug-in. <http://cute-test.com/projects/cute/wiki/Elevenator>, 2017. [Online; accessed 13-April-2017].
- [GM16] Elias Geisseler and Philipp Meier. GslAtorPtr - C++ Core Guidelines Pointer Checker and Support Library Refactorings. <https://eprints.hsr.ch/528/>, 2016. [Online; accessed 15-June-2017].
- [gs] git scm.com. Homepage of Git. <https://git-scm.com/>. [Online; accessed 15-June-2017].
- [Lav] Stephan T. Lavavej. MinGW Distro - nuwen.net. <https://nuwen.net/mingw.html>. [Online; accessed 15-June-2017].
- [Lib17] Boost Library. Implementation of string_view in the boost library. http://www.boost.org/doc/libs/1_63_0/boost/utility/string_view.hpp, 2017. [Online; accessed 12-April-2017].
- [Mac16] Neil MacIntosh. Paper on the design of string_span. <http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0123r1.pdf>, 2016. [Online; accessed 31-March-2017].
- [mgk11] mgkrebbs. Answer to "Should I fix typos/grammatical errors in quotation?". <https://english.stackexchange.com/a/16623>, 2011. [Online; accessed 18-May-2017].

-
- [Mic16] Microsoft. Github repository for the Guideline Support Library. <https://github.com/Microsoft/GSL>, 2016. [Online; accessed 15-June-2017].
- [Ora] Oracle. Java SE Downloads. <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. [Online; accessed 15-June-2017].
- [s⁺09] shsteimer et al. Answer to "GOTO still considered harmful?" (1). <http://stackoverflow.com/a/47472>, 2009. [Online; accessed 7-April-2017].
- [Sch] Christian Schenk. Download MiKTeX. <https://miktex.org/download>. [Online; accessed 15-June-2017].
- [SCH⁺16] Peter Sommerlad, Thomas Corbat, Marcel Huber, et al. Cevlop on Windows. <https://wiki.ifs.hsr.ch/CPlusPlus/ExW1#6>, 2016. [Online; accessed 15-June-2017].
- [SG14] Toni Suter and Fabian Gonzalez. CharWars Rise of the fallen strings: Replace C-String Library calls with C++ std::string Operations. <https://eprints.hsr.ch/373/>, 2014. [Online; accessed 06-June-2017].
- [SS15] Bjarne Stroustrup and Herb Sutter. C++ Core Guidelines. <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>, 2015. [Online; accessed 15-June-2017].
- [sta16] stackoverflow. When should static_cast, dynamic_cast, const_cast and reinterpret_cast be used? <http://stackoverflow.com/a/332086>, 2016. [Online; accessed 15-June-2017].
- [Tea] The GIMP Team. GIMP - GNU Image Manipulation Program. <https://www.gimp.org/>. [Online; accessed 15-June-2017].
- [vdZ⁺] Benito van der Zander et al. TeXstudio. <http://www.texstudio.org/>. [Online; accessed 15-June-2017].
- [Wal08] Rob Walker. Answer to "GOTO still considered harmful?" (2). <http://stackoverflow.com/a/46638>, 2008. [Online; accessed 7-April-2017].
- [zKS16] Özhan Kaya and Kevin Schmidiger. CCGLator - C++ Core Guidelines Constructor Rules Checker and Quick-fixes. <https://eprints.hsr.ch/522/>, 2016. [Online; accessed 15-June-2017].