

CCGLadiator

C++ Core Guidelines Rules Checker and Quick
Fixes

Term Project

Department of Computer Science
University of Applied Science Rapperswil

Fall Term 2016

Authors:	Rolf Bislin, Kilian Diener
Advisors:	Peter Sommerlad, Felix Morgner
Project Partner:	IFS Institute for Software
Duration:	19.09.2016 - 23.12.2016
Workload:	240 hours, 8 ETCS / Student
Link:	http://sinv-56012.edu.hsr.ch

I Abstract

Within C++, there is a much smaller and cleaner language struggling to get out.

Bjarne Stroustrup, *The Design and Evolution of C++*

To extract this smaller and cleaner language, Bjarne Stroustrup and Herb Sutter released the C++ Core Guidelines [SS15] in 2015 which describe a multitude of rules. These enforce the use of modern C++ which is a resource-leak free, statically type-safe and overall simpler and safer language.

In this project an already existing plug-in for the Eclipse CDT environment is extended which checks written source code for violations of the rules provided in the Core Guidelines, highlights them and offers Quick Fix options to repair the faulty code in an instant.

In the scope of this project several new rules from the sections "C: Classes and Class Hierarchies" and "ES: Expressions and Statements" are added with the according Checker and Quick Fixes.

II Management summary

II.1 Introduction

Within C++, there is a much smaller and cleaner language struggling to get out.

Bjarne Stroustrup, *The Design and Evolution of C++*

To extract this smaller and cleaner language, Bjarne Stroustrup and Herb Sutter released the C++ Core Guidelines [SS15] in 2015 which describe a multitude of rules. These enforce the use of modern C++ which is a resource-leak free, statically type-safe and overall simpler and safer language. To support these Guidelines a support library called GSL [Mic16] was released by Microsoft. It offers a variety of functions and types proposed in the Core Guidelines to better support a wide set of rules.

II.2 Approach

In this project the bachelor thesis from Kaya and Schmidiger CCGLator [zKS16] is extended. It is a plug-in for the Eclipse CDT environment which checks written source code for violations of Core Guideline Rules, highlights it and offers Quick Fix options to fix the faulty code in an instant.

As a first step the plug-in structure was analysed and the functionality of such a Checker and Quick Fix had to be understood. Shortly after, the first set of rules had to be analysed which involved a lot of research into C++ itself to understand exactly why such a rule was necessary and how a fix can be offered. Additionally the means by which an AST can be traversed and edited had to be understood. Afterwards in a steady rotation new rules were analysed and implemented. In the last few weeks a large emphasis was set on fixing false positives and other bugs inside the code.

II.3 Results

In the scope of this term project 8 rules originating from the section "C: Classes and Class Hierarchies" and "ES: Expressions and Statements" were added to the plug-in with according Checker and Quick Fixes. The plug-in now advises the user for a safe style to cast, enforces a proper usage of swap functions and checks for

an improper use of variables. Any one of them represent a meaningful addition to write code in a modern and safe way.

Besides the rules a lot of helper methods were implemented simplifying future contributions to the plug-in.

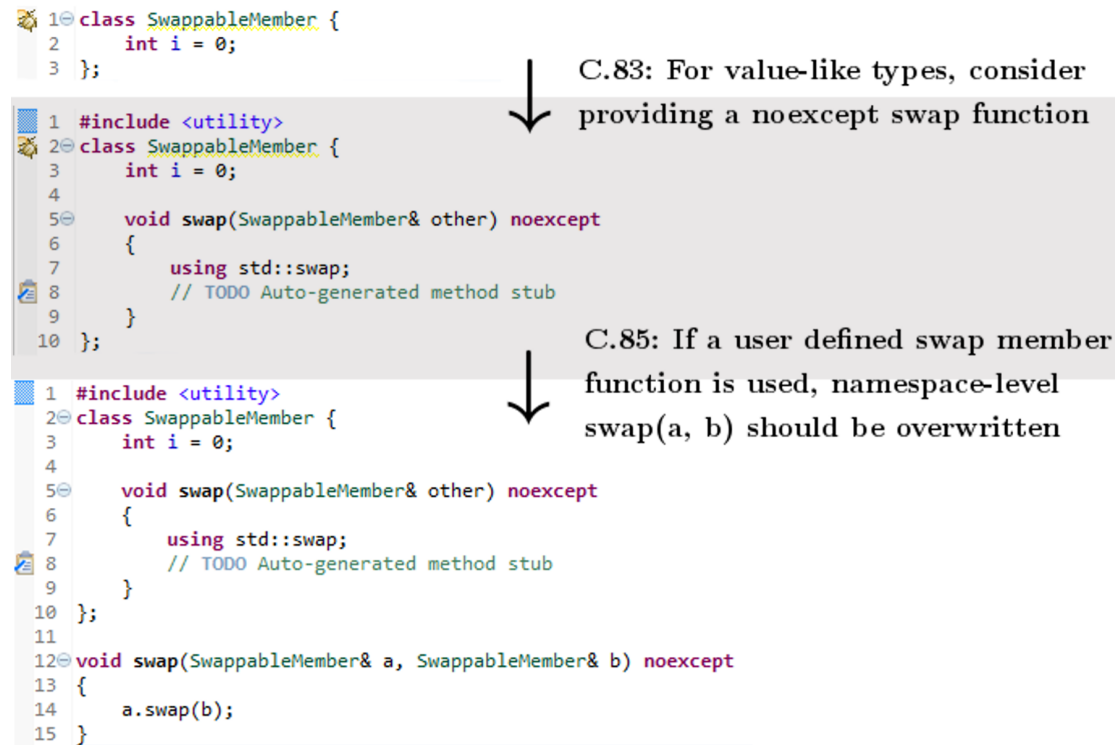


Figure 1: Execution of two Quick Fixes

II.4 Forecast

A lot of the rules defined in the Core Guidelines still remain unimplemented. For future work the addition of new rules is quite easily possible because the plug-in allows for an easy extension. Additionally in the already implemented rules some false positives are highlighted which is another option for future work.

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	Forecast	iii
III	Declaration of Authorship	iv
1	Introduction	5
1.1	Previous Work	5
1.2	Scope Definition	5
1.2.1	Minimal Scope	5
1.2.2	Optimal Scope	6
1.2.3	Maximum Scope	6
1.3	Eclipse CDT	6
1.3.1	Cevelop	6
2	Analysis	7
2.1	C++ Core Guidelines	7
2.1.1	GSL: Guideline Support Library	8
2.2	How do we recognize a swap function in the AST	9
2.3	Rule C.83: For value-like types, consider providing a noexcept swap function	10
2.3.1	Enforcement	10
2.3.2	Pre-Fix Code	11
2.3.3	Post-Fix Code	11
2.4	Rule C.84: A swap function may not fail / Rule C.85: Make swap noexcept	12
2.4.1	Enforcement	12
2.4.2	Pre-Fix Code	12
2.4.3	Post-Fix Code	12
2.5	New Rule C.85: If a user defined swap member function is used, namespace-level swap(a, b) should be overwritten	13
2.5.1	Enforcement	14
2.5.2	Pre-Fix Code	14
2.5.3	Post-Fix Code	14

2.6	Rule C.164: Avoid conversion operators	15
2.6.1	Enforcement	16
2.6.2	Pre-Fix Code	16
2.6.3	Post-Fix Code	16
2.7	ES.26: Don't use a variable for two unrelated purposes	17
2.7.1	Enforcement	17
2.8	Rule ES.46: Avoid lossy (narrowing, truncating) arithmetic conversions	19
2.8.1	Enforcement	20
2.8.2	Pre-Fix Code	20
2.8.3	Post-Fix Code	20
2.9	Rule ES.49: If you must use a cast, use a named cast	22
2.9.1	Enforcement	22
2.9.2	Pre-Fix Code	24
2.9.3	Post-Fix Code	24
2.10	ES.74: Prefer to declare a loop variable in the initializer part of a for-statement	25
2.10.1	Enforcement	25
2.10.2	Pre-Fix Code	26
2.10.3	Post-Fix Code	26
2.11	Review	26
3	Implementation	27
3.1	ASTHelper	28
3.1.1	analyseSwapFunction	28
3.1.2	isReturnType	28
3.1.3	hasConstParameter	29
3.1.4	getTypeFromExpressionElement	29
3.1.5	getTypeFromBinding	29
3.1.6	getNamespace	29
3.2	ASTFactory	30
3.2.1	newSwapFunction and newNamespaceSwapFunction	30
3.2.2	newDeclarationStatement	30
3.3	ASTComment	30
3.4	SetAttributeQuickFix	30
3.4.1	Limitation on Rule Names	31
3.4.2	Set an attribute on a IASTForStatement	31
3.4.3	Other changes	31
3.5	Rule C.83: For value-like types, consider providing a noexcept swap function	32
3.5.1	Checker	32

	To Do's	32
3.5.2	Quick Fix	32
	Add Swap Member Function	33
	Change Parameter of swap function to a reference	33
3.6	Rule C.84/85: Make swap noexcept	34
3.6.1	Checker	34
3.6.2	Quick Fix	34
3.7	New Rule C.85: If a user defined swap member function is used, namespace-level swap(a, b) should be overwritten	35
3.7.1	Checker	35
3.7.2	Quick Fix	35
3.8	Rule C.164: Avoid conversion operators	36
3.8.1	Checker	36
3.8.2	Quick Fix	36
3.9	ES.26: Don't use a variable for two unrelated purposes	37
3.9.1	Checker	37
3.9.2	Quick Fix	37
3.10	Rule ES.46: Avoid lossy (narrowing, truncating) arithmetic conver- sions	38
3.10.1	Checker	38
3.10.2	Quick Fix	39
3.10.3	ProjectIncluder and ASTModifier	40
3.10.4	To Do's	40
3.11	Rule ES.49: If you must use a cast, use a named cast	41
3.11.1	Checker	41
3.11.2	Quick Fix	41
3.11.3	To Do's	41
3.12	ES.74: Prefer to declare a loop variable in the initializer part of a for-statement	42
3.12.1	Checker	42
3.12.2	Quick Fix	42
3.13	Testing	43
3.13.1	Checker	43
3.13.2	Quick Fix	44
3.13.3	To Do's	44
	During JUnit tests the Formatter recognises template pointy brackets with typedefs as binary expression	44
	GSL Project Includer	45
	Strange space character	45

4 Conclusion 46

4.1	Result	46
4.1.1	Pull Request for C.84/C.85	46
4.2	Future Work	46
A	Project organisation	I
A.1	Approach	I
A.2	Project Plan	I
B	User Manual	II
B.1	Installation	II
B.2	Configuration	II
B.3	Usage	III
C	Developer Manual	IV
C.1	Local Development Environment	IV
C.1.1	Prerequisite	IV
C.1.2	Eclipse SDK	V
C.1.3	CCGLator Workfolder	V
C.1.4	Coding	VI
	plugin.xml	VI
	Checkers, Visitors and Quick Fixes	VI
	ASTHelper, ASTFactory	VI
	Testing	VI
C.2	Continuous Integration Server	VII
C.3	Project Management Environment	VIII
C.4	This Document	VIII
D	C++ Codes used for the AST Images	IX
	Bibliography	X

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.

1.1 Previous Work

In the last term Kaya Özhan 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 abide to these rules.

1.2 Scope Definition

In this term project the aim is to improve the already existing plug-in by implementing the later mentioned rules defined in the Core Guidelines.

1.2.1 Minimal Scope

For the Minimal scope requirements the following rules should be implemented:

- C: Classes and Class Hierarchies
 - C.83: For value-like types, consider providing a noexcept swap function
 - C.84/85: A swap function may not fail/Make swap noexcept
 - new C.85: If a user defined swap member function is used, namespace-level swap(a, b) should be overwritten.
 - C.164: Avoid conversion operators
- ES: Expressions and Statements
 - ES.26: Don't use a variable for two unrelated purposes

1.2.2 Optimal Scope

In the optimal scope the aim is to implement the following rules in addition to the minimal scope.

- ES: Expressions and Statements
 - ES.46: Avoid lossy (narrowing, truncating) arithmetic conversions
 - ES.49: If you must use a cast, use a named cast
 - ES.74: Prefer to declare a loop variable in the initializer part of a for-statement

1.2.3 Maximum Scope

If the project goes better than expected rules such as the following will be added to the scope.

- ES: Expressions and Statements
 - ES.22: Don't declare a variable until you have a value to initialize it with
 - ES.56: Write `std::move()` only when you need to explicitly move an object to another scope

1.3 Eclipse CDT

Eclipse CDT 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.

1.3.1 Cevalop

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

2 Analysis

In this chapter an overview of the C++ Core Guideline Rules [SS15] is given, we analyse what a swap function is and do a detailed analysis of the implemented rules in the scope of this project. 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 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 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

GSL (Guideline Support Library) 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 How do we recognize a swap function in the AST

The following chapters all cover some rules which have to do with swap functions. Rule C.83 (chapter 2.3) is about when to provide a member swap function, Rule C.84/85 says that such swap functions are not allowed to fail and should be declared `noexcept` (chapter 2.4) and the new Rule C.85 suggests to provide a namespace-level swap function if there is a member-swap function (chapter 2.5).

Therefore, we need to recognize a swap function for multiple rules which is described in this chapter.

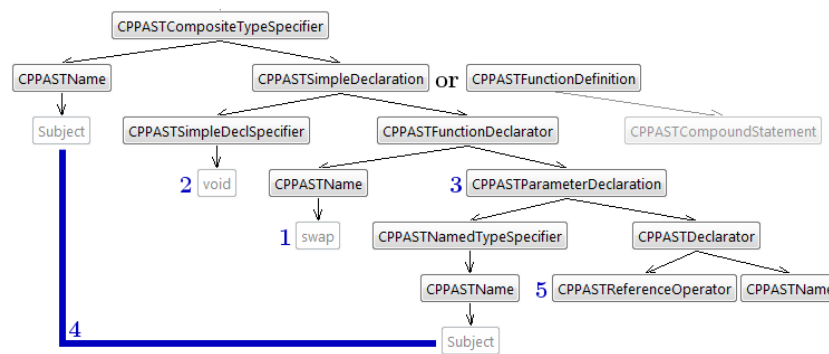


Figure 2: AST Nodes used to recognise swap functions

A swap member function is a non-const function in a class with the name "swap" [1] (see Figure 2) with the return type "void" [2] and takes exactly one non-const parameter [3] of the same class [4] as a reference [5].

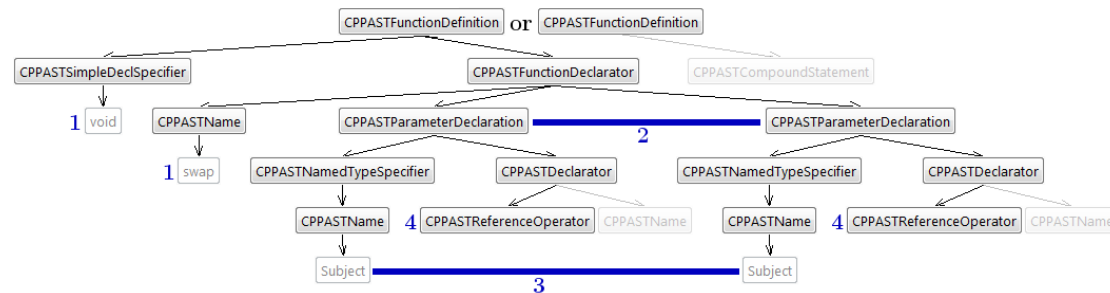


Figure 3: AST Nodes used to recognise swap functions

To a namespace level swap function nearly the same rules apply [1] (see Figure 3) but it is not in a class (or it alternatively is a friend function) and takes exactly two parameters [2] of the same class [3] as non-const reference [4].

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

To comfortably swap out two variables of value-like types a swap function can be useful. If the custom value-like type needs some special handling a noexcept swap function should be provided. The Checker of this rule should help the programmer to recognise such situations.

2.3.1 Enforcement

As stated by the C++ Core Guidelines Rule [SS15, C.83: Enforcement] a class with member variables and without virtual functions is most likely a value-like type and should provide a swap function. In the Guidelines there is a second enforcement point but that corresponds to the Rule C.84.

If we find a swap member function (as described in chapter 2.2) then the rule is respected for that class.

If the only reason that we don't find a swap function is that the function parameter is not by reference, the programmer most likely just made a mistake. Then we can, in addition to the Quick Fix below, provide a small Quick Fix to change that parameter to a reference as an alternative to a completely new swap function.

The rule does not apply to classes where there are one or more virtual functions present (see Figure 4).

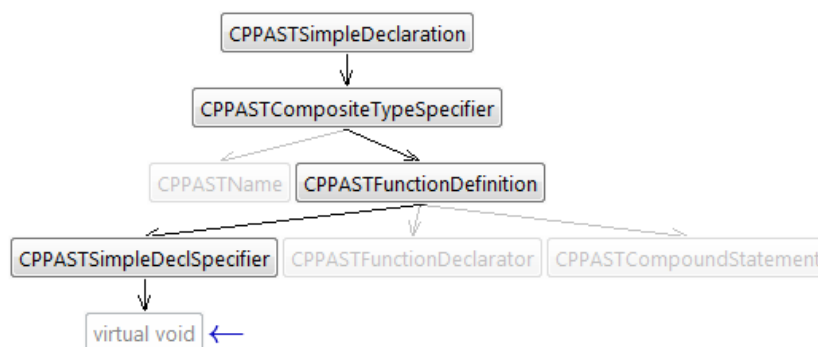


Figure 4: AST Nodes used to enforce C.83

These are the checks we used to implement this rule (see chapter 3.5) but currently there are too many false positives. That means there are a lot of classes which get marked but where the `std::swap()` is optimal. A possible way to improve the rule might be to only mark classes where the types of the member variables provide a swap function.

2.3.2 Pre-Fix Code

```
1 struct A {  
2     void swap(A& a);  
3 };  
4 struct Subject {  
5     A a;  
6  
7  
8     /* no swap and  
9      *  
10    * no virtual function */  
11 };
```

Listing 1: C.83 Pre-Fix

2.3.3 Post-Fix Code

```
1 struct A {  
2     void swap(A& a);  
3 };  
4 struct Subject {  
5     A a;  
6     void swap(Subject& other) noexcept {  
7         using std::swap;  
8         // TODO Auto-generated method stub  
9     }  
10    /* no virtual function */  
11 };
```

Listing 2: C.83 Post-Fix

2.4 Rule C.84: A swap function may not fail / Rule C.85: Make swap noexcept

Because these two rules are so similar we treat them as one rule. For the Rule C.85 we defined a new one which can be found under (2.5) and a pull request was provided to the Core Guidelines (see chapter 4.1.1).

These two rules can be summarised into one because they treat the same problem. In addition they're pretty straightforward to understand and enforce. If there is a swap function present, it should always be declared noexcept so it's guaranteed to succeed or the program terminates. If this isn't enforced, other functions which use the swap function may operate on a broken state.

2.4.1 Enforcement

As explained in chapter 2.2 it is needed to check for three different kinds of swap function. If one is found, the Quick Fix allows to automatically add the "noexcept" specification to the function. This can be set in the "CPPASTFunctionDeclarator" node.

2.4.2 Pre-Fix Code

```
1 struct swappableMember {  
2     int i;  
3 };  
4  
5 void swap(swappableMember& a,  
6           swappableMember& b)  
7 {  
8     auto tmp = a;  
9     a = b;  
10    b = tmp;  
11 }
```

Listing 3: C.84/85 Pre-Fix

2.4.3 Post-Fix Code

```
1 struct swappableMember {  
2     int i;  
3 };  
4  
5 void swap(swappableMember& a,  
6           swappableMember& b) noexcept  
7 {  
8     auto tmp = a;  
9     a = b;  
10    b = tmp;  
11 }
```

Listing 4: C.84/85 Post-Fix

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

This rule is defined by us because C.84/85 can be merged into one rule and it represents an useful addition to the swap rules.

If a swap() member function exists there should also be a swap(a, b) implementation for the same member type in the namespace. This enables the Argument Dependent Lookup (ADL) to choose the best fitting function for the call. This makes it possible to implement a specialised swap function which can improve the swap process based on the non-static data members of the respective class.

For example, it may be enough to only swap a member field and not the whole class or the cache of the class can be disregarded for the swap process, resulting in a slimmer swap function.

Additionally a lot of algorithm functions use the swap(a, b) syntax which would as well use the namespace-level one.

```
1 int main() {  
2     using std::swap;  
3     ownSwap::swappableMember a {1};  
4     ownSwap::swappableMember b {2};  
5     swap(a, b); //ownSwap::swap(a,b) is used because of ADL.  
6     return 0;  
7 }
```

Listing 5: C.85 ADL lookup example

This rule will be deprecated if the Unified Call Syntax is standardised [Sut16].

2.5.1 Enforcement

As the first step, classes are searched for a member swap function. If there is one present, a friend function is searched in the same class. If this doesn't succeed the namespace of this function is searched for a swap(a, b) function. Because a namespace can be defined over multiple files, it is necessary to search the index for a matching swap function. If there is no swap function found, the Quick Fix adds a swap function with the matching parameters.

2.5.2 Pre-Fix Code

```
1 namespace ownSwap {
2 struct swappableMember {
3     auto a;
4     void swap(swappableMember &other)
5         noexcept {
6         auto temp = a;
7         a = other.a;
8         other.a = temp;
9     }
10 };
11 /* no namespace-level swap function for
12    swappableMember */
13
14 }
```

Listing 6: C.85 Pre-Fix

2.5.3 Post-Fix Code

```
1 namespace ownSwap {
2 struct swappableMember {
3     int a = 0;
4     void swap(swappableMember &other)
5         noexcept {
6         int temp = a;
7         a = other.a;
8         other.a = temp;
9     }
10 };
11 void swap(swappableMember& a,
12           swappableMember& b) noexcept {
13     a.swap(b);
14 }
```

Listing 7: C.85 Post-Fix

2.6 Rule C.164: Avoid conversion operators

Implicit conversions can be surprising which this simple example demonstrates:

```
1 struct A;
2 struct B {
3     B()=default;
4     B (const A& x) {}
5 };
6 struct A {
7     operator B() { return B();}
8 };
9 int main () {
10     A foo;
11     B bar = foo; // conversion operator
12     B baz {foo}; // conversion constructor
13     bar = foo; // conversion operator
14 }
```

Listing 8: C.164 Example

In line 13 as well as line 15 the conversion operator, in line 14 however the conversion constructor gets called. If there was more code between line 13 and 15 the programmer might even have overlooked that foo and bar are of two different classes and might not realize that a conversion takes place. (Additionally usage of conversion operators and conversion constructors may lead to inconsistent behaviour.)

If however the conversion operator as well as the conversion constructor (by Rule C.46) get declared `explicit`, the programmer has to convert the object consciously and will notice if something is amiss.

2.6.1 Enforcement

If a non-explicit conversion operator is found (See CPPASTConversionName in Figure 5) the function should be flagged with a Tooltip which encourages the removal of the function.

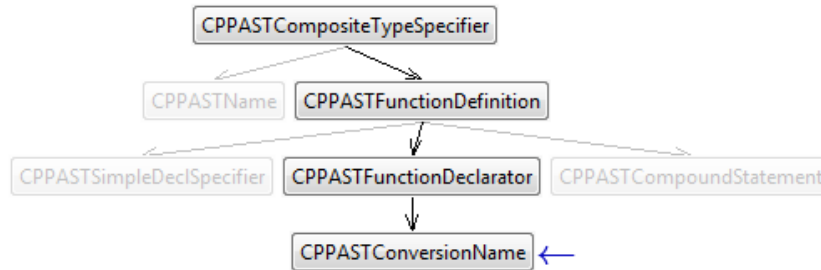


Figure 5: CPPASTConversionName in the AST

As an alternative it might be viable for the programmer to apply a provided Quick Fix to make the function explicit.

2.6.2 Pre-Fix Code

```

1 struct To {};
2
3 struct From {
4     operator To*() {
5         return new To();
6     }
7 };
  
```

Listing 9: C.164 Pre-Fix

2.6.3 Post-Fix Code

```

1 struct To {};
2
3 struct From {
4     explicit operator To*() {
5         return new To();
6     }
7 };
  
```

Listing 10: C.164 Post-Fix

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

To improve the readability of written code, variables should not be recycled. This behaviour only leads to confusion if somebody tries to understand the code.

```
1 void function() {
2   int i;
3   i = 3;
4   i = 5;  //bad: i recycled
5 }
```

Listing 11: ES.74 Example

2.7.1 Enforcement

To enforce this rule, all usages of a variable can be received from the translation unit [1]. Afterwards each usage must be checked if it is an assignment to the variable or if it is incremented or decremented. As soon as two usages are found [2], the variable is highlighted.

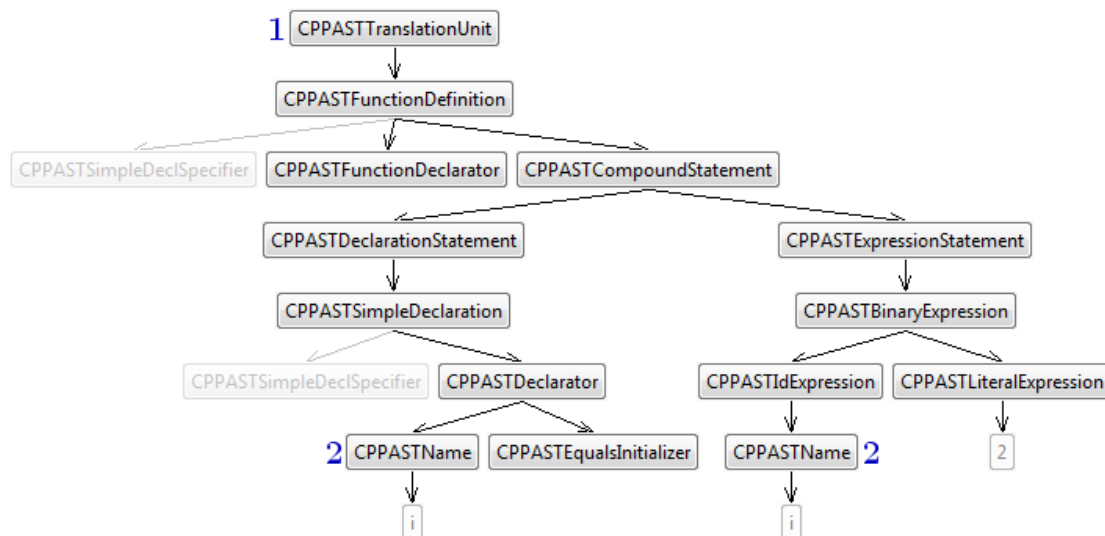


Figure 6: AST Nodes used to check for a recycled variable

But if the first reference [1] is inside an if-statement [2] and the second reference [3] inside the else-statement of the same if-statement [4] it's not a recycled variable and won't be highlighted.

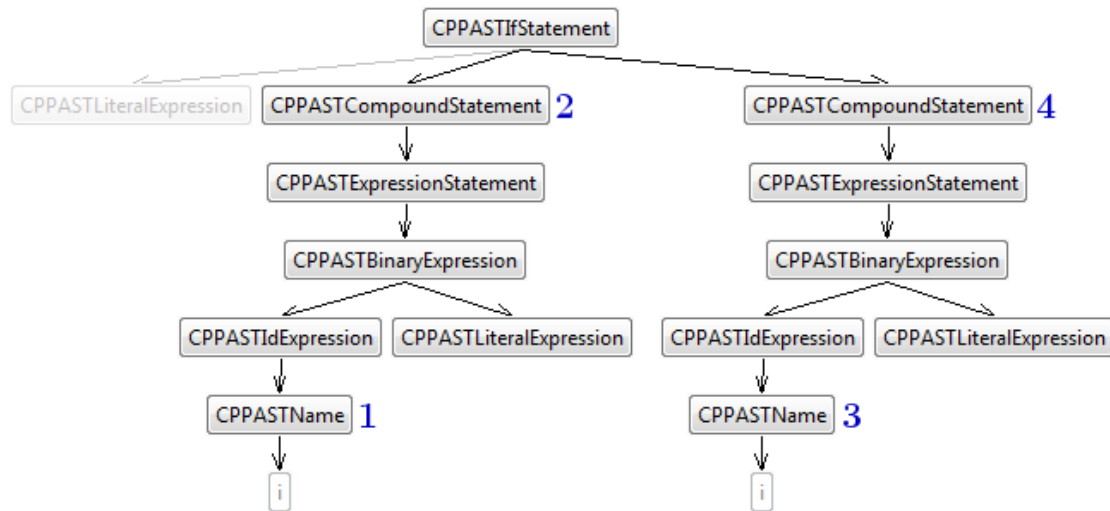


Figure 7: AST Nodes used to check for a recycled variable inside an if-statement

This rule is prone to find a lot of false positives. For this purpose the rule should be implemented cautiously and only report the clear cut cases.

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

In lossy arithmetic conversions, like assigning 7.9 to an integer, the loss of information (here '.9') is often by accident. Therefore, this rule states that such (automatic) conversions should be avoided, except by explicitly using `narrow()` or `narrow_cast()`. The difference between the two is that `narrow()` throws an exception if you really lose information. (But not for example if you cast 7.0 to `int`)

Now we need to know what cases there are in which lossy arithmetic conversions can occur and what kind of lossy conversions there are.

Lossy arithmetic conversions can occur at:

- (Assignment) Declarations
- Assignments
- C-style Casts
- Function Arguments

The corresponding chapters in the C++ Standard [Fou16b, "Integral promotions" [conv.prom] and "Numeric conversions" [string.conversions]] are tricky to understand but the `cppreference.com` page "Fundamental types" [cpp] is quite helpful. Lossy conversions on function arguments are most likely unintended. For all other cases lossy conversions are used quite often in a reasonable way. We should only flag the less likely reasonable conversion types (by default):

- loss of floating point
All or at least the following:
 - `float` → `char`
 - `double` → `int`
- conversion from integer to `char`
 - (unsigned) `long` → `char` (or `char16_t` or `char32_t`)
 - (unsigned) `long long` → `char` (or `char16_t` or `char32_t`)

All others in this category are too common to flag (by default).

The following conversion types are common and therefore we do not flag them (by default):

- loss of floating point precision
- loss of integer range size
- loss of signed attribute
- narrowing char conversions

2.8.1 Enforcement

Because lossy conversions can occur in multiple different situations we have to check multiple different node types for it.

(Assignment) Declarations	To recognise lossy conversions in (assignment) declarations we need to check the declaration type [1] (See Figure 8) and the type of the CPPASTEqualsInitializer node [2].
Assignments	To recognise assignments we need to check the operand types of CPPASTBinaryExpression nodes [3] with operator type "assignment" (values 17-27) and check the types of the two expression elements [4].
C-style Casts	Casts have the class CPPASTCastExpression [5]. it contains a TypeId [6] to the target casting type.
Function Arguments	On a CPPASTFunctionCallExpression [1] (See Figure 9) we have to check the provided [2] and the requested [3] arguments.

2.8.2 Pre-Fix Code

```

1 /* no special include */
2 void foo(int i) {}
3 int main(){
4     double d = -7.0;
5     unsigned u = 0;
6     foo(d);
7     u = d;
8 }
```

Listing 12: ES.46 Pre-Fix

2.8.3 Post-Fix Code

```

1 #include <gsl>
2 void foo(int i) {}
3 int main(){
4     double d = -7.0;
5     unsigned u = 0;
6     foo(gsl::narrow<int>(d));
7     u = gsl::narrow_cast<unsigned>(d);
8 }
```

Listing 13: ES.46 Post-Fix

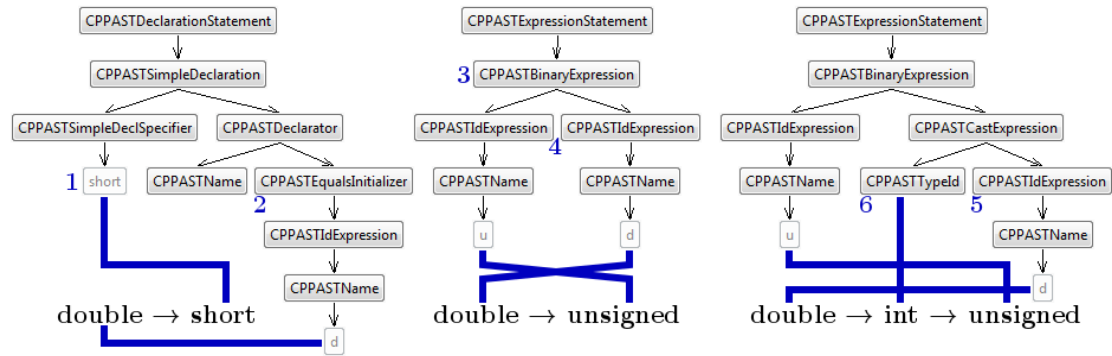


Figure 8: AST Nodes used for checking for narrow casts

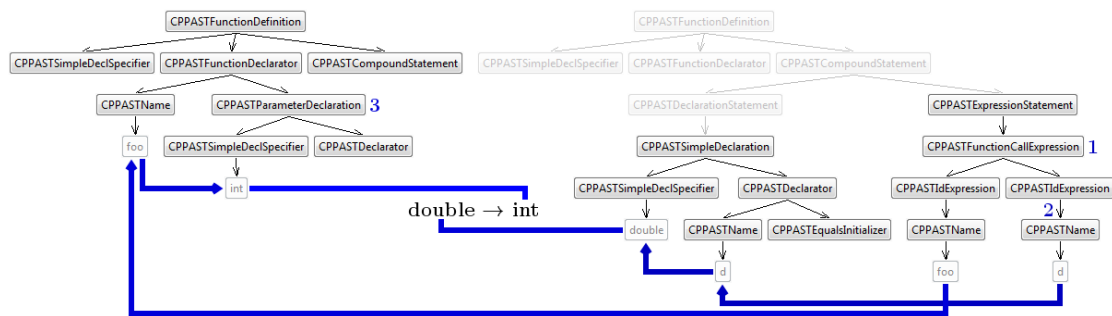


Figure 9: AST Nodes used for checking function arguments for narrow cast

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

Named casts are more specific and allow for better discoverability of some errors. Therefore, old C-style and function-style casts shouldn't be used anymore.

Something noteworthy is the following description of how C-style casts and function-style casts are handled in C++:

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]

2.9.1 Enforcement

Any `IASTCastExpression` [1] with the operator-type "op_cast" can be flagged.

Function-style cast of basic types are `ICPPASTSimpleTypeConstructorExpression` nodes [2] with an `IASTSimpleDeclSpecifier` node [3] as `IASTDeclSpecifiers`.

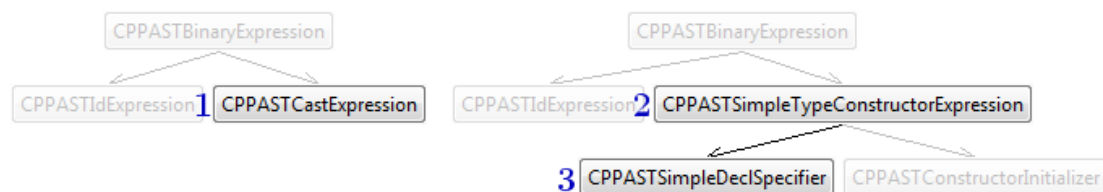


Figure 10: AST Nodes used to check for casts

Below is a list with the named casts mentioned by the C++ Core Guidelines[SS15, ES.49: Reason]. For the first four of them we provide Quick Fixes the programmer can choose from.

(This analysis is mostly based on information from stackoverflow [sta16])

static_cast	<p>This named cast is most often used.</p> <p>It doesn't cast through virtual inheritance and it can't cast away constness or volatility.</p> <p>So casting from <code>const int *</code> to <code>int *</code> wouldn't work and would require <code>const_cast</code>. This Quick Fix will be provided for any marked cast.</p>
dynamic_cast	<p>Used for polymorphism. To cast pointers or references where the target type is a child class of the source type. It would be possible to recognise when this is the case and only then show this Quick Fix as an option.</p>
const_cast	<p>This Quick Fix is useful if either the target or source type is a const pointer or reference but not both. It wouldn't be too difficult to check for this usecase and provide this Quick Fix dependent on that.</p>
reinterpret_cast	<p>This is a dangerous cast. As the name suggests it reinterprets the binary code stored as if it was of the new type. Knowing when this could be of use is quite difficult, if not impossible, therefore this Quick Fix gets provided for any marked cast.</p>
std::move std::forward	<p>Gives an rvalue-reference. Useful for only a few usecases like swapping huge datastructures. (See [ein16] and [Pot13])</p> <p>We don't provide a Quick Fix to these two named casts because it is impossible that a C-style cast converts to an rvalue-reference.</p>
gsl::narrow gsl::narrow_cast	<p>The Quick Fixes for <code>gsl::narrow</code> and <code>lstlininegsl::narrow_cast</code> are provided by Rule ES.46 in chapter 2.8 for <code>gsl::narrow</code> and <code>gsl::narrow_cast</code> are provided by Rule ES.46 in chapter 2.8</p>

2.9.2 Pre-Fix Code

```
1 struct parent {
2     virtual int value() { return 4; } };
3 struct child : parent {
4     int value() { return 42; } };
5
6 int main() {
7     int i = 2;
8     child fortyandtwo = child{};
9     parent *four = &fortyandtwo;
10    const int *ci = &i;
11    float f = 4.2;
12
13    /* Casts */
14    long l = (long) i;
15    child *fourtytwo =
16        (child *) four;
17    int *nci = (int *)ci;
18    char *b =
19        (char *)(&f);
20 }
```

Listing 14: ES.49 Pre-Fix

2.9.3 Post-Fix Code

```
1 struct parent {
2     virtual int value() { return 4; } };
3 struct child : parent {
4     int value() { return 42; } };
5
6 int main() {
7     int i = 2;
8     child fortyandtwo = child{};
9     parent *four = &fortyandtwo;
10    const int *ci = &i;
11    float f = 4.2;
12
13    /* Casts */
14    long l = static_cast<long>(i);
15    child *fourtytwo =
16        dynamic_cast<child *>(four);
17    int *nci = const_cast<int *>(ci);
18    char *b2 =
19        reinterpret_cast<char *>(&f);
20 }
```

Listing 15: ES.49 Post-Fix

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

The variable used to initialise the for loop should be declared at the for statement. This allows code optimisers to speed up the execution [SS15, ES.74: Discussion] as well as making the code clearer to read.

```
1 int i; // BAD: i is visible outside the loop
2 for (i = 0; i < 100; ++i) {
3     // ...
4 }
5 // i is still visible here and isn't needed
```

Listing 16: ES.74 Bad Example

2.10.1 Enforcement

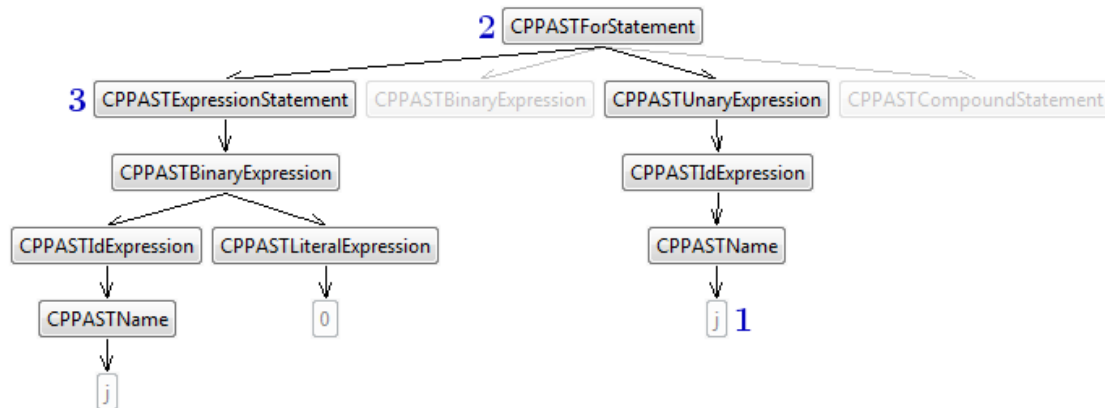


Figure 11: AST Nodes used to check for the initialiser

As a first step the loop variable must be found out. This can be done through the iteration-statement [1] of the for-statement [2]. With this variable we can search for all references of it in the whole AST. At this point a decision must be reached. If the variable is never referenced again in the whole tree, the initialisation [3] can be done in the for-loop as well, which can be done as a Quick Fix.

On the other hand if the variable is referenced somewhere in the AST, no decision can be reached if the variable can be declared in the initialiser-statement. At this point only a warning will be displayed.

Furthermore the use of multiple variables in a for loop and the possibility that the initialiser part is empty has to be considered. The case of an empty initialiser is easily checkable with minor additions to the check above but when two loop variables can be deduced the complexity needed to check for all variants is greatly increased.

2.10.2 Pre-Fix Code

```
1 void function() {  
2     int j;  
3     for (j = 0; j < 0; j++) {  
4     }  
5 }
```

Listing 17: ES.74 Pre-Fix

2.10.3 Post-Fix Code

```
1 void function() {  
2     for (int j = 0; j < 0; j++) {  
3     }  
4 }  
5 }
```

Listing 18: ES.74 Post-Fix

2.11 Review

With the implementation of these rules, the plug-in advises the user for a safe style to cast, enforces a proper usage of swap functions and checks for an improper use of variables. Any one of them represents a meaningful addition to get the code written in a modern and safe way.

3 Implementation

As a starting point, the CCGLator Bachelor Thesis [zKS16] plug-in 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 SetAttributeQuickfix
- 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 [Fou16a] providing us with the needed information about the CDT environment.

3.1 ASTHelper

In the ASTHelper methods are exposed which handle problems encountered over different rules. Extracting them into a new class minimises duplicated code and cleans-up the rules.

In addition to the methods of CCGLator, several new ones had to be implemented so that the new rules can be handled comfortably.

3.1.1 analyseSwapFunction

This method can analyse a function and identify if it is a namespace-level, friend or class swap function. If the function is neither of them, a list of reasons is returned.

We check a function for the following properties and if they apply we add them to an Enum-List:

- Does it have the wrong name?
- Is the parameter not a reference?
- Is there a parameter which is const?
- Is the function const?
- Is the return type something else than void?

If the function has the correct name we additionally check if it is a member, friend or namespace level function and if the parameter types and amounts are correct.

If the resulting list contains only one entry which is either "IsNamespaceFunction", "IsFriendFunction" or "IsMemberFunction" then the function is a valid swap function of said type.

3.1.2 isReturnType

To check the swap function for the `void` return type this method is used. As a parameter it takes the function as well as an integer value representing the return type. For resolution from an int to the actual type, refer to the IASTSimpleDeclSpecifier [Fou16a] interface. When the type requested equals the return type from the supplied function, the method returns the value `true`.

3.1.3 hasConstParameter

The `analyseSwapFunction` has to check if all the parameters of a function are `const`. This method allows us to do that. Based on the `IASTFunctionDeclarator` all the parameters are analysed. As soon as one of the function parameters is `const`, the value `true` is returned.

3.1.4 getTypeFromExpressionElement

This method is used by the ES.46 Checker and Quick Fix. It returns the `IType` of an `IdExpression` of the `IdExpression` inside one or multiple convoluted `IASTCastExpressions`. Additionally it can fill a provided list with all `TypeIds` of found `IASTCastExpressions` such that in the end there is a complete list of intermediate casting steps.

3.1.5 getTypeFromBinding

Every name in the program code (e.g. of variables) has an `IBinding` object. It contains where in the code the exact same element appears. Textually equal names but of different elements have different bindings. This function tries to convert `IBindings` to either an `ICPPMember` or an `ICPPVariable` in order to be able to get the `IType` from the binding. It is used by the ES.46 Checker and Quick Fix as well as the method `getTypeFromExpressionElement`.

3.1.6 getNamespace

In the Checker as well as the Quick Fix for C.85 (see chapter 3.7), the namespace node has to be found from a given declaration. For this a loop analyses the parent of a node recursively. If a namespace is found before the `TranslationUnit` is reached, it is returned. When the declaration is inside the global namespace, no matching node is found and the value `null` is returned.

3.2 ASTFactory

The ASTFactory allows to create new ASTNodes which can be used in the Quick Fixes to change the AST to match the rule.

3.2.1 newSwapFunction and newNamespaceSwapFunction

These two methods support the creation of a new swap function either for the use inside a class or in the namespace scope. Both of them need the type from the swap-function provided as an argument.

3.2.2 newDeclarationStatement

This function supports the creation of a new declaration Statement as used in the Quick Fix for ES.74. The assignment of a value can be done with a IASTExpression containing the wished value or if none is given the value 0.

3.3 ASTComment

Because the CDT CPPNodeFactory is missing a factory function for creating IASTComment nodes we had to create a custom ASTComment node. This class extends the CDT ASTNode class and implements the IASTComment interface.

It stores a char array which you can set and read via `setComment` and `getComment`. `setComment` makes some sanity checks and automatically adds `"/"`, `"/"` and `"/"` where needed, as well as a newline at the end if necessary.

As a workaround for a CDT bug we added the `setText` function where sanity checks do not get executed. This allows us to put any text and or code on any position where a comment can be. For more information about said bug see chapter 3.4.2

3.4 SetAttributeQuickFix

When a user wishes to ignore a rule in a certain place, the possibility to add a custom attribute is provided. With it rules can be suppressed. This class handles the addition of these ignore attributes to AST Nodes. In its original state it had some limitations which we had to eliminate.

3.4.1 Limitation on Rule Names

Originally the Quick Fix could only handle rule names of the format "C.##" (where # is a digit). One of the first rules we tackled was C.164 which already did not fit in this format. The result was that the Quick Fix added an ignore attribute for Rule C.16 instead. To fix this issue we switched to a regular expression. Now we use the regex "(ES|C)\.\d+", which will need further modification if a rule from a new Guideline Section gets implemented. But this regular expression resulted in another issue.

Previously Sub-Checkers looked like "C.4601" which was Sub-Checker 01 of Rule C.46 and the ignore attribute was for "C.46". With this regex that would have resulted in an ignore attribute for "C.4601". Therefore, we changed the naming format of the rule names (including filenames) to have an underline between the rule and sub-rule numbers (e.g. "C.46_01").

3.4.2 Set an attribute on a IASTForStatement

For ES.74 the ignore attribute had to be set on a for-statement which wasn't supported by the original. After implementing it, the ignore attribute was still not set on the node. After a long trial and error phase we concluded that it wasn't a problem with the plug-in but instead with CDT not supporting it. As it turned out the problem was indeed a bug in CDT which doesn't support attributes on IASTForStatements contrary to what the C++ standard defines. In the standard an attribute is allowed on all types, variables, names and blocks [Foul6b, "Attributes" [dcl.attr]] with a for-loop representing a block. So for a workaround an IASTComment without slashes is added in front of the for-statement which contains the code for the attribute.

As soon as this bug is fixed, the related codepart can be deleted in the SetAttributeQuickFix.

3.4.3 Other changes

Other changes for this Quick Fix were removing the limitation to IASTDeclaration nodes and the proper handling for IASTCompositeTypeSpecifier, IASTCompositeTypeSpecifier, IASTDeclarator, IASTExpression and IASTStatement nodes and more minor changes.

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

As mentioned in the analysis chapter 2.2 about how to recognise swap functions as well as in chapter 3.1.1 about how that is implemented, this is one of multiple rules which require the recognition of why or why not a function is a swap function.

3.5.1 Checker

The Checker visits all IASTDeclarations and if the declaration is of a class or struct (ICPPASTCompositeTypeSpecifier) it applies multiple checks for that type:

- Does it have no member variables?
- Does it have a parent class?
- Does it have a virtual function?
- Does it already have a swap function?

If any of them are true the Checker doesn't mark anything. During the last two checks a list of functions gets generated where the only reason that it does not qualify as a swap member function is that the parameter is not by reference.

If however non of those checks apply the class gets marked as well as any found "nearly"-swap-functions.

To Do's

In order to reduce the amount of false positives for a future version of this plug-in it might make sense to only check for member variables of types which provide a swap function themselves.

3.5.2 Quick Fix

For this rule two Quick Fixes are provided. The first generates a new swap function whilst the second one fixes swap functions where the parameter is not by reference. We provide this Quick Fix because one can quite easily forget to make a parameter by reference.

Add Swap Member Function

We generate a swap member function using the CompoundStatement factory function from the CDT CPPNodeFactory. We add a `"using std::swap;"` statement and the comment `"// TODO Auto-generated method stub"` for which we had to create our own ASTComment node (see chapter 3.3). This generated swap member function we insert at the end of the class or struct using ASTRewrite.

Change Parameter of swap function to a reference

In this simple Quick Fix we get the Parameter Declarator and replace it with a copy with an added reference operator.

3.6 Rule C.84/85: Make swap noexcept

In the CCGLator several rules already check if something has to be noexcept or not. For this purpose they added a BaseNoexceptVisitor. This Checker also inherits from this class simplifying the check.

3.6.1 Checker

The Checker visits all IASTDeclSpecifier and checks if the parent node is a function definition or function declaration. Afterwards the function is analysed with the analyseSwapFunction (see chapter 3.1.1) method in the ASTHelper. If this returns that it actually is a swap function, it is additionally checked for an already existing noexcept tag. If none is found the function is highlighted.

3.6.2 Quick Fix

The Quick Fix is already implemented from the CCGLator rules. Rule C.84 is simply added into the ShouldBeNoExceptQuickFix.

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

This rule meant a lot of work because a namespace can be defined over multiple files. So the whole index has to be searched to get all the swap functions. The downside to this index search is, that swap functions inside another file can only be seen when the index is rebuilt. So the highlighting can sometimes not occur instantly, confusing the user. Sadly nothing can be done about index rebuilding because this is handled by the Eclipse framework.

3.7.1 Checker

The Checker starts off with checking if there is a user defined swap member function or a friend function present. If none is found, the Checker is finished. Otherwise it starts to search the AST and the Index for a namespace level swap function.

In the AST-check the namespace is fetched. Afterwards all the functions in this namespace are analysed (see chapter 3.1.1) and if no namespace level swap function is found, the check continues with the index search.

The index-check starts off by getting all the names which contain "swap" from the index. Every single occurrence of this name then is checked if it is a valid swap function, and if the enclosing namespace is the same as the one from the user defined swap function. If there still isn't a matching namespace swap function, the member swap function is highlighted.

3.7.2 Quick Fix

The Quick Fix creates a new swap function and adds it at the end of the AST. This function has the matching parameters inferred from the marked node and a function call to the member swap function.

3.8 Rule C.164: Avoid conversion operators

Note that this rule was the first implemented rule which needed modification to the `SetAttributeQuickFix` because the rule number was higher than 99. See chapter 3.4.1 "Limitation on Rule Names" for more information about that.

3.8.1 Checker

For each `IASTDeclarator` we check if it is not already explicit and if it has an `ICPPASTConversionName` node as a name.

3.8.2 Quick Fix

The Quick Fix is already implemented from the CCGLator rules. Rule C.164 is simply added into the `DeclareFunctionExplicitQuickFix`.

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

As apparent from the analysis, this rule is hard to implement without getting false positives. To counter this problem the rule only highlights the clear cut cases.

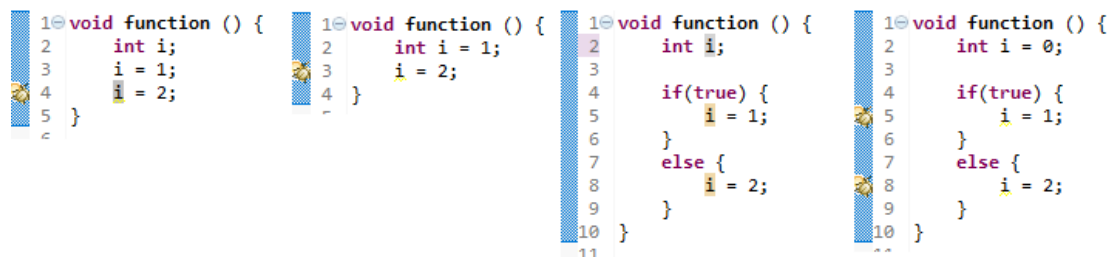
3.9.1 Checker

This Checker is executed for every declaration found in the AST. First up two counters are initialised handling the allowed and used assignments of a variable. If the declaration already assigns a value, the counter is increased.

Afterwards all declarations and references of this variable are searched. For each of these occurrences it is checked if it assigns a new value to the variable. Only in this case the counter is increased.

As soon as the counter for the used initialisations exceeds the allowed one, the usage is highlighted.

For the case that an initialisation is inside the if and one inside the else statement from the same conditional statement, the counter is decremented allowing to assign a value twice without the rule applying.



```

1 void function () {
2   int i;
3   i = 1;
4   i = 2;
5 }

```

```

1 void function () {
2   int i = 1;
3   i = 2;
4 }

```

```

1 void function () {
2   int i;
3   if(true) {
4     i = 1;
5   }
6   else {
7     i = 2;
8   }
9 }

```

```

1 void function () {
2   int i = 0;
3   if(true) {
4     i = 1;
5   }
6   else {
7     i = 2;
8   }
9 }

```

Figure 12: ES.26 Checker for different scenarios

3.9.2 Quick Fix

For this rule only the "set ignore attribute" Quick Fix is applicable because the decision what to do with a recycled variable can't be done by the plug-in. This is a task for the developer.

This rule ended up being one of the most complex rules to implement and needed multiple refactorings in order to correct some issues.

The problem-ids are divided up in 12 categories some of them disabled by default. If all of them were enabled by default there would be too many flags. Not in the sense of "false positive" but some of the lossy conversions are quite often used in a reasonable way.

Each of the 6 main categories has one variant for conversions of function arguments and one variant for all other cases. This is because according to the Guidelines, conversion of function arguments are more likely to be bad.

	Normal	Function Argument
Floating Point \rightarrow Integer	X	X
Integer (short , int) \rightarrow Char	X	X
Integer (long , long long) \rightarrow Char		X
Narrowing Integer Conversions		X
Lossy Floating Point Conversions		X
Signed \rightarrow Unsigned		X

As mentioned in the analysis chapter about this rule there are multiple cases where a lossy conversion can occur.

The IASTExpression visit function is used for detecting conversions on function arguments as well as assignment expressions (with and without casts). An IASTExpression can be a simple "op_assign". But also "op_plusAssign" and similar count as an assignment expression.

The IASTDeclarator visit function is used to detect conversions on (assignment) declarations.

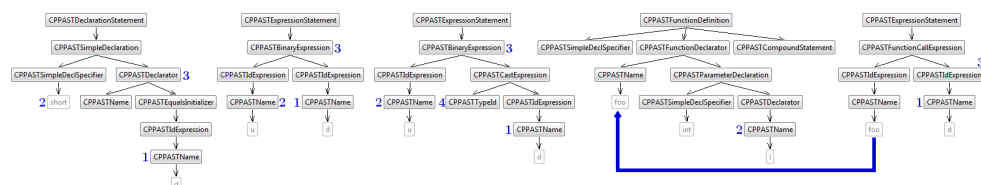


Figure 13: AST Nodes used for Checking for narrow casts

To get the type name we try to find an IASTName node [1 & 2] from which we can get the IType of it via the binding (see chapter 3.1.5). The advantage of IType is that using the CDT SemanticUtil we can resolve typedefs and recognise more lossy casts. To our knowledge IASTTypeId nodes [4] of cast expressions can not easily be converted or resolved to IType objects therefore we have to use the raw signature of those nodes.

Therefore, the found IType objects get converted to strings, normalised via a switch case and then get analysed with a big switch case. There the correct problem id gets returned or null if everything is well. (See Figure 14 to get a feeling for the size of the code block.) If no problem is found with the casting from the source to the target type and if we have intermediate casts because of cast expressions we also test each intermediate casting step.

[illegible]

Figure 14: Big switch cases of ES.46

Term Project

to collect said information is how to traverse the AST Nodes. One thing to note is the following.

If we have one `IASTCastExpression` in an `IASTBinaryExpression` we use its `IASTTypeId` as the type for the narrow cast, because that is the behaviour one would expect and equal to the behaviour of the ES.49 Quick Fixes (see chapter 3.11.2). However if there are multiple convoluted `IASTCastExpressions` we use the target type for the narrow cast. (One could try to find the narrowest type from the type list but we already spent a lot of time on this rule.)

Finally once the data in `ES46QuickFixData` is complete, we generate the cast function call with the found type and subject and then replace the old node with it. Depending on which Quick Fix got called a `narrow` or `narrow_cast` function call gets generated. As a last step the GSL (see chapter 2.1.1) header needs to be included and linked for which we use the `ProjectIncluder`.

3.10.3 ProjectIncluder and ASTModifier

Because `gsl::narrow` and `gsl::narrow_cast` are from the GSL library (see chapter 2.1.1) we need to make sure it is accessible when using this Quick Fix. Luckily there is already a solution for this by the `CharWars` Plugin from the `GslAtorPtr` project [GM16, Chapter 3.1.4] which we could reuse. The `ProjectIncluder` and `ASTModifier`

To avoid plug-in dependencies we copied the few needed classes to our project source, changed their package name and created some adapter classes. Apart from an additional file header, which states exactly this, and the changed package name, only a few lines got uncommented.

The `ProjectIncluder` checks if there is already a GSL project linked and if not creates it from plug-in resource files and links it. The `ASTModifier` helps with adding the include header to cpp files.

3.10.4 To Do's

It might make sense to search for a better solution than the big switch case mentioned above.

And as mentioned above one could try to find the most narrow type in the list of multiple convoluted cast expressions.

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

Implementing this rule based on the information collected in the analysis chapter 2.9 was quite easy.

3.11.1 Checker

In the IASTExpression visitor we just have to check if it is an IASTCastExpression with the operator type "op_cast" or if it's an ICPPASTSimpleTypeConstructorExpression with an IASTSimpleDeclSpecifier node as DeclSpecifier to find all C-style casts and function-style casts of basic types to mark them.

3.11.2 Quick Fix

The Quick Fix is quite straightforward. Get the type from the first child of the IASTCastExpression or the ICPPASTSimpleTypeConstructorExpression and the cast subject from the second child (or its child). Based on this the new cast expression can be created which then replaces the marked expression. The cast expression is either `static_cast`, `dynamic_cast`, `const_cast` or `reinterpret_cast`, based on which Quick Fix got called.

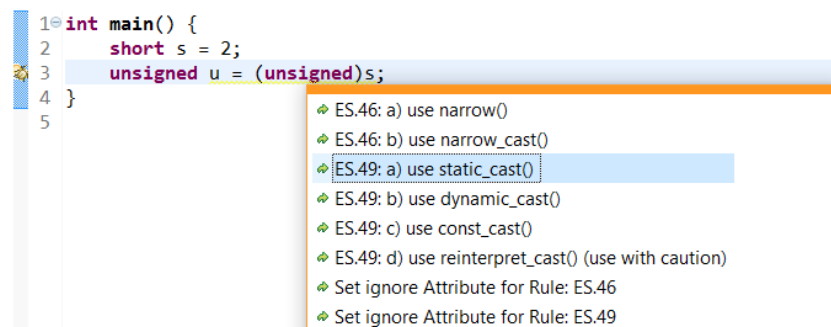


Figure 15: The order in which ES.46 and ES.49 Quick Fixes get listed

3.11.3 To Do's

In future it might be worth to try to make an algorithm which recognises the appropriate cast function and puts that at the top of the list of the Quick Fixes.

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

In this rule it was necessary to differentiate between two scenarios. In the first one the variable is assigned in the for statement but not declared, in the second one the initialiser part of the for statement is empty.

3.12.1 Checker

As a first step the loop variable has to be defined. It can be found in the iteration-expression of the loop. Once the variable is found, references of it are analysed. If all uses are assigning a new value to itself, a Quick Fix can be offered but as soon as the variable is used differently, the plug-in can't fix the problem and only the variable is highlighted.

As discussed in the analysis chapter 2.10, when two loop variables are present a lot of additional cases are created. In this project only the base case is implemented, stopping whenever more than one loop variable is found.

3.12.2 Quick Fix

The Quick Fix deletes all occurrences of the loop-variable outside the for-loop and declares the variable in the initialiser. Because a value is needed for the declaration, the last assigned value is used. If none is found the value 0 is assigned.

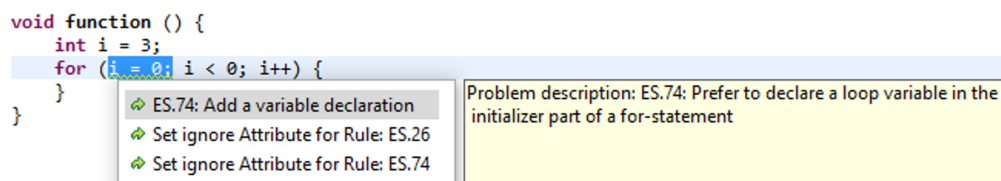


Figure 16: Quick Fix for ES.74 applicable

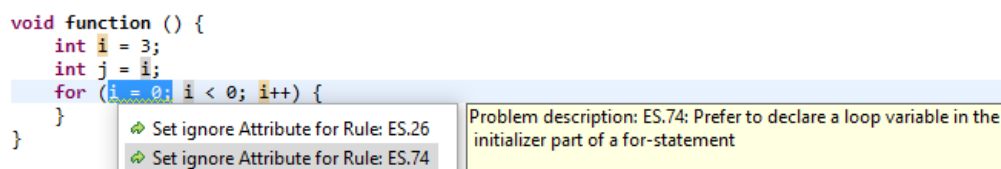


Figure 17: Quick Fix for ES.74 not applicable

3.13 Testing

In this project all the rules implemented are tested using the IFS CDT Testing Tools [fSR16a]. These tools allow an 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.

CCGLator already defined 372 tests for their own rules. In the scope of this project an additional 415 tests were implemented to fully test our own rules.

3.13.1 Checker

To test a Checker, the configuration supports a "MarkerPosition" 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.

```
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 19: Two tests for the C.85 Checker

3.13.2 Quick Fix

A similar approach is possible for the testing of 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.

```
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 20: Test for the C.85 Quick Fix

3.13.3 To Do's

There are some workarounds for some strange behaviours during JUnit tests which might get fixed in the future.

During JUnit tests the Formatter recognises template pointy brackets with typedefs as binary expression

The Quick Fix automatically issues the Code Formatter which shows some strange behaviour. For some reason in code such as `gsl::narrow_cast<myint>(f);` where "myint" is a typedef the pointy brackets (< and >) get recognised as binary expressions. The formatter then adds spaces in front and behind each of those symbols and the Quick Fix test would normally fail.

Our current solution is to just change the formatter option for such JUnit tests with typedefs.

GSL Project Includer

If using Maven the first Quick Fix where the GSL Project gets included always fails during JUnit tests. It is independent on which code gets used. When changing the order of the JUnit tests it still is the first such Quick Fix which fails. Locally in Eclipse everything works correct.

Therefore, we created an `ES46_00AvoidMavenBug` JUnit Test class where the assertions don't get executed. Independently if it would succeed or fail, it reports success. After that the JUnit Tests can execute normally.

Strange space character

In two Quick Fix tests a space character gets added where it is unexpected. We haven't found a reason for this. The two suspects are:

- `ES46_01_08AvoidLossySignedToUnsignedFunctionArgumentConversionsQuickFixTest`
`FunctionQuickFixWithExistingGSL`
- `ES46_02_08AvoidLossySignedToUnsignedFunctionArgumentConversionsQuickFixTest`
`FunctionQuickFixWithExistingGSL`

4 Conclusion

This section gives an overview of the results achieved in this project as well as pending work possible to be done in another term project or bachelor thesis.

4.1 Result

This project extended the existing plug-in with 8 new rules enforcing the correct use of swap functions, C-style casts, variable declarations and narrowing casts. For each rule, tests were written resulting in roughly 400 test cases simplifying future work. The majority of the work went into making the Checkers and Quick Fixes react correctly to all the possible, although sometimes not sensible, code variations.

Additionally the base project CCGLator [zKS16] was modified and generalised to better support the addition of new rules.

4.1.1 Pull Request for C.84/C.85

As written in the analysis for C.84 and C.85 (see chapters 2.4 and 2.5) the rule is a useful addition into the C++ Core Guidelines [SS15]. For this purpose a pull request was created [DB16], detailing the merge of C.84/C.85 and the addition of the new Rule C.85. Sadly no feedback was received until the end of this project.

4.2 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 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.

Additionally take a note of the chapters named "To Do's" in the implementation chapters of Rules C.83 (chapter 3.5.1), ES.46 (3.10.4), ES.49 (3.11.3) and Testing (3.13.3).

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

Usually we divided the rules. Each one of us dealt with a 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. In case of design or comprehension questions, we asked each other for a second opinion.

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 240 hours per student. This results in 17 hours and 15 minutes per student per week over the timespan of 14 weeks. The actual achieved time per student is:

- Rolf Bislin: 253 hours
- Kilian Diener: 220 hours

A report based on worked hours per week is found in the next diagram.

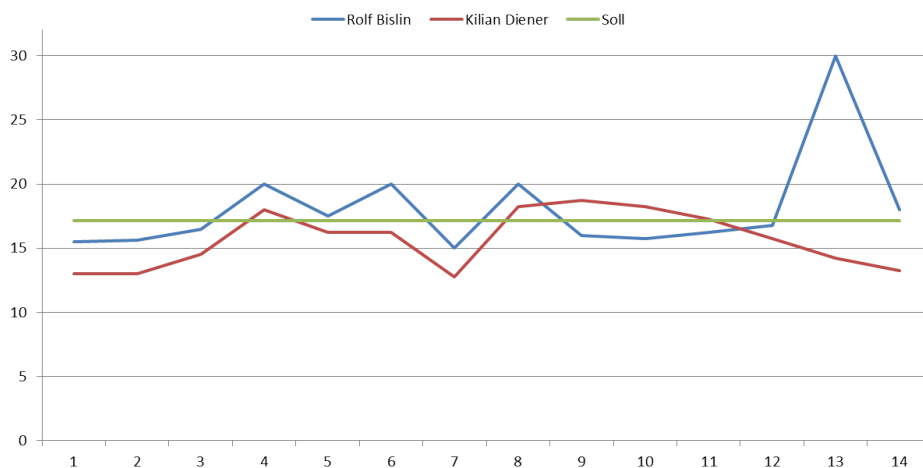


Figure 18: Planned vs. actual hours worked per week

B User Manual

Here is a quick overview on how to install, configure and use the CCGLator Plugin for Eclipse.

B.1 Installation

To use the CCGLator Plugin install it via "Help → Install New Software..." and use the Update-Site "<http://sinv-56012.edu.hsr.ch/updatesite>" or the path to the "updatesite" folder on the CD.

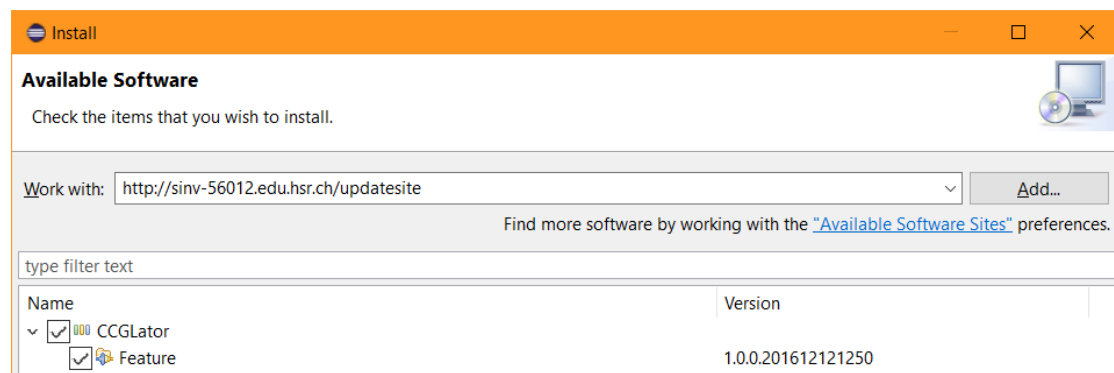


Figure 19: Installing the CCGLator Plugin via the update site

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:

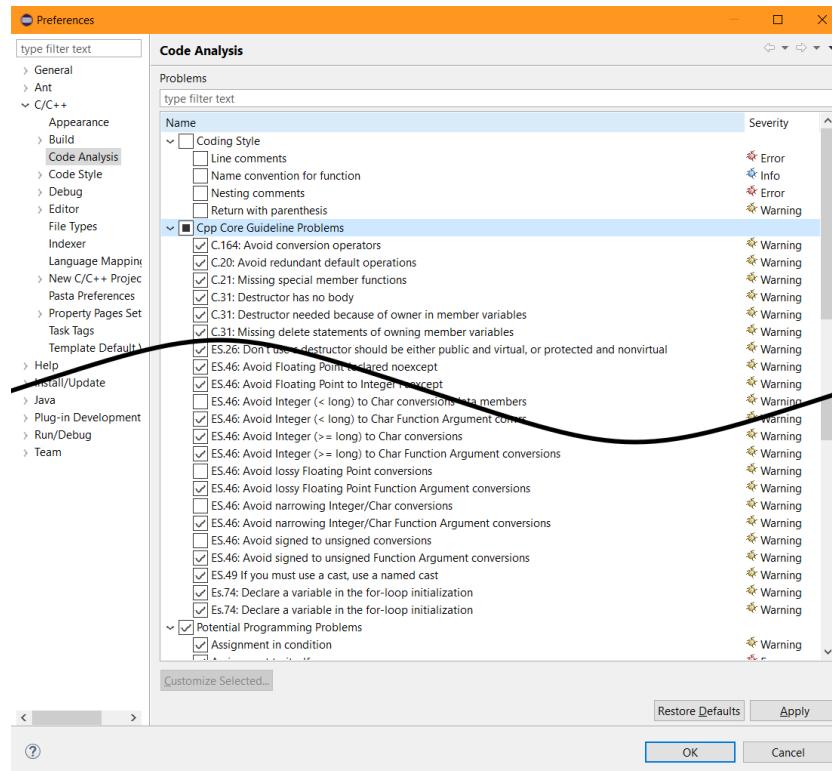


Figure 20: The Code Analysis Selection Screen

B.3 Usage

Any found issue of the enabled rules will get marked with a yellow squiggly underline in the code [1] (see Figure 21). Using the icons on the left [2] the Quick Fix list is opened [3]. From that list a Quick Fix can be chosen to apply to the code.

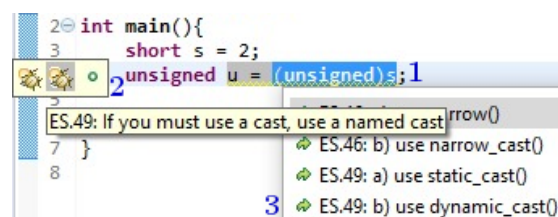


Figure 21: The Code Analysis Selection Screen

C Developer Manual

This developer manual covers how to set up and use the development environment for the code base of the CCGLator Plugin locally as well as on a continuous integration server. Additionally it mentions the project management tools used for this term project and lists the software used to create this document.

C.1 Local Development Environment

This chapter is to ease the participation in the development of this plug-in.

C.1.1 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
- 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.1.2 Eclipse SDK

To develop a Eclipse Plugin the Eclipse SDK is used. It can be found on the Eclipse Download Page [Foua]. After downloading, unpacking and running it, some additional Eclipse Plugins are needed. Below is a list of them with the respective update-sites:

C/C++ Development Tools	Default Eclipse Update-Site
C/C++ Development Tools SDK	
IFS CDT-Testing Feature	https://www.evelop.com/cdt-
IFS CDT-Testing Tools Feature	testing/development/
pASTa Feature	
(We do not recommended to install the Example-Code package.)	

C.1.3 CCGLator Workfolder

To set up the Eclipse workfolder for the CCGLator Plugin follow these steps:

1. Clone the git project
(`C:\CCGLadiator>git clone https://hmuster@git.hsr.ch/git/CCGLadiator workspace`)
2. Open Eclipse with the previously created Workspace
(e.g. "C:\CCGLadiator\workspace")
3. Open File → Import → General → Existing Project into Workspace
4. Select the Workspace as Root Directory
(e.g. "C:\CCGLadiator\workspace")
5. Click Finish

C.1.4 Coding

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

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

plugin.xml

In this file the newly added Checkers and Quick Fixes have to be activated so that Eclipse can find these rules and enable them.

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.

ASTHelper, ASTFactory

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

Testing

Every rule has a set of tests to verify the plug-in. If a new rule is added, tests for the Checker and Quick Fix have to be implemented accordingly.

To execute the JUnit tests run or debug one of the test suites (or a single test class) as a "JUnit Plug-in Test"

C.2 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 Plugin, 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 and schedule an automatic build.

Correctly configured the current plugin version can be installed by using the generated update-site package:

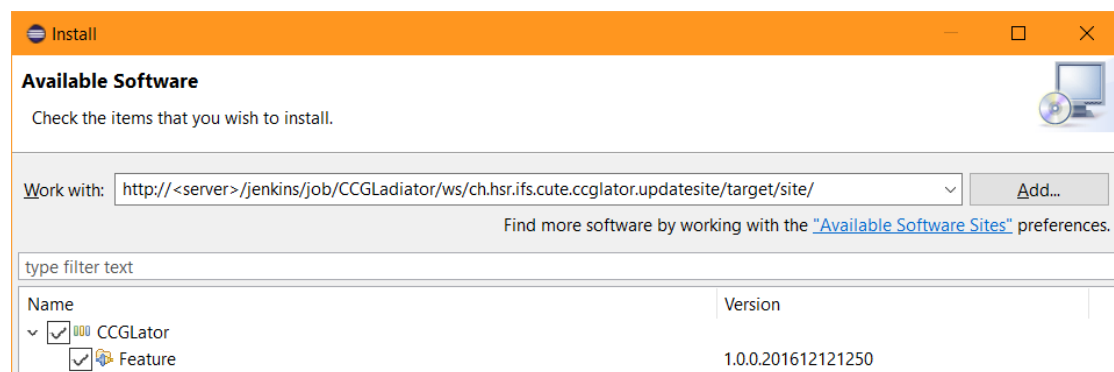


Figure 22: Installing the CCGLator Plugin via the update site generated by jenkins

C.3 Project Management Environment

The virtual server provided by the HSR also 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 shorter link to the plug-in update site.

C.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]

D C++ Codes used for the AST Images

Figure 2 Figure 3	<pre> 1 struct Subject { 2 void swap(Subject &a); 3 }; 4 void swap(Subject &l, Subject &r) {} </pre>
Figure 4	<pre> 1 struct Subject { 2 virtual void foo(int i) {} 3 }; </pre>
Figure 5	<pre> 1 struct To {}; 2 struct From { 3 operator To*() { 4 return new To(); 5 } 6 }; </pre>
Figure 6	<pre> 1 int main() { 2 int i = 2; 3 i = 2; 4 } </pre>
Figure 7	<pre> 1 int main() { 2 int i; 3 if(true) { 4 i = 1; 5 } 6 else { 7 i = 2; 8 } 9 } </pre>
Figure 8	<pre> 1 int main(){ 2 double d = -7.0; 3 unsigned u = 0; 4 short s = d; 5 u = d; 6 u = (int) d; 7 } </pre>
Figure 9	<pre> 1 void foo(int i) {} 2 int main(){ 3 double d = -7.0; 4 foo(d); 5 } </pre>
Figure 10	<pre> 1 int main() { 2 double d =5; 3 unsigned u = 0; 4 u = (int) d; 5 u = int(d); 6 } </pre>
Figure 11	<pre> 1 void function() { 2 int j; 3 for (j = 0; j < 0; j++) { 4 } 5 } </pre>
Figure 13	See Figures 8 and 9

References

- [cpp] cplusplus. Online reference for C and C++. <http://cplusplus.com>. [Online; accessed 15-December-2016].
- [DB16] Kilian Diener and Rolf Bislin. Pull Request for the new rule C.85. <https://github.com/isocpp/CppCoreGuidelines/pull/815>, 2016. [Online; accessed 15-December-2016].
- [ein16] einpoklum. What is `std::move()`, and when should it be used? <http://stackoverflow.com/a/27026280>, 2016. [Online; accessed 01-December-2016].
- [Foua] Eclipse Foundation. The Eclipse Project Downloads. <http://download.eclipse.org/eclipse/downloads/>. [Online; accessed 15-December-2016].
- [Foub] The Apache Software Foundation. Apache maven project. <https://maven.apache.org/>. [Online; accessed 16-December-2016].
- [Fouc] X.Org Foundation. libxtst 1.2.3-1. https://www.archlinux.org/packages/extra/x86_64/libxtst/. [Online; accessed 16-December-2016].
- [Foud] X.Org Foundation. xorg-server-xvfb 1.18.4-1. https://www.archlinux.org/packages/extra/x86_64/xorg-server-xvfb/. [Online; accessed 16-December-2016].
- [Fou16a] Eclipse Foundation. Eclipse CDT API Documentation. <http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.cdt.doc.isv%2Freference%2Fapi%2Fhelp-doc.html>, 2016. [Online; accessed 16-December-2016].
- [Fou16b] Standard C++ Foundation. Standard for Programming Language C++. <http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4618.pdf>, 2016. [Online; accessed 16-December-2016].
- [fSR16a] Institute for Software Rapperswil. Updatesite for CDT-Testing tools released by IFS. <https://www.cevelop.com/cdt-testing/neon/>, 2016. [Online; accessed 01-December-2016].
- [fSR16b] Institute for Software Rapperswil. Homepage of the Cevlop IDE. <https://cevelop.com>, 2016. [Online; accessed 15-December-2016].

- [fSR16c] Institute for Software Rapperswil. Homepage of the Institute for Software Rapperswil. <https://ifs.hsr.ch>, 2016. [Online; accessed 15-December-2016].
- [GM16] Elias Geisseler and Philipp Meier. GslAtorPtr - C++ Core Guidelines Pointer Checker and Support Library Refactorings. <https://eprints.hsr.ch/528/>, 2016. [Online; accessed 16-December-2016].
- [gs] git scm.com. Homepage of Git. <https://git-scm.com/>. [Online; accessed 15-December-2016].
- [Lav] Stephan T. Lavavej. MinGW Distro - nuwen.net. <https://nuwen.net/mingw.html>. [Online; accessed 15-December-2016].
- [Mic16] Microsoft. Github repository for the Guideline Support Library. <https://github.com/Microsoft/GSL>, 2016. [Online; accessed 15-December-2016].
- [Ora] Oracle. Java SE Downloads. <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. [Online; accessed 15-December-2016].
- [Pot13] Potatoswatter. Whats the difference between `std::move` and `std::forward`. <http://stackoverflow.com/a/9716708>, 2013. [Online; accessed 01-December-2016].
- [Sch] Christian Schenk. Download MiKTeX. <https://miktex.org/download>. [Online; accessed 15-December-2016].
- [SCH⁺16] Peter Sommerlad, Thomas Corbat, Marcel Huber, et al. Cvelop on Windows. <https://wiki.ifs.hsr.ch/CPlusPlus/ExW1#6>, 2016. [Online; accessed 15-December-2016].
- [SS15] Bjarne Stroustrup and Herb Sutter. C++ Core Guidelines. <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>, 2015. [Online; accessed 21-September-2016].
- [sta16] stackoverflow. When should `static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast` be used? <http://stackoverflow.com/a/332086>, 2016. [Online; accessed 01-December-2016].
- [Sut16] Herb Sutter. Unified Call Syntax. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4165.pdf>, 2016. [Online; accessed 06-October-2016].

- [Tea] The GIMP Team. GIMP - GNU Image Manipulation Program. <https://www.gimp.org/>. [Online; accessed 15-December-2016].
- [vdZ⁺] Benito van der Zander et al. TeXstudio. <https://miktex.org/download>. [Online; accessed 15-December-2016].
- [zKS16] Özhan Kaya and Kevin Schmidiger. CCGLator - C++ Core Guidelines Constructor Rules Checker and Quick-fixes. <https://eprints.hsr.ch/522/>, 2016. [Online; accessed 06-October-2016].