HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

INSTITUTE FOR
SOFTWARE

Semester Thesis, Institute for Software

Constificator

University of Applied Sciences Rapperswil

Fall Semester 2015
18. December 2015

Supervisor:	Prof. Peter Sommerlad
Authors:	Benny Gächter & Felix Morgner
Technical Advisor:	Silvano Brugnoni
Duration:	15.09.2015 - 18.12.2015
Extent of work:	240 Hours, 8 ECTS per student
Website:	`https://www.cevelop.com`

Abstract

Const is a substantial part of the C++ programming language but is often used inconsistent or not at all. A proper use of **const** is not just a matter of good programming style it makes the code more readable and prevents erroneous use. The goal of this thesis is to develop an Eclipse plug-in based on CDT and CODAN to help programmers place the keyword at the right position.

To determine whether a specifier can be **const**, static code analysis is required. To break down the complexity, the analysis is split into three different categories: local variables, function parameters, and class members. To enable an automatic and correct placement of **const**, in addition to the plug-in, a patch of CDT is necessary, because CDT does not support writing **const** on the right-hand side of the declaration specifier.

The result of this thesis is a plug-in that can be integrated and distributed with Cevelop and code for the **const** placement that can be committed into CDT. The plug-in marks specifiers that can be declared **const** and offers a quick-fix or for multiple line changes a change dialog. In addition to those markers there are informational markers for specifiers for which the constificator plug-in can not determine **const** qualification with absolute certainty.

Declaration of Authorship

We declare that this semester thesis and the work presented in it was done by ourselves and without any assistance, except what was agreed with the supervisor. All consulted sources are clearly mentioned and cited correctly. No copyright-protected materials are used in this work without permission of the respective copyright holders. The LaTeX source code for this document is based on "HSR-LaTeX-Template" by Florian Bentele.

Place and date	Benny Gächter
Place and date	Felix Morgner

Management Summary

The main goal of this semester thesis is to develop an Eclipse plug-in that is able to decide if a variable, a function parameter or a member function can be **const** qualified. It will then offer an appropriate refactoring of the code. In addition, the placement of the **const** keyword shall be adjustable in the Eclipse project settings. This will be delivered as a patch for CDT. According to the standard, the programmer is allowed to place **const** on the left or right-hand side of the *declaration-specifier* (ISO14, dcl.spec). Currently the **const** keyword is placed on the left-hand by Eclipse CDT. Our patch will also enable the user to align all **const**-qualifications to be on either the left or the right-hand side where applicable.

Motivation

Constificator

The **const** qualifier may appear in any type of specifier to specify constness of the object being declared. Such an object cannot be modified. An attempt to do so will result in a compile time error. Therefore, a consistent use of **const** makes code clearer because one can differentiate between modifying and inspecting functions and mutable and immutable objects. Furthermore, **const** qualified objects may be subject to optimization.

Const placement

By enabling the default placement of the **const** qualifier to be on the right-hand side of the type specifier, reading code can be simplified. As a rule of thumb, declarations are to be read from right to left. In every place except for the *declaration-specifier* the *cv-qualifiers* (ISO14, dcl.type.cv) have to be placed on the right-hand side. As an example of the confusion that might arise from placing the **const** on the left hand side consider this code:

Listing 1: Inconsistent placement of const

```
const int * left{};
int * const right{};
```

When trying to figure out what part of the declaration is **const**, inexperienced as well as experienced programmers might get confused. One might expect that the first line declares a **const**ant int-pointer. But if that were the case, what would the second line declare? If it was read as declaring an **int**-pointer **const**ant then the **const** keyword would be applied to the name `right` which is impossible since names cannot be **const**-qualified, only types can.

Goals

The main goal of this semester thesis is to analyze code and find variables, function parameters, and member functions that can be const qualified. It is imperative that the refactorings cover all corner cases and do not result in broken code. This includes changing the overload resolution set or modifying the behaviour of the program in any other way. The placement of the **const** qualifier can be adjusted in the project settings and is applied in a similar way as the format code action.

Results

The final product of this thesis are two plug-ins that can be integrated and deployed with Cevelop and a patch for CDT to align the placement of **const**. All insights gathered and steps needed to achieve the above goals are written down in this document.

Constificator

The plug-in is able to decide for all local variables, function parameters, and member functions if the can be **const** qualified. The plug-in cannot expand macros or refactor template variables. Since the decision for some definitions is rather complex the analysis takes a certain amount of time especially for larger project. For the future development one could enhance the performance by optimizing the checks or add functionality to inspect macros and templates.

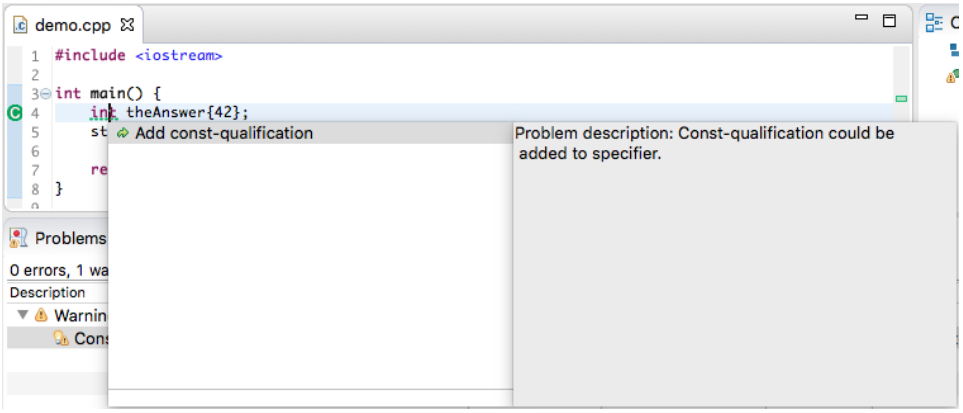


Figure 1: Applyig a quick-fix on a local variable

Const placement

The patch allows the user of Eclipse CDT to choose on which side of the *declaration-specifier* they want to place the **const** qualifier.

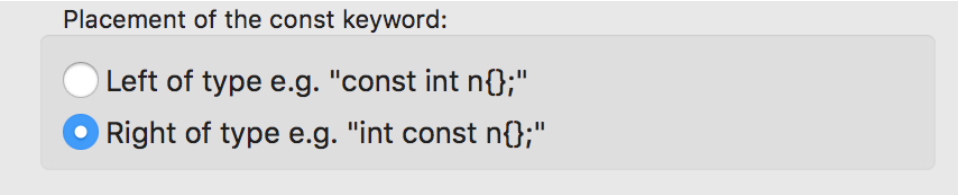


Figure 2: Choices for where to place the **const**

It also provides a functionality to align all **const**-qualifications across a file or even a whole project.

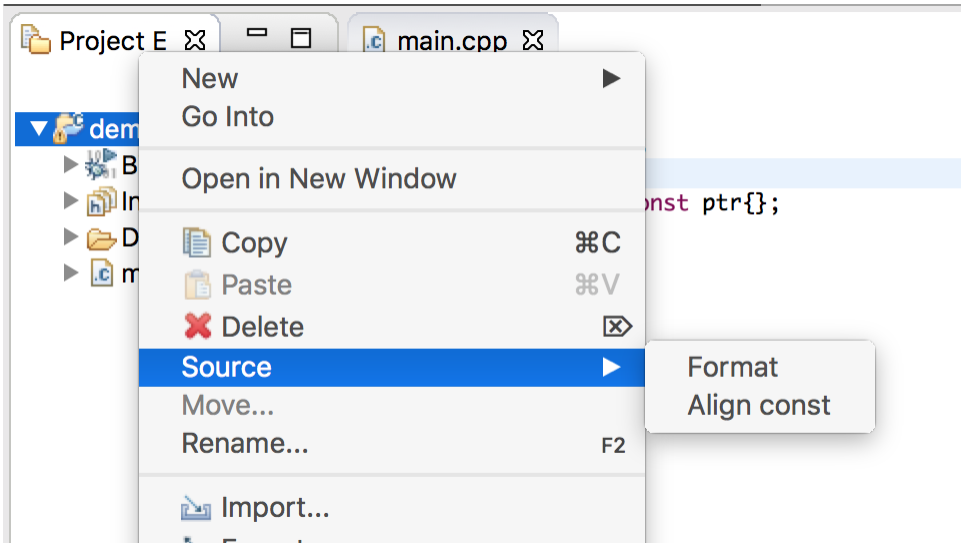


Figure 3: Apply **const** alignment to a project

Contents

1 Introduction **9**
- 1.1 Why does **Const**-Correctness matter? . . . 9
- 1.2 CDT Const Placement . . . 9
- 1.3 Constficator . . . 9
- 1.4 Approach to a solution . . . 10
- 1.5 CODAN . . . 10
  - 1.5.1 Checkers . . . 10
  - 1.5.2 AST visitors . . . 10
  - 1.5.3 Quick-Fixes . . . 11
  - 1.5.4 Markers and problems . . . 11
- 1.6 Abstract Syntax Tree . . . 11
  - 1.6.1 AST transformations . . . 13
  - 1.6.2 Bindings . . . 13
- 1.7 Const Placement . . . 13
  - 1.7.1 ASTWriter . . . 13

2 Analysis **14**
- 2.1 Local variables . . . 14
  - 2.1.1 Categorization . . . 14
  - 2.1.2 Variables of non-pointer-type . . . 14
  - 2.1.3 Variables of pointer-type . . . 18
  - 2.1.4 Variables of reference-type . . . 24
  - 2.1.5 Variables passed as function parameters . . . 24
- 2.2 Members of class types . . . 26
  - 2.2.1 Member variables . . . 26
  - 2.2.2 Member functions . . . 27

3 Implementation **29**
- 3.1 The ADMR Cycle . . . 29
  - 3.1.1 Analyze . . . 29
  - 3.1.2 Decide . . . 30
  - 3.1.3 Mark . . . 30
  - 3.1.4 Refactor . . . 30
- 3.2 packages . . . 30

3.2.1 ch.hsr.ifs.constificator . . . 31
3.2.2 ch.hsr.ifs.constificator.core . . . 32
3.3 Helpers . . . 33
3.3.1 Type system . . . 33
3.3.2 Pointer handling . . . 34
3.4 Problems and Decisions . . . 34
3.4.1 Informational Marker . . . 34
3.4.2 Performance issue for rewrite . . . 34
3.5 CDT const placement . . . 35
3.5.1 Additions to the ASTWriter . . . 35
3.5.2 User Interface . . . 35
3.6 Testing . . . 37
3.6.1 Testing with bitcoin . . . 37

4 Conclusion **38**
4.1 Achievements . . . 38
4.2 Limitations . . . 38
4.3 Outlook . . . 39

Appendices **40**

A User Manual **41**
A.1 Installation . . . 41
A.2 Usage . . . 42
A.2.1 Definitive Markers . . . 42
A.2.2 Informational Markers . . . 42
A.2.3 Refactoring . . . 43
A.2.4 Deactivation of markers . . . 43

B Personal Review **44**
B.1 Felix Morgner . . . 44
B.2 Benny Gächter . . . 45
B.3 Timelog . . . 46

C Build Infrastructure **47**
C.1 git . . . 47
C.2 Teamcity . . . 48
C.3 Eclipse Tycho . . . 48
C.4 Quality assurance . . . 48

D Protocols **49**
D.1 15.09.2015 . . . 49
D.2 25.09.2015 . . . 50
D.3 29.09.2015 . . . 50
D.4 06.10.2015 . . . 51

D.5 13.10.2015 54
D.6 20.10.2015 59
D.7 27.10.2015 60
D.8 02.11.2015 62
D.9 10.11.2015 63
D.10 17.11.2015 65
D.11 24.11.2015 66
D.12 01.12.2015 67
D.13 08.12.2015 68
D.14 15.12.2015 69

Introduction

The goal of this thesis is to write an Eclipse plug-in, called constificator, that analyzes code and makes it as **const** as possible. Thus, enhancing the quality of code. The constificator is based on the CDT CODAN infrastructure, that analyses the source code of a project and points out problems and offers improvements.

1.1 Why does Const-Correctness matter?

Simply said, **Const**-Correctness is just another form of type safety. It prevents you from inadvertently changing something one does not expect would be changed. **Const** can be used to explicitly distinguish between inspector and mutator methods and thus make your code easier to read. However you have to distinguish between the logical and physical state of your object. "The constness of a method should makes sense from outside the object" (iso) “The constness of a method must make sense to the object’s users, and those users can see only the object’s logical state.” (iso)

1.2 CDT Const Placement

By default, Eclipse CDT places **const** on the left hand side of the type specification. This behaviour is not wrong but it seems inconsistent since it is placed on the right hand side in every other case. Besides that, declarations are read from right to left. Writing **const** on the right-hand side makes reading declarations easier. During this project we will examine the code writing infrastructure of Eclipse CDT and evaluate if there is a simple way to change the **const** placement.

1.3 Constficator

The scope for the constificator is divided into mandatory and optional requirements. The must have requirements define the minimal feature set to release the plugin whereas the optional requirements are nice-to-have features.

Mandatory features:

- Refactoring of local variables
- Refactoring of function parameters
- Refacotring of member functions
- Refacotring of member variables

Optional features:

- Refactoring of lambdas
- Placement of const

If the constificator cannot determine with certainty whether a specifier can be **const** qualified or not an informational marker has to be placed with a remark that the programmer has to do further investigations in order to refactor the code.

1.4 Approach to a solution

To find out if a specifier can be declared **const** we will make use of the Abstract Syntax Tree (AST) which is provided by CDT (Eclb) and use checkers and visitors from CODAN (Ecla).

1.5 CODAN

CODAN, which stands for CODe ANalysis, is a framework in CDT that provides tools to perform static code analysis. It also offers facilities to mark problems in code and to offer changes to improve code quality.

1.5.1 Checkers

In CODAN terminology, a checker is the part of a plug-in that analyzes code and checks for problems. In that sense, a checker is the entry-point for static code analysis in Eclipse CDT. In most instances, the checker itself doesn't do the analysis itself. Rather it uses so called AST visitors to traverse the abstract syntax tree and analyze it for possible problems.

1.5.2 AST visitors

The visitors are used to traverse the AST. The constificator plug-in uses visitors to find nodes that represent declarations of local variables, function parameters, and functions. These declarations are then inspected and if possible marked.

1.5.3 Quick-Fixes

A quick-fix is a utility to transform code in order to fix a problem discovered by a CODAN checker. Quick-fixes have access to the complete AST tranformation infrastructure provided by Eclipse CDT. Thus they can perform arbitrarily complex changes to the source code being analyzed. In addition, quick-fixes can be used to start refactorings if user involvement is required to determine the extend of the change.

1.5.4 Markers and problems

Once a checker has determined that a problem exists in the code it can report it. This allows Eclipse to place a marker at the position of the node on which a problem has been reported. Markers can be associated with relating quick-fixes. This

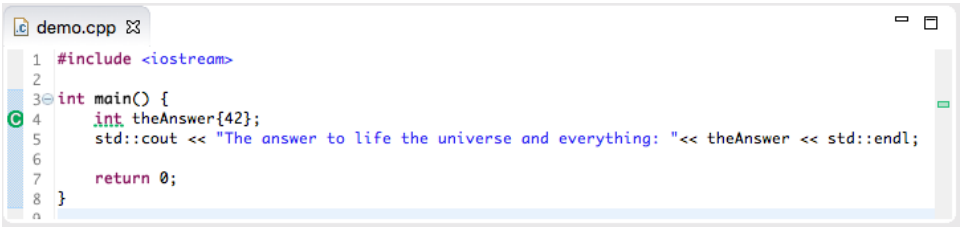


Figure 1.1: Marker example

allows the programmer to click on the marker and select a quick-fix to be applied.

Markers can also be associated to different categories like *Info*, *Warning* and *Error*. The set of categories available can be expanded via the Eclipse marker subsystem.

1.6 Abstract Syntax Tree

Every source file is represented as a tree-form known as AST. Each node or subtree in the AST represents an element of the source code. The nodes are all subclasses of IASTNode. Every subclass is specialized to represent an element of the C++ programming language (Las). Thus the AST can be cosidered as representing the exact **syntactic** structure of the source-code. However there is no **semantic** information about the program contained inside the AST.

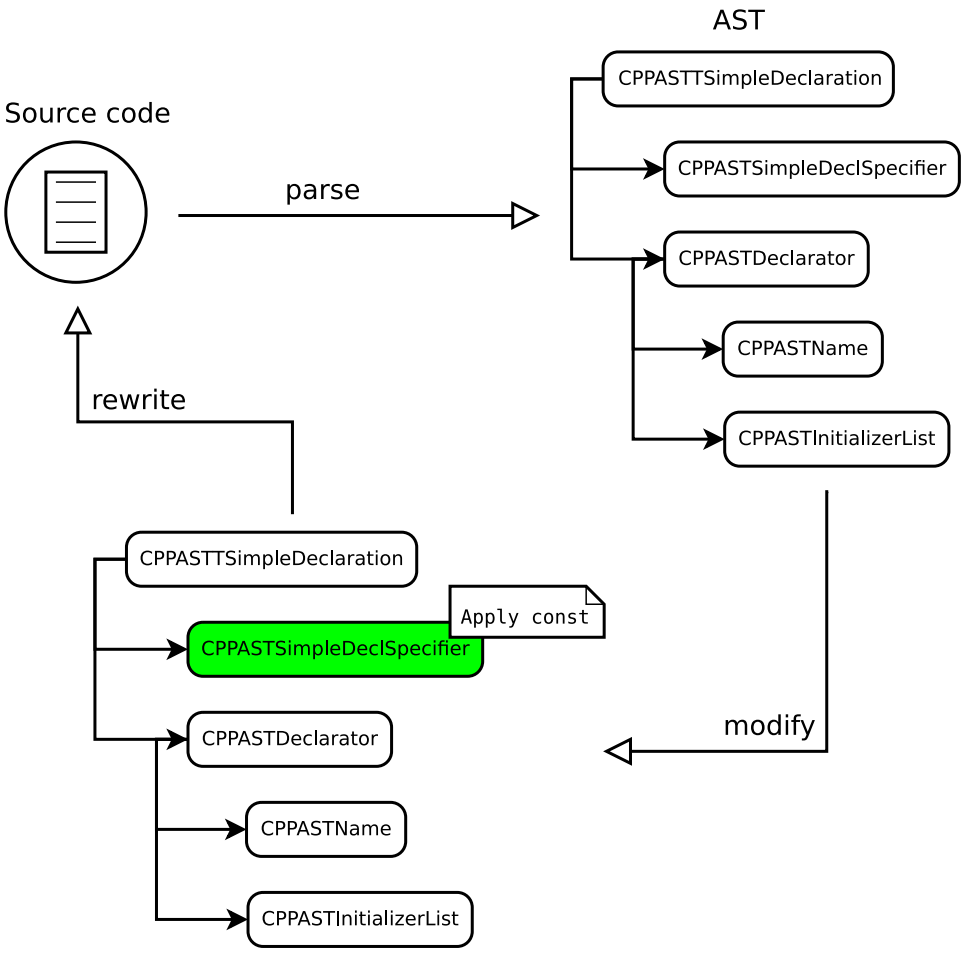


Figure 1.2: AST Workflow: Transforming a simple declaration

Figure 1.2 shows the transformation of the following code

Listing 1.1: Simple non-const declaration

```
int answer{42};
```

into this code:

Listing 1.2: Simple const declaration

```
int const answer{42};
```

Constificator makes extensive use of the Abstract Syntax Tree provided by CDT and the functions built around it.

1.6.1 AST transformations

When the AST of a file is retrieved from Eclipse CDT, all of its nodes are in the so-called *frozen* state. A node that is *frozen* cannot be manipulated and every attempt to do so raises an exception. This sutiation is owed to the fact that multiple different subsystems, like CODAN for example, access the AST from different threads. In order to change a node in the AST one must copy the node in question. After copying, the node can be manipulated in any applicable way.
To apply the transformation to the AST one has to use an instance of the class ASTRewrite. A rewrite allows for nodes to be added, removed and replaced. However, the rewrite itself does not change the AST directly but rather applies a textual change to the file resulting in a new AST when the file is parsed again by Eclipse.

1.6.2 Bindings

Bindings can be used when the AST cannot deliver enough information. They provide extended information and connections across multiple trees. This is especially useful when it comes to function definitions and declarations which should be found across multiple files.

1.7 Const Placement

Another aspect of this thesis is to harmonize the **const** placement. Currently Eclipse writes **const** on the left-hand side of the declspecifier. This is according to the standard (ISO14, const.placement) legal but problematic because in every other place **const** has to be on the right-hand side. By adding an option to write **const** on the right-hand side of decl specifiers the readability of code can be increased by making the **const** placement consistent.

1.7.1 ASTWriter

The ASTWriter is responsible for writing changes in the AST back into the file. This is the place where a change is needed in order to adjust the **const** placement. The **const** is represented as an attribute of the decl-specifier in the AST. By changing the order in which the ASTRewriter places the attributes arround the decl-specifier the placement of **const** can be adjusted.

Analysis

In order to be **const** qualified a specifier must not violate a set of criteria. Some of these criteria apply to all kinds of specifiers but for each type of specifier there is also a set of unique criteria. This criteria apply to code adhering to *ISO-14882:2014* and are specified in this chapter. The whole plug-in builds on top of these rules.

2.1 Local variables

Because C++ can be a very complex language with some language features that can be hard to understand a set of rules was specified. These rules make sure that every corner case is covered.

2.1.1 Categorization

In order to break down complexity the rules are categorized into three difficulty levels. Rules of easy and medium difficulty can be implemented so that they can decide with certainty if a variable can be **const**. For the difficult rules a more in-depth analysis would be required to determine **const** qualification without a doubt. Therefore an informational marker is placed if the plug-in suspects a possible **const** qualification.

- green is for easy rules
- yellow is for medium rules
- red is for difficult rules

2.1.2 Variables of non-pointer-type

Plain Old Data types (PODs) qualify as a starting point because they are the easiest possible case.

Listing 2.1: Intial code for const qualification of a non-pointer type local variable

```
void func()
  {
  int var{42};
  }
```

Because `var` is never modified in Example 2.1 it can be declared **const**. A marker is to be placed at the position of the specifier. After applying the quick-fix `int` should be changed to `int const` as seen in the next example.

Listing 2.2: Expected quick-fix output for non-pointer, non-reference type local variable

```
void func()
  {
  int const var{42};
  }
```

Conditions: To qualify as a target for transformation, the declaration of the variable **must not** violate any of the following conditions:

- C1 `var` is never used as the left-hand operand of a modifying binary expression
- C2 `var` is never used as the operand of a modifying unary expression
- C3 `var` is never passed to a function taking a reference to non-const-qualified type
- C4 `var` is never used to bind a reference to non-const-qualified type
- C5 the address of `var` is never passed to a function taking an *arbitrarily-const-qualified pointer* to *non-const-qualified type*
- C6 the address of `var` is never assigned to an *arbitrarily-const-qualified pointer* to *non-const qualified type*
- C7 the address of `var` is never passed to a function taking a *reference* to a *const-qualified pointer* to *non-const-qualified type*
- C8 the address of `var` is never used to bind a *reference* to a *const-qualified pointer* to *non-const-qualified type*

For conditions C3, C5 and C7 there exist exceptions. If and overload exists for the function, to which `var` is passed, which only differs in the *cv-qualification signature* (ISO14, conv.qual) of the respective parameter (see Example 2.3), it might be possible to add **const**-qualification to the *declaration-specifier* of `var`. However, it is not possible to guarantee that **const**-qualification can be added as this change might result in a change to the programs semantics.

Examples/Notes: The modifying binary operators are the the assignment and and compound assignment operators like =, +=, -=, etc. (ISO14, expr.ass). The postfix and prefix increment and decrement operators (ISO14, expr.post.incr, expr.pre.incr) are considered as modifying unary operators.

Listing 2.3: Function overload with more **const**-qualified parameter

```
void f(int & param)
  {
  // ...
  }

void f(int const & param)
  {
  // ...
  }
```

This example shows an overload for a functions that only differs in the *cv-qualification signature* of its parameter. In the trivial case it would be possible to decide if the behaviour of both overloads differs. But in the generic case, this problem can't be decided as this would mean that the *Halting Problem* would have to be solvable, which it is not (BM84).

Listing 2.4: Illegal reference binding

```
int main()
  {
  int var{1};
  int * & ref = &var;
  }
```

The above example is illegal according to the C++ standard, since binding of *references to non-const-qualified type* to temporary objects is forbidden (ISO14, dcl.init.ref). The code would be legal if the address gets stored in a variable with a declared type of *non-const-qualified pointer to non-const-qualified type* to which the reference would be bound as seen in Example 2.5.

Listing 2.5: Legal reference binding

```
int main()
  {
  int var{1};
  int * ptr{&var};
  int * & ref = ptr;
  }
```

However, since assigning the address of a variable to an *arbitrarily-const-qualified pointer to non-const-quaified type* would violate Condition C6 this is not being

considered here. Additionally this section only covers variables of *non-pointer type*. For a discussion of variables of *pointer-type* see 2.1.3.

Objects of class-type

Objects of *class-type* (ISO14, class) are somewhat similar to objects of *non-class-types*. One major difference is that *class-type* objects have member functions and data members which might be accessed. This extends the set of conditions that **must not** be violated.

Listing 2.6: Auxiliary code for class-type objects

```
struct cls
  {
  explicit cls(int number) : m_number{number} { }

  int number() const
    {
    return m_number;
    }

  void number(int const num)
    {
    m_number = num;
    }

  private:
    int m_number{};
  };
```

Listing 2.7: Initial code

```
void func()
  {
  cls var{42};
  }
```

Because `var` is never modified in Example 2.17 it can be declared **const**. A marker is to be placed at the position of the specifier. After applying the quick-fix `cls` should be changed to `cls const` as seen in Example 2.18.

Listing 2.8: Expected quick-fix output

```
void func()
  {
  cls const var{42};
  }
```

Conditions: To qualify as a target for transformation, the declaration of the variable **must not** violate any of the following conditions:

C9 `var` is never passed to a function taking a reference to *non-const-qualified type*

C10 `var` is never used to bind a *reference to non-const-qualified type*

C11 the address of `var` is never passed to a function taking an *arbitrarily-const-qualified pointer* to *non-const qualified type*

C12 the address of `var` is never assigned to an *arbitrarily const-qualified pointer* to *non-const qualified type*

C13 the address of `var` is never passed to a function taking a *reference* to a *const-qualified pointer* to *non-const-qualified type*

C14 the address of `var` is never used to bind a *reference* to a *const-qualified pointer* to *non-const-qualified type*

C15 all *non-static members* accessed on `var` are declared const

2.1.3 Variables of pointer-type

Variables of *pointer-type* are special in that **const**-qualification may be added at any pointer level. Therefore checks must be applied at every level.

Plain Old Datatypes (PODs)

Listing 2.9: Initial code

```
void func()
  {
  int * ptr = nullptr;
  }
```

Since neither the pointer itself nor the pointee are modified in Example 2.9 both can receive **const**-qualification. A marker should be placed on the pointer operator - the star - as well as on the *type-specifier*. The expected output after applying both quick-fixes can be seen in Example 2.10.

Listing 2.10: Expected quick-fix output

```
void func()
  {
  int const * const ptr = nullptr;
  }
```

Conditions: To qualify for a transformation, a pointer **must not** violate any of the following conditions:

C16 `ptr` is never used as the left-hand operand of a modifying binary expression

C17 `ptr` is never used as the operand of a modifying unary expression

C18 `ptr` is never passed to a function taking a *reference* to a *non-const-qualified pointer* to *arbitrarily-const-qualified type*

C19 `ptr` is never used to bind a *reference* to a *non-const-qualified pointer* to *arbitrarily-const-qualified type*

C20 the address of `ptr` is never passed to a function taking an *arbitrarily-const-qualified pointer* to a *non-const-qualified pointer* to *arbitrarily-const-qualified type*

C21 the address of `ptr` is never assigned to a variable with a declared type of *arbitrarily-const-qualified pointer* to *non-const-qualified pointer* to *arbitrarily-const-qualified* type

C22 the address of `ptr` is never passed to a function taking a *reference* to *const-qualified pointer* to *non-const qualified pointer* to *arbitrarily const-qualified type*

C23 the address of `ptr` is never used to bind a *reference* to *const-qualified pointer* to *non-const-qualified pointer* to *arbitrarily-const-qualified type*

To qualify for a transformation, a pointer-level **must not** violate any of the following conditions:

C24 the result of dereferencing `ptr` is never used as the left-hand operand of a modifying binary expression

C25 the result of dereferencing `ptr` is never used as the operand of a modifying unary expression

C26 the result of dereferencing `ptr` is never passed to a function taking a *reference* to *non-const-qualified type*

C27 the result of dereferencing `ptr` is never used to bind a *reference* to *non-const-qualified type*

C28 `ptr` is never passed to a function taking an *arbitrarily-const-qualified pointer* to *non-const-qualified type*

C29 `ptr` is never assigned to an *arbitrarily-const-qualified pointer* to *non-const-qualified type*

C30 `ptr` is never passed to a function taking a *reference* to an *arbitrarily-const-qualified pointer* to *non-const-qualified type*

C31 `ptr` is never used to bind a *reference* to an *arbitrarily-const-qualified pointer* to *non-const-qualified type*

If ptr is an arbitrarily deeply nested pointer with nesting depth of at least 2, the following special condition resulting from (ISO14, conv.qual) applies when trying apply const qualification.

C32 When ptr is initialized with another pointer or the address of another pointer and const-qualification is applied at any level L_i with $i > 0$ const-qualification must be applied to every level L_n with n being an element of $(0, i)$.

Note that the level numbers are determined by reading the type of a declaration right-to-left.

Examples/Notes For a description of modifying binary and unary operator see 2.1.2.
Condition C32 causes the following code to be valid.

Listing 2.11: Valid code due to C32

```
int main()
  {
  int * * pptr{};
  int const * const * const * ppptr{&pptr};
/* ^         ^       ^       ^
   |         |       |       |
   L3        L2      L1      L0 */
  }
```

While the following is considered invalid due to the missing const-qualification of L1.

Listing 2.12: Invalid due to C32

```
int main()
  {
  int * * pptr{};
  int const * const *       * ppptr{&pptr};
/* ^         ^       ^       ^
   |         |       |       |
   L3        L2      L1      L0 */
  }
```

This has some wide ranging implications. Consider this code:

Listing 2.13: Initial code for C32 example

```
int main()
  {
  int * ptr{};
  int * * pptr{&ptr};

  *pptr = nullptr;
  // ...
  }
```

Assuming, for the purpose of illustration, that it is unknown at the moment if the code represented by `// ...` changes the object pointed to by `ptr`, tt seems plausible, on first glimpse, that it could be transformed into:

Listing 2.14: Illegal assumption based on C32

```
int main()
  {
  int * ptr{};
  int const * * const pptr{&ptr};

  *pptr = nullptr;
  // ...
  }
```

However, due to the limitations arising from (ISO14, dcl.init.ref) this code would be illegal even though the int pointed to by the pointer pointed to by pptr is never actually changed. Thus the maximum **const**-qualification possible is:

Listing 2.15: Maximum const qualification for C32 example

```
int main()
  {
  int * ptr{};
  int * * const pptr{&ptr};

  *pptr = nullptr;
  // ...
  }
```

If the object pointed to by `ptr` is constifiable, the intuitive solution:

Listing 2.16: Intuitive solution

```
int main()
  {
  int const * ptr{};
  int const * * const pptr{&ptr};

  *pptr = nullptr;
  // ...
  }
```

would be correct.

Class types

Please note that we do not actually advocate the use of **new** but rather use it for the sake of compactness.

Listing 2.17: Initial code for class types

```
void func()
  {
  cls * ptr = nullptr;
  }
```

Listing 2.18: Expected quickfix output for class types

```
void func()
  {
  cls const * const var = nullptr;
  }
```

Conditions: With regard to the constificator conditions C33 - C46 and C48 are somewhat redundant. Only Condition C47 is unique to pointers to objects of class-type. They are only listed for the sake of completeness.
To qualify for transformation, a pointer **must not** violate any of the following conditions:

C33 `ptr` is never used as the left-hand operand of a modifying binary expression

C34 `ptr` is never used as the operand of a modifying unary expression

C35 `ptr` is never passed to a function taking a *reference* to a *non-const-qualified pointer* to *arbitrarily-const-qualified type*

C36 `ptr` is never used to bind a *reference* to a *non-const-qualified pointer* to *arbitrarily-const-qualified type*

C37 the address of `ptr` is never passed to a function taking an *arbitrarily-const-qualified pointer* to a *non-const-qualified pointer* to *arbitrarily-const-qualified type*

C38 the address of `ptr` is never assigned to a variable with a declared type of *arbitrarily-const-qualified pointer* to *non-const-qualified pointer* to *arbitrarily-const-qualified type*

C39 the address of `ptr` is never passed to a function taking a *reference* to *const-qualified pointer* to *non-const qualified pointer* to *arbitrarily const-qualified type*

C40 the address of `ptr` is never used to bind a *reference* to *const-qualified pointer* to *non-const-qualified pointer* to *arbitrarily-const-qualified type*

To qualify for transformation, a pointer **must not** violate any of the following conditions:

C41 the result of dereferencing `ptr` is never passed to a function taking a *reference* to *non-const-qualified type*

C42 the result of dereferencing `ptr` is never used to bind a *reference* to *non-const-qualified type*

C43 `ptr` is never passed to a function taking an *arbitrarily-const-qualified* pointer to *non-const-qualified type*

C44 `ptr` is never is never assigned to an *arbitrarily-const-qualified pointer* to *non-const-qualified type*

C45 `ptr` is never passed to a function taking a *reference* to a *const-qualified pointer* to *non-const-qualified type*

C46 `ptr` is never used to bind a *reference* to a *const-qualified pointer* to *non-const-qualified type*

C47 all member-functions called either by first dereferencing `ptr` or by using the member-of-object operator need to be const-qualified.

If ptr is an arbitrarily deeply nested pointer with nesting depth of at least 2, the following special condition resulting from (ISO14, conv.qual) applies when trying apply const qualification.

C48 When ptr is initialized with another pointer or the address of another pointer and const-qualification is applied at any level L_i with $i > 0$ const-qualification must be applied to every level L_n with n being an element of $(0, i)$.

2.1.4 Variables of reference-type

Variables of reference-type (which might be confusing, maybe it would be better to call them variables of pointer-type with reference-trait) are special in some ways:

- There cannot be any pointers to references (ISO14, operator pointer-to) (ISO14, no pointer to references))
- thus references are the point from which on out no further indirection is possible in declarations
- references are more like a special trait for pointers in that they alias what is referenced like an auto-dereferencing pointer
- References might or might not require storage (ISO14, 8.3.2.4)

Other than that, references behave much like the 'thing' they are referring to. This means that the ruleset depends solely on the type of the refered entity, with the exception that the reference itself shall not be **const**-qualified in any way.

2.1.5 Variables passed as function parameters

Variables that are passed as function parameters can be transformed if and only if they do not create overload resolution clashes. Therefore additional rules are required.

Pass by Value

All the above rules for non ref-type and non-pointer variables have to apply for the function scope. Since parameter declarations that differ only in the presence or absence of const are equivalent (ISO14, over.load) there is no other special requirement for parameters passed by value

Listing 2.19: Pass by value example

```
int f(int);
int f(const int); //redeclaration of f(int)
```

C49 Const qualified variables passed by value can be refactored without checking overloads

Pass by pointer

Basically the above rules for pointer variables apply, with the addition that overload resolution must be done in order to make sure that there are no clashes.
Since only the const type specifiers at the outermost level of the parameter type specification are "ignored" (ISO14, over.load) const type specifiers buried within

a parameter type specification are used to distinguish between overloaded function declarations.

Listing 2.20: Pass by pointer example

```
int f(int *)
int f(int const *) //overload
```

C50 There exists no overload O for the function F whose signature would clash with the new signature of F if **const**-qualification were to be applied to any of Fs parameters.

Listing 2.21: Example for pointer argument

```
void f(int const * arg) { }
void f(int * arg) { } // arg is never modified && the object ↩
    pointed to arg is never modified

int main()
  {
  int * ptr{};
  f(ptr);
  }
```

The best Constificator can do in this situation is to mark `arg` **const** in the second overload. Marking both the pointee and the pointer const would result in a duplicate definition for `f`

Listing 2.22: Duplicate definition of f

```
void f(int const * arg) { }
void f(int * const arg) { } // The best Constificator can do

int main()
  {
  int * ptr{};
  f(ptr);
  }
```

Providing an informational marker for this situation might be good idea since this situation might point to a badly named function or a bad design.

Pass by lvalue-reference

Mostly the same rules apply to function parameter of pointer-type with lvalue-reference-trait except that:

Listing 2.23: Pass by reference exception

```
void f(int &) { }
```

cannot be refactored to:

Listing 2.24: Illegal refactoring for pass by reference parameters

```
void f(int & const) { }
```

since references can never be const-qualified themselves.

2.2 Members of class types

2.2.1 Member variables

The general rules with regard to pointer/non-pointer/reference types apply here too. Only rules specific to class members are listed here.

Listing 2.25: Intial code for members of class types

```
struct s
  {
  s(int var) : m_var{var} { }

  int var() { return m_var; }

  private:
    int m_var{};
  };
```

Since no member functions exists that modify `m_var` it should be **const**-qualified resulting in the code seen in Listing 2.26.

Listing 2.26: Expected quick-fix output

```
struct s
  {
  s(int var) : m_var{var} { }

  int var() { return m_var; }

  private:
    int const m_var{};
  };
```

Conditions:

C51 `m_var` is initialized in the constructor initiliazer list.

C52 `m_var` is never modified in any way, shape or form as described in the Local variables section

C53 `m_var` is not declared as mutable.

Notes: Even though it might be possible to refactor the initialization of member variables from the body of a constructor to its initilizer list, we only consider member variables constifiable that are already initialized via a constructor intializer list. Other member variables will get flagged as 'possibly constifiable'.
More often then not passing returning non-const references to class members is wrong. Never the less we consider returning a non-const reference to a class member as a violation of the respective conditions (binding non-const reference to) found in 2.1.

2.2.2 Member functions

Listing 2.27: Initial code for member functions

```
struct s
  {
  s(int var) : m_var{var} { }

  int fun() { return m_var; }

  private:
    int m_var{};
  };
```

Since `fun` does neither call any non-const non-static member function nor modifies any non-static data member a marker should be placed on the *declarator* of the function. Applying the suggested quick-fix should result in the code seen in Listing 2.28

Listing 2.28: Expected quick-fix output

```
struct s
  {
  s(int var) : m_var{var} { }

  int fun() const { return m_var; }

  private:
    int m_var{};
  };
```

Conditions:

C54 `fun` does not modify any of the members of s

C55 `fun` does not call any non-const-qualified member functions

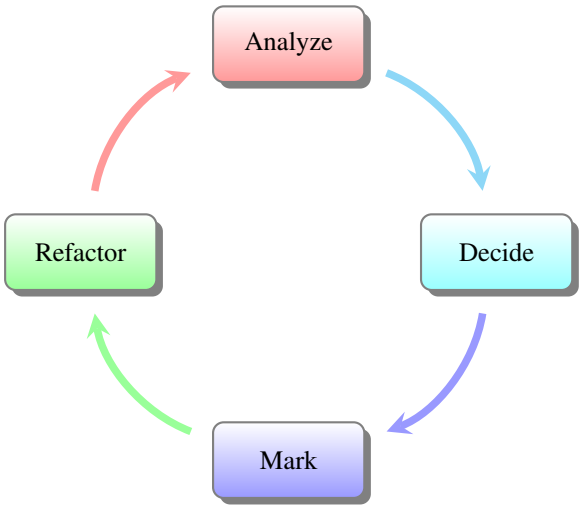
C56 There is no overload for `fun` that has the same reference-qualification and the same const-qualification that would arise when adding const-qualification to non-const var. A informational marker should be placed if a clash would occur, since this clash might indicate a programming problem with regard to the naming of the functions.

Implementation

The constificator consists of two plugins. The *ch.hsr.ifs.constificator* which contains the checkers, visitors quickfixes and refactorings. The logic, called deciders, are in the *ch.hsr.ifs.constificator.core* plugin. This seperation encourages to design a modular system that is open for furture improvements and addotions.

3.1 The ADMR Cycle

The plug-in follows a work flow that can be divided into the four steps analyze, decide, mark and refactor (ADMR). Each step represents one task that is handled by the constificator.



3.1.1 Analyze

As a first step the checker calls a visitor which traverses the Abstract Syntax Tree (AST) looking for declarations of variables, functions and function parameters. In order to analyze a source file the AST has to be parsed. CDT offers so called Visitors, which visit all or a subset of all nodes of the given AST.

Listing 3.1: Visit example

```
public int visit(IASTParameterDeclaration ↩
    parameterDeclaration) {
  ICPPASTParameterDeclaration cppParameterDeclaration = as(↩
      ICPPASTParameterDeclaration.class,parameterDeclaration);

  if (cppParameterDeclaration == null) {
    return PROCESS_SKIP;
  }

  return PROCESS_CONTINUE;
}
```

The above code example visits all function parameter declarations.

3.1.2 Decide

The nodes passed from the visitor are inspected according to the criteria defined in chapter 2. The criteria are divided into three categories: easy, medium and difficult. For criteria marked as easy and medium a decision can be made with certainty. The difficult criteria would require an in depth flow analysis of the code which cannot be done within this thesis due to timely limitations. Therefore the decision for **const** qualification has to reflect this three states and can be "yes", "maybe" or "no". "Yes" means a **const** qualification can be done without violating any restriction. "No" means a **const** qualification is not possible. If the decision is a "maybe" it requires further inspection by the programmer to make sure that the refactoring will not break the code.

3.1.3 Mark

Based on the decisions a marker is placed at the node. The marker can be informational, for nodes where a **const** qualification could not be determined with certainty ("maybe" decision) or definitive ones ("yes" decision).

3.1.4 Refactor

This action is triggered by the user and causes a quickfix for single changes or a change dialog for multiple changes. The node is handed over to the ASTRewriter which writes the changes to the code.(Vog)

3.2 packages

This section describes the package structure for the code providing the functionality required by constificator.

3.2.1 ch.hsr.ifs.constificator

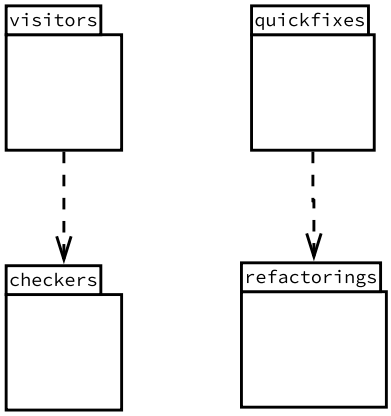


Figure 3.1: package diagram for ch.hsr.ifs.constificator

This subproject includes all front-facing code of constificator.

checkers During the work on constificator we identified four different caegories of constifiable nodes. We decided that each of these categories deserved its own marker type. We therefore settled with 4 separate checker classes. Furthermore, this allow the user to disable some checks if they want to do so.

quickfixes Similar to the checkers, the quick-fixes are devided into four categories. This is owed to the fact, that each category requires some slightly different processing to constify the selected node. Since our markers all follow the same concept for reporting problems, we were able to extract a fairly large chunck of common functionality for the quick-fixes.

refactorings In some situations, changing the **const**-qualification of a node might require multiple changes to the same or even different files. Simply applying the transformation might be confusing to the user. We therefore created an extremely simple refactoring. This refactoring makes use of the infrastructure provided by the Language Toolkit (Fre) included in Eclipse to show a simple change preview dialog. This dialog allows the user to get a good overview of the changes that will be applied. Having this dialog also allows the user to deselect certain changes.

visitors The AST visitors for the different categories of constifiable nodes all reside in this package. They are the entry-point to into the constificator core. Each visitor does some amount of preprocessing to ease the work for the core subsystem. This allows us do make some high-level descisions on whether or not a specific node might be constifiable in the first place. For example, we are able to filter out declarations that stem from a macro expansion, on this level.

3.2.2 ch.hsr.ifs.constificator.core

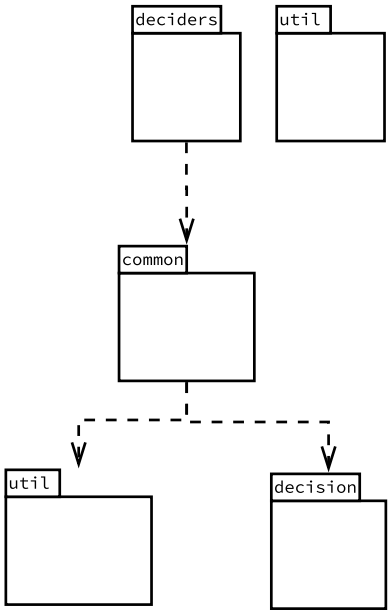


Figure 3.2: package diagram for ch.hsr.ifs.constificator.core

This subproject provides the core functionality for constificator.

util

This package contains general utilities such as functions to traverse the AST or do type processing. These utilities are used extensively throughout the code for constificator. The reach from simple functions to check if a certain value is contained in an array of objects to decaying and comparing types. An overview of the more important utilities is provided in 3.3

deciders

Constificator uses what we call deciders. The core responsibility of a decider is to check if one of the conditions found in chapter 2 is violated. Each decider is responsible to decide constifiability of one category of nodes.

decision A decision is made for every object that is inspected by the checkers. The decision can be YES (**const**-qualifiaction possible), NO (**const**-qualifiaction not possible) or MAYBE (**const**-qualifiaction is too complex to be determined by static code analysis and needs further inspection by the developer).

common Even though the details of constifiability for different nodes differ slightly, there are two major supersets of variables. Thus we extracted to common parts of the decision making process into this package. It contains the decision making core for pointer and non-pointer variable declarations. We make use of Java 8 lambda expressions to create an almost 1:1 mapping between our ruleset and the code.

Listing 3.2: Mapping a name to whether or not a condition is violated

```
(n) -> isLeftHandSideInModifyingBinaryExpression(n)
```

Listing 3.2 shows an example of a lambda expression used for non-pointer variables. Labmda expression of this form are added to a list that represents the ruleset that applies to this variable category. The conditions collected in such a list are evaluated until one of them the is violated e.g. returns `true`.

util The `util` package contains definitions for the conditions used by the different deciders. Each of the public functions in this package checks one of the rules defined in chapter 2. This eases debugging and allow for easy future expansion. This package also contains functions that are required to support the evaluation of certain conditions. As an example, the function `constOverloadExists(...)` in the class `MemberFunctionUtil` checks whether or not an overload for a member function exists, that is more const qualified than the function passed to it.

3.3 Helpers

Because deciding about constness can be complex neither CDT nor CODAN could provided us all the necessary functions, we had to extend the existing infrastructure.

3.3.1 Type system

Eclipse CDT offers a sophisticated subsystem to work with the C++ type system. Nonetheless we had to extend it in some ways.

Type comparison

The type processing offered by Eclipse CDT provides functionality to compare types regardless of whether or not they are *typedef-name*s of just plain type names. However, we needed to be able to compare types without considering their **const**-qualification. We use this functionality for overload resolution and, as an example, to find function declarations that only differ in the top-level **const**-qualification of their parameters.
We also created a helper function that determines the **const**ness of a pointer variable at a specific pointer level. This functionality is required so that we can decide

if the rules for *Qualification conversions* (ISO14, conv.qual) are violated. It also allows us to decide if and how the *cv-qualification signature* of a function parameter declarations differs from the declaration of a related overload of the function.

3.3.2 Pointer handling

The language definition of C++ (ISO14) allows for some rather interesting expressions with regard to pointers. As an example consider the code of Listing 3.3

Listing 3.3: Complex pointer dereference

```
int * * * ppptr{};

***ppptr = 4; // Expression 1

**&*(*&*&*(*&ppptr)) = 4; // Expression 2
```

The expressions 1 and 2 are equivalent. The latter is just a much more complicated way to express the former. As this arguably strange expression is allowed by definition, we needed a utility to determine the actual dereference level of pointer dereferencing expressions.

3.4 Problems and Decisions

This section contains some of the more noticeable problems and important decisions we made during the process of implementation.

3.4.1 Informational Marker

Because some decisions are fairly complex and would require an in-depth control flow analysis, or might even be impossible to decide on a computer, we decided to "best effort" decisions in those situation. For example, if an overload for a function exists that only differs in the *cv-qualification signature* of its parameters, we place an informational marker. This signals to the user, that we cannot decide the situation definitively and that they should take a look themselves.

3.4.2 Performance issue for rewrite

After including standard headers the quickfix took several seconds which influenced the user experience heavily. After some investigation we located the ASTRewrite function as bottleneck. Prof. Sommerlad and Silvanon Brugnoni helped us to sort out the problem by pointing out that we probably acquired tha AST with skipping already indexed headers.

Listing 3.4: Skip indexed headers for ast

```
// Before
IASTTranslationUnit ast = tu.getAST(index);

// After
int options = ITranslationUnit.AST_SKIP_INDEXED_HEADERS);
IASTTranslationUnit ast = tu.getAST(index, options);
```

Listing 3.4 shows the change needed to exclude already indexed header from AST generation.

3.5 CDT const placement

As a second part of this thesis that the default placement of **const** had to be revised in CDT. CDT places **const** on the left-hand side of the declaration specifier which is correct but among most C++ developers seen as inconsistent and wrong. Because, as a rule of thumb, type declarations are to be read from right to left and in every other place the **const** qualifier **must** not be on the left-hand side of the specifier.

3.5.1 Additions to the ASTWriter

Const is represented as an attribute of the corresponding specifier. By making the order in which the ASTRewriter writes the attributes to the file adjustable, the user of our patch can choose on which side of the declaration specifier the **const** keyword should be placed.

3.5.2 User Interface

Our patch adds a simple and straightforward user interface to CDT. The user can choose where to place the **const** keyword from within the Eclipse preferences. The settings can be found in the "Code Style" section.

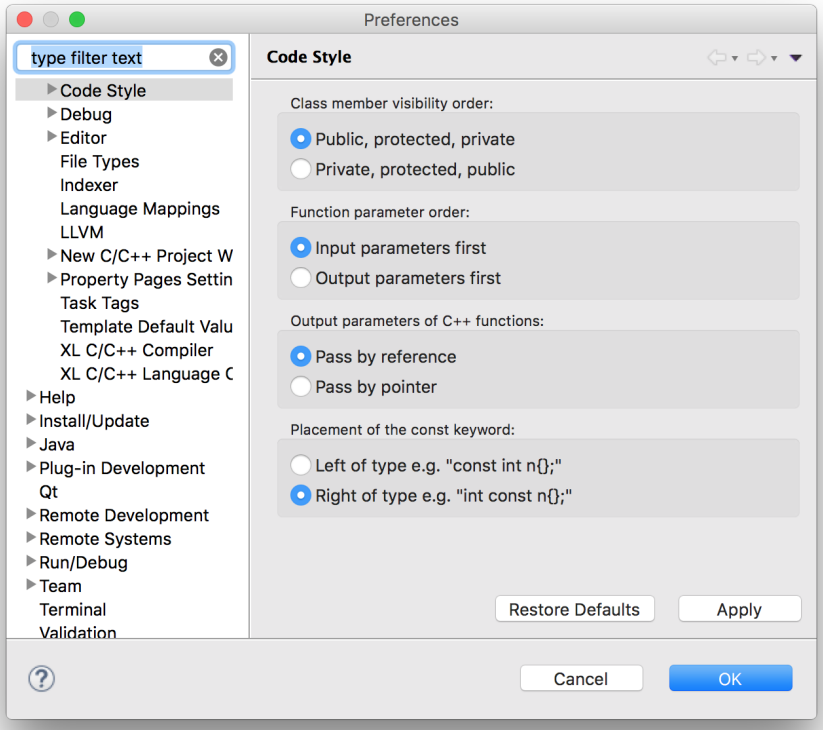


Figure 3.3: **const** placement preferences

In addition to the configuration UI, we provide a menu entry similar to "Format" called "Align const" that allows the user to align all **const** qualifications according to the selected preferences.

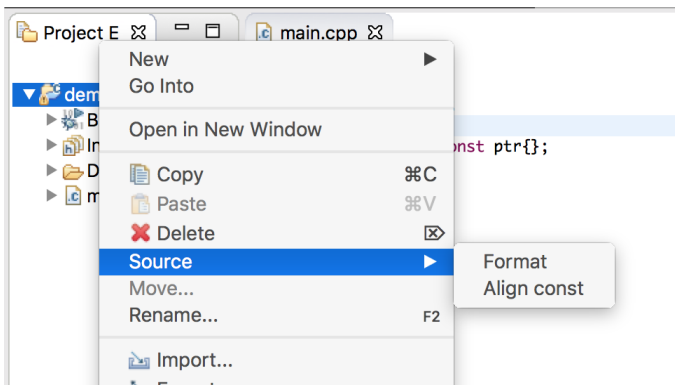


Figure 3.4: Apply **const** alignment to a project

3.6 Testing

To ensure the quality of our plugin and prevent any code breaking changes test cases are of paramount importance to the constificator. The test cases resemble the rules provided in the analysis and are implemented using the RTS testing infrastructure. Each rule has its own test case and can thus be tested individually.

3.6.1 Testing with bitcoin

In addition to the test cases the constificator is tested with a real world project. Bitcoin has been chosen because it is a large open source project written in C++ but has many developers and has grown organically and thus uses a variety of language features. It uses macros, inline functions, templates third party libraries and a number of C-Style features. Bitcoin has at the moment 110008 lines of code.[1] The constificator was able to analyze the source code and found in total 6443 problems. Because an "apply all" feature for adding **const** is not implemented we picked 30 random problems and applied the quick-fix. After each refactoring we compiled the code again and ran the tests to see if we broke code. Our changes all resulted in valid code that passed the tests.

[1] Commit 7a5040155ed59f8c9c51734bb2ee29f1593eaa6a, counted with find . -name '*.h' -o -name '*.cpp' | xargs wc -l

Conclusion

Constificator is a helpful plug-in that can be used for everyday programming even in larger projects. It motivates the programmer to use **const** whenever possible. Thus, making code cleaner and easier to understand.

4.1 Achievements

The following cases can be handled by constificator with certainty:

- Local variables
- Function parameters
- Member functions
- Member variables
- Lambda parameters

In the following cases an informational marker is set because constificator can not decide whether or not the change would result in a semantic change of the program.

- Variables passed to functions for which an overload exists that would take the variable in a more **const**-qualified fashion.

Our patch to Eclipse CDT allows the user to choose on which side of the *declaration specifier* the **const** keyword is to be placed. It also allows them to align the **const** keyword on a file or project basis.

4.2 Limitations

Because C++ is a complex language we were not able to cover all aspects of **const** qualification. The plug-in cannot expand macros, handle variadic function arguments or templates. We did not manage to create a mass-refactoring to apply all fixes associated with the definitive markers.

4.3 Outlook

To further improve the plug-in it would be useful to implement the decision for variadic function arguments or templates. Also, a code audit in terms of performance could be useful to the constificator because we were not able to cover this aspect during this thesis. A possibility to discard markers, especially the informational ones, would be a nice feature as well.
As for the **const** placement patch, one possibility to improve on it would be to offer integration as a "Save Action" for the file wide alignment of the **const** keyword.

Appendices

User Manual

A.1 Installation

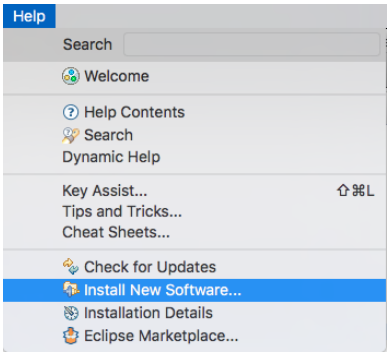


Figure A.1: Install new software in Eclipse

Click on Help and select install new Software

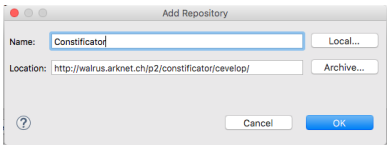


Figure A.2: Add update site

Add the `http://walrus.arknet.ch/p2/constificator/cevelop` update site

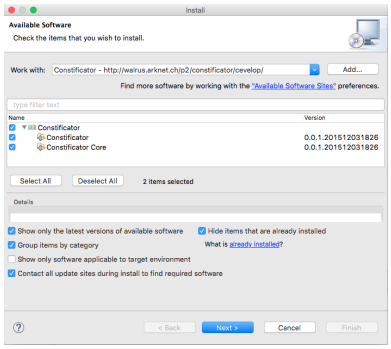


Figure A.3: Install constificator plug-in

Select the plugin and click install

A.2 Usage

Start a new C++ project or open an existing project from your workspace. Constificator will start to analyze your code and place markers for variables that can be const. There are two kinds of markers. Informational markers and definite markers.

A.2.1 Definitive Markers

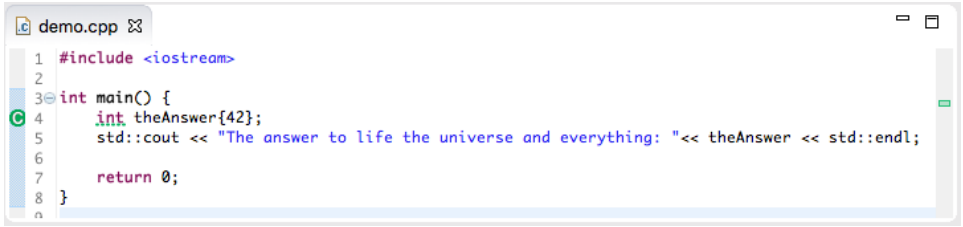


Figure A.4: Definitive marker for local variable

Definite markers have a green icon and are only placed if it is absolutely safe to use the keyword const at this position. This means a placement of const will not have any side effects.

A.2.2 Informational Markers

Informational markers are placed if there is a possibility to make your code const but the constificator cannot determine if there are any side effects. It is advised that you inspect your code carefully and decide whether a refactoring is appropriate or not.

A.2.3 Refactoring

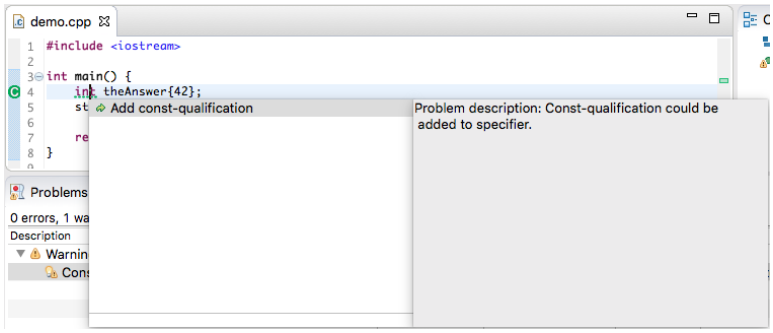


Figure A.5: Applyig a quick-fix on a local variable

To open the change wizard click on add const or use the shortkey control+1. The wizard shows a preview of the pending changes. Make sure the changes are correct before applying them. Changes may affect multiple files, especially if you refactor classes.

A.2.4 Deactivation of markers

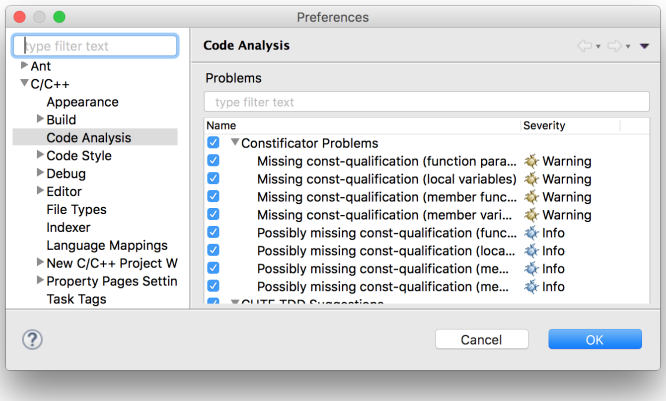


Figure A.6: Applyig a quick-fix on a local variable

The formal and informational markers can be deactivated separately. The settings can be found in the project properties under "Code Analysis".

Personal Review

This chapter is dedicated to a personal reflection on ourselves and the work we have done.

B.1 Felix Morgner

Working on our project has been an intense but rather pleasurable experience to me. Due to my personal experience with and love for the C++ programming language I was very excited from the getgo. For a long time I have wished to contribute something to the C++ community and I feel that this project allowed me to do so.

At the beginning of our project I was a somewhat sceptical whether the weekly meeting with Prof. Peter Sommerlad and Silvano Brugnoni would be necessary. In hindsight, I have to admit that these meetings were what kept us on track during these 14 weeks. I greatly appreciate the open, direct and fair feedback we received from Prof. Sommerlad. This feedback allowed us to evaluate if we were on track and to adjust our efforts in order to stay on track during the semester. I would also like to thank Silvano Brugnoni for helping us out with the more complicated quirks of Eclipse as a development environment and CDT as an "SDK". I am also thankful for the support Silvano provided us with during the last days of our project by pointing out major issues in our documentation.

Working with Benny Gächter was a pleasant experience. The fairly different levels of experience we both posses with regard to C++ tended to spawn very interesting discussions about different aspects of the language. I enjoyed the occasions on which we read through the standard and discussed low-level as well as high-level language concepts. I would like to thank Benny for the energy he invested into our project and for challenging me to improve on my skills.

On a more negative note, I would like to describe my experience in working with Eclipse CDT. After having used CDT as a "SDK", I am under the impression that it really needs some form of cleanup or modernization. Having worked on the code-base itself I feel that the evolution is clearly visible in terms of code style and quality. Aside from the code itself, I found the lack of documentation on CDT APIs rather disturbing.

I would like to summarize, that the project has been an overall pleasant experience.

B.2 Benny Gächter

What motivated me to write this thesis was, that I think C++ is a powerful, but often inconvenient solution because of the lack of tooling. This Thesis gave me the chance to change this at least a bit. That is why we both, Felix and I, made it a requirement to write a plug-in that we wanted to use ourselves.

The start of the project was very though for me because I am relatively new to C++ programming. I knew a lot of the concepts that C++ uses from my experience with Python or Java/. But C++ leaves a lot more freedom to the programmer which makes it sometimes very complex. Felix Morgner advised me to read through the standard to get a better understanding. He took great efforts to help me improve my knowledge about C++. Because of our different skill sets and levels we were able to complement each other in various aspects. I enjoyed working with Felix very much and I would like to thank him for his patience and all the effort he put into this project.

The weekly meetings with Prof. Sommerlad and Silvano Brugnoni were always very helpful because Prof. Sommerlad understood to give critical but fair feedback that kept us on track while Silvano Brugnoni was always there to answer our more implementation specific questions. I always felt that our opinions and concerns have been taken seriously.

During the implementation we faced our biggest challenge: mapping our rule set to the information we could retrieve from the AST. This had multiple reason. First, the whole CDT project seems to lack documentation. The second problem was the internal organization of CDT. A lot of APIs grew over the past few years without being "trimmed" which sometimes made it hard to understand whats really happening. I often had to figure out how thinks work by "try and error" which sometimes was a bit frustrating.

Overall I think this project was a success and I had a good time. I was able to learn a lot of new things not just about C++ and Eclipse plug-in development but I also grew closer to the C++ community.

B.3 Timelog

The following line chart shows the spent time per week.

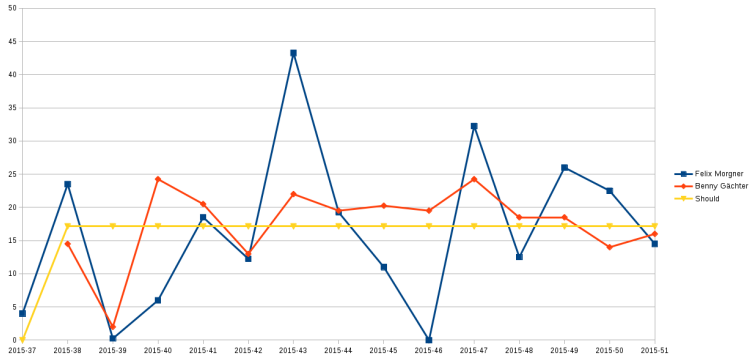


Figure B.1: Spent time

The drop in the second week is because we visited the CppCon in Bellevue, WA. The drop around calendar week 46 by Felix Morgner was because he moved to a new house that week and had to finish his CoPro testat. Besides that we tried to have a steady and balanced work load for each week.

Build Infrastructure

This chapter describes the infrastructure that was used to develop and build the plug-ins.

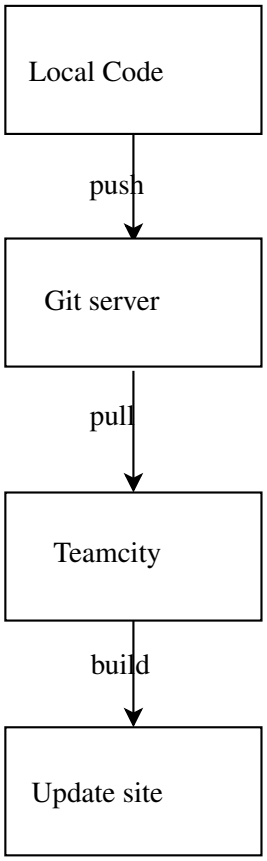


Figure C.1: Deployment Diagram

C.1 git

The git server, provided by the HSR, was used as version control system. It was chosen because of its great flexibility, general ease of use, and good integration into other systems such as continuous integration/continuous delivery.

C.2 Teamcity

Teamcity was used as continuous integration tool. It was configured to pull the latest commits from the git repository, compile the code and run the tests.

C.3 Eclipse Tycho

Tycho is a set of Maven plugins and extensions for building Eclipse plugins and OSGi bundles with Maven (Fou). We used tycho to run the JUnit Plug-in Test in our code and to build a deployable update-site for constificator.

C.4 Quality assurance

To ensure the quality of the constificator a number of measures have been taken. The plug-in was developed using test driven development. For all features and rules, there was from the beginning on a test case. This allowed refactoring without fear of regression. In addition to the test cases we used continous integration to automatically run all tests after every commit. As a second quality measure pair programming was used, because four eyes can see more than two and having someone watching you code can be fairly motivating to produce good code.

Protocols

D.1 15.09.2015

Agenda

- Project setup
- discuss scope
- Tools to use

Decisions

- Agenda must be sent to all participants 24h before meeting
- Project Infrastructure must be finished by 21.09.2015
- A scope must be defined until the next meeting
- A time line has to be worked out until the next meeting
- Agendas and Minutes can be done in a "casual" way in the wiki
- Next meeting will be held in Bellvue, WA, between 20.09.2015 and 25.09.2015 exact time and location will be fixed on-site
- All findings during analysis must be documented. Flip charts should be photographed and uploaded into the wiki

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.2 25.09.2015

Agenda

- Review Timeline
- Review workspace setup/Infrastructure
- Review Project scope
- Discuss goals for next week

Decisions

- Timeline has to be more detailed
- Scope should be written more verbosely (Split up goals)
- Example code for each refactoring case should be provided

Participants

- Prof. Peter Sommerlad
- Felix Morgner
- Benny Gächter

D.3 29.09.2015

Agenda

- Review edited scope
- Review mind map

Decisions

- Timeline must be finished until Friday
- Vereinbarung Urheber und Nutzungsrechte" should be signed

Participants

- Prof. Peter Sommerlad
- Felix Morgner
- Benny Gächter
- Silvano Brugnoni

D.4 06.10.2015

Agenda

- Review edited timeline
  - Grouped marking and refactorings more closely
  - Placed AST rewriter a little earlier
  - SW12 Feature Freeze is a little symbolic
  - Focus on finishing the thesis and poster at this point (SW12)
- Look at development progress
  - Finding of local variables made good progress
  - Finding non-const member variables made progress too
- Talk about issues and challenges of last week
  - Finding of local variables made good progress
  - Finding non-const member variables made progress too
- We had issues with dependency resolution regarding AST traversal (cost us 4 hours)
  - come to IFS quicker in case of such problems
- Preliminary checkout of the CDT code turns out it failed to compile and dependency resolution was tricky (2h)
  - ask in IFS for that, especially Thomas or Lukas
- What version of CDT should we 'fix' (8.7 or 8.8)
  - HEAD
- Outlook for the next week
  - Find local variable definitions
  - Handle things like int * ptr; correctly
  - Don't make refs const
  - Offer quick fix for local variable definitions
  - Find member variable definitions
  - Offer quick fix for member variable definitions
  - We know where to fix the const placement
  - We know how to make const placement configurable

- Put update site in place (http://walrus.arknet.ch/p2/)

- General questions

  - Should we try to factor out complex control flows to make local variables const?

```
void fun1()
  {
  char someVar;

  if(condition)
    {
    int const temp = otherFun(x, y)
    someVar = anotherFun(temp);
    }
  else
    {
    if(otherCondition)
      {
      someVar = anotherFun(42);
      }
    else
      {
      someVar = 'z';
      }
    }
  // more code not writing to someVar
  }
```

would be transformed to something like this:

```
char EXTRACTED_INIT_SOMEVAR()
  {
  if(condition)
    {
    int const temp = otherFun()
    return anotherFun(temp);
    }
  else
    {
    if(otherCondition)
      {
      return anotherFun(42);
      }
    else
      {
      return 'z';
      }
    }
  }
```

```
void fun1()
  {
  char const someVar = EXTRACTED_INIT_SOMEVAR;
  // more code not writing to someVar
  }
```

- When refactoring member variables, we might need to factor out an 'init' member function, should we leave that for the advanced goals?

```
 struct s1
  {
  s1(size_t a, size_t b, std::string c)
    {
    // do some complicated calculations to determine↩
         m_member
    m_member = resultOfComplicatedCalculation;
    }

  private:
    char m_member;
  };
```

if `m_member` can be const it would be transformed to something like this:

```
struct s1
  {
  s1(size_t a, size_t b, std::string c) : m_member{↩
      EXTRACTED_INIT_MEMBER(a, b, c)}
    {
    }

  private:
    char EXTRACTED_INIT_MEMBER(size_t a, size_t b, ↩
        std::string c)
      {
      // do some complicated calculation here
      return resultOfComplicatedCalculation;
      }

  private:
    char const m_member;
  };
```

Even though we are not sure if this is always safe to do. This is open to discussion. It should impose no problem as long as the initialization is not depend on any object state.

- review current test cases

Decisions

- Agenda should be more verbose (e.g. example code)
- Silvano/Thomas should be contacted earlier in case of problems
- Items on the agenda should be clear statements that can be checked
- Link to update site will be sent on Wednesday
- Complex control flows can be left out

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.5 13.10.2015

Agenda

- Review goals
  - Find local variable definitions
    - Handle things like int * ptr; correctly
    - Achieved
    - Don't make refs const
    - Achieved
- Offer quick fix for local variable definitions
  - Not yet working
- Find member variable definitions
  - Achieved
- Offer quick fix for member variable definitions
  - Not yet working
- We know where to fix the const placement
  - Achieved

  - At least I assume that org.eclipse.cdt.ui.actions.FormatAllAction, org.eclipse.cdt.internal.formatter.Scribe, org.eclipse.cdt.internal.core.model.ASTStringUtil and org.eclipse.cdt.internal.formatter.CodeFormatterVisitor are good starting points.
  - If possible, I will talk to Thomas about that this week in order to achieve the goal of implementing the placement.

- We know how to make const placement configurable
  - (Semi-)Achieved
  - I believe that org.eclipse.cdt.internal.ui.preferences.CodeFormatterPreferencePage is the right place to start

- Put update site in place
  - Achieved
  - Reachable at http://walrus.arknet.ch/p2/

Refactor non-const parameters (by value)

```
#include <iostream>

int answer(int question)
  {
  return question * 2;
  }

int main()
  {
  auto const question = 21;
  auto const fourtytwo = answer(question);
  std::cout << fourtytwo << '\n';
  }
```

Would be refactored to something like this:

```
#include <iostream>

int answer(int const question)
  {
  return question * 2;
  }

int main()
  {
  auto const question = 21;
  auto const fourtytwo = answer(question);
```

```
  std::cout << fourtytwo << '\n';
  }
```

Refactor non-const parameters (by lvref)

```
#include <string>
#include <iostream>

void answer(std::string &question, bool &answer)
  {
  if(question.size() % 42)
    {
    answer = false;
    }
  else
    {
    answer = true;
    }
  }

int main()
  {
  using namespace std::literals;

  auto reply = false;
  auto question = "Should reply be const?"s;

  answer(question, reply);

  std::cout << "The answer is " << (reply ? "yes" : "no") << ↩
      '\n';
  }
```

Should be tranformed to:

```
#include <string>
#include <iostream>

void answer(std::string const &question, bool &answer)
  {
  if(question.size() % 42)
    {
    answer = false;
    }
  else
    {
    answer = true;
    }
  }

int main()
  {
```

```
  using namespace std::literals;

  auto reply = false;
  auto const question = "Should reply be const?"s;

  answer(question, reply);

  std::cout << "The answer is " << (reply ? "yes" : "no") << ↩
      '\n';
  }
```

Note that through making the first parameter of the function 'answer(...)' const, we can make the string 'question' const too. We cannot, on the other hand, change the second parameter since we need to be able to write to it.

Refactor non-const member-variables

```
#include <string>
#include <iostream>

struct philosopher
  {
  explicit philosopher(std::string const &name, std::size_t ↩
      const age = 41) : m_name{name}, m_age{age} { }

  std::string name()
    {
    return m_name;
    }

  std::size_t age()
    {
    return m_age;
    }

  void grow_older()
    {
    ++m_age;
    }

  private:
    std::string m_name;
    std::size_t m_age;
  };

int main()
  {
  auto v = philosopher{"vroomfondel"};

  std::cout << v.name() << " is " << v.age() << " years old\n↩
      ";

  v.grow_older();
```

```
  std::cout << v.name() << " is " << v.age() << " years old\n↩
      ";
  }
```

Would be refactored to something like this:

```
#include <string>
#include <iostream>

struct philosopher
  {
  explicit philosopher(std::string const &name, std::size_t ↩
      const age = 41) : m_name{name}, m_age{age} { }

  std::string name()
    {
    return m_name;
    }

  std::size_t age()
    {
    return m_age;
    }

  void grow_older()
    {
    ++m_age;
    }

  private:
    std::string const m_name;
    std::size_t m_age;
  };

int main()
  {
  auto v = philosopher{"vroomfondel"};

  std::cout << v.name() << " is " << v.age() << " years old\n↩
      ";

  v.grow_older();

  std::cout << v.name() << " is " << v.age() << " years old\n↩
      ";
  }
```

Decisions

- Implement as stated above

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.6 20.10.2015

What has been done

- What has been done last week
- Implementing the Quick Fix turned out to be more complicated than expected. Even though AbstractAstRewriteQuickFix is more or less easily worked with, we had difficulties with getting the associated node. Since we wanted to mark the place where the const would go, e.g. the type of a declaration, we were unable to get the associated node using the provided API. For now, we changed the way we place the markers so that it works at least for 'simple' local variables (e.g. no ref or ptr qualification).
- Thomas gave us good hints on where to start working on the const placement, and we have a very crude patch (basically hardcoded const to go on the right). We also have implemented the related preferences, but they are currently not used for determining the position.
- We reworked our internal infrastructure (better modularization) to be more flexible when developing in parallel.
- The Quick Fix currently does not support refactoring OTAs (One Time Assignments)
- We lost some of our power since Felix was moving to a new living place last week.
- Detection of multiple writing access is still buggy, but its getting better It works for 'simple' variables but fails on pointers. for example in:

```
char * const boo{};
*boo = 1;
*boo = 2;
boo still gets flagged, even though the pointee can't be↩
    constified.
```

Review goals

- Refactor non-const parameters (by value) - Not achieved
- Refactor non-const parameters (by lvref) - Not achieved
- Refactor non-const member-variables - Not achieved
- Implement the ability to configure the const placement - Semi-achieved

Goals for next week

- Implement Quick Fix for local variables
  - 'simple' variables and 'pointer' types
  - Refactor OTAs
- Correctly identify multiple accesses
- Identify variables that are assigned inside of control structure blocks (if/else and the like)
- Detect which parts of 'multi-stage' pointers (e.g char * * * foo) can be constified
- Everything that was on the list for last week.

What is getting deferred

- Implementing const placement is getting deferred until next week, since we need two people to work on our 'core-bussiness'.

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.7 27.10.2015

Owing to the fact that we had to redo our analysis during the course of the week, we changed our goals/priorities according to the list below.

Review of last week

- Define a proper ruleset for Local variables
  - We realized during last weeks meeting that are analysis was incomplete and not well structured and formalized. We fixed that issue during the course of the week. Our goal was to formalize a ruleset which catches all cases and is formally correct and verified against ISO 14882:2014. - Done
- Implement Checker/Quick Fix for local variables
  - Finished up to C25
  - We are confident to have a complete implementation by Tuesday evening.
- Write RTS tests for markers for all rules - Done
- write RTS tests for quickfixes for all rules - Done
- Refactor OTAs (One Time Assignments)
  - We are unsure on whether or not we should do this in the first place. It seems (more or less) trivial for integral and floating point types as well as for C-style string literals but for instances of class-type we are unsure on whether or not a full static analysis of the code would be necessary. It might be possible for example that the default constructor of a class-type might influence global state (e.g by reading from cin) and thus 'optimizing away' that constructor call for const-improvement would be a bad idea. - not done yet

Goals for next week

- Finish implementation for all rules
- Specify rules for member functions
- Add const placement option to AST Rewriter
  - The 'option pane' already exists, it just doesn't have any effect other than saving it to the preferences at this time.

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.8 02.11.2015

Review of last week

- Finish implementation for all rules
  - Done until C30 - We were slower as expected because we had to change the way how checker and quickfix exchange information
  - C30 and above turn out to be a little tricky due to how the different means of initializing a variable differ in the AST
- Specify rules for function parameters
  - Specified under Local variables
- Refactor OTAs (One Time Assignments) -No progress
- Add const placement option to AST Rewriter - No progress The 'option pane' already exists, it just doesn't have any effect other than saving it to the preferences at this time.
- Write RTS tests for markers for all rules - Done
- write RTS tests for quickfixes for all rules - Done
- Refactor OTAs (One Time Assignments)
  - We are unsure on whether or not we should do this in the first place. It seems (more or less) trivial for integral and floating point types as well as for C-style string literals but for instances of class-type we are unsure on whether or not a full static analysis of the code would be necessary. It might be possible for example that the default constructor of a class-type might influence global state (e.g by reading from cin) and thus 'optimizing away' that constructor call for const-improvement would be a bad idea. - not done yet

Goals for next week

- Rework timeline
- Segregate the ruleset into basic, medium and advanced difficulty
- Finish Implementation up to and including the medium difficulty levels (local variables)
- Implement informational markers for the currently implemented advanced difficulty rules

- Finish Implementation up to and including the medium difficulty levels (function parameters)
- Implement informational markers for the advanced rule in the function parameters ruleset
- Implement custom marker icon
- Write ruleset draft for class members

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.9 10.11.2015

Review of last week

- Rework timeline - done
- Segregate the ruleset into basic, medium and advanced difficulty - done
- Finish Implementation up to and including the medium difficulty levels (local variables)
  - Finding easy local variables
  - Finding medium local variables
  - Refactor easy local variables
  - Refactor medium local variables
- Implement informational markers for the currently implemented advanced difficulty rules
- The informational markers have been added to the plugin xml
- Finish Implementation up to and including the medium difficulty levels (function parameters)
  - Finding easy function parameters - done
  - Finding medium function parameters - done
  - Refactor easy function parameters - done

  - Refactor medium function parameters - done
  - NOTE: function parameter refactoring is currently not capable of overload resolution so it might produce clashes
  - NOTE: currently only function definitions get refactored. We haven't found a "change signature refactoring", should we roll our own or are we missing something?

- Implement informational markers for the advanced rule in the function parameters ruleset - in progress
- Implement custom marker icon
- A custom marker category has been defined and the marker code is refactored to use the new IDs. We still need an icon
- Write ruleset draft for class members - not done

Goals for next week

- Add custom icon
- Finish implementation of the informational markers
- Implement overload resolution
- Formalize ruleset for class members
- Implement "change signature" refactoring?
- Implement finding of non-const class members
- Implement const-placement analysis for CDT

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.10 17.11.2015

Review of last week

- Add custom icon - Done
- Finish implementation of the informational markers
  - (local variables) not done
  - (member variables) not done
  - (function parameters) not done
  - (member functions) not done
- Implement overload resolution - in progress
- Formalize ruleset for class members - Done, Members of class types
- Implement change preview for function parameter refactoring - Done
- Implement finding of non-const class members - not done
- Implement const-placement for CDT - not done

Goals for next week

- Finish overload resolution
- Implement finding of non-const class members
- Informational markers for local variables
- Informational markers for member variables
- Informational markers for function parameters
- Informational markers for member functions
- Implement working setting for const-placement

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.11 24.11.2015

Review of last week

- Finish overload resolution
  - Local function declarations and definitions - done
  - Definitions/declarations in separate files - not done
  - How can we access the actual AST when we get a PDOMCPPFunction from the index?

  Which AST do you want? I assume either the place of definition and/or any declarations?

```
void resolveASTName(IBinding binding, IIndex index, ↩
    ICProject project) {
  IIndexName[] declarations = index.findNames(binding, ↩
      IIndex.FIND_DECLARATIONS); // may return null
  for (IIndexName declaration : declarations) {
    ITranslationUnit tu = CoreModelUtil.↩
        findTranslationUnitForLocation(declaration.↩
        getFile().getLocation(), project);
    IASTTranslationUnit ast = tu.getAST(index, ↩
        ITranslationUnit.AST_SKIP_INDEXED_HEADERS); // do↩
         not reparse headers that are already present in ↩
        the index.
    IASTNode node = ast.getNodeSelector(null).findName(↩
        declaration.getNodeOffset(), declaration.↩
        getNodeLength()); // your IASTNode, may return ↩
        null.
  }
}
```

  Be advised, tu.getAST(...) is somewhat expensive, as it usually reparses the file. To speed things up, I recommend that you cache the ASTs in a Map<ITranslationUnit, IASTTranslationUnit>.

- Implement finding of non-const class members - member functions not done
- Informational markers for local variables - done
- Informational markers for member variables - done
- Informational markers for function parameters - done
- Informational markers for member functions - not done
- CDT const placement
  - Move settings to 'Code-Style' - done

  - Place const respecting the preferences - done
  - Update-site - done
  - Mass refactoring - not done

Goals for next week

- Transform wiki pages to thesis (markdown to latex and some additional formatting/content)
- generic lambdas?
- Member functions
- CDT const placement mass refactoring
- Test constificator with real projects

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.12 01.12.2015

Review of last week

- Transform wiki pages to thesis (markdown to latex and some additional formatting/content) - Done
- CDT const placement mass refactoring - done
- Finish overload resolution - done
- member functions - in progress
- Mass refactoring for const placement - done

Goals for next week

- Cleanup/refactor code
- remove dead code
- reduce duplicate code
- unify API naming
- Write thesis
- Write about Analysis
- Write about Implementation
- Write about Refactoring real life code
- member functions

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.13 08.12.2015

Review of last week

- Cleanup/refactor code
  - reduce duplicate code
  - unify API naming
- remove dead code
- Write thesis
  - Write about Analysis
  - Write about Implementation
  - Write about Refactoring real life code
- member functions

Goals for next week

- Cleanup/refactor code
- remove dead code
- reduce duplicate code
- unify API naming
- Write thesis
- Write about Analysis
- Write about Implementation
- Write about Refactoring real life code
- member functions

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

D.14 15.12.2015

Review of last week

- member functions - done
- Write thesis - in progress
  - Write conclusion
  - Make screenshots and graphics
  - Write about Refactoring real life code
  - Make Poster

Participants

- Prof. Peter Sommerlad
- Silvano Brugnoni
- Felix Morgner
- Benny Gächter

List of Figures

1 Applyig a quick-fix on a local variable 4
2 Choices for where to place the **const** 5
3 Apply **const** alignment to a project 5

1.1 Marker example 11
1.2 AST Workflow: Transforming a simple declaration 12

3.1 package diagram for ch.hsr.ifs.constificator 31
3.2 package diagram for ch.hsr.ifs.constificator.core 32
3.3 **const** placement preferences 36
3.4 Apply **const** alignment to a project 36

A.1 Install new software in Eclipse 41
A.2 Add update site 41
A.3 Install constificator plug-in 42
A.4 Definitive marker for local variable 42
A.5 Applyig a quick-fix on a local variable 43
A.6 Applyig a quick-fix on a local variable 43

B.1 Spent time 46

C.1 Deployment Diagram 47

Listings

1 Inconsistent placement of const 3
1.1 Simple non-const declaration 12
1.2 Simple const declaration 12
2.1 Intial code for const qualification of a non-pointer type local variable 15
2.2 Expected quick-fix output for non-pointer, non-reference type local variable 15
2.3 Function overload with more **const**-qualified parameter 16
2.4 Illegal reference binding 16
2.5 Legal reference binding 16
2.6 Auxiliary code for class-type objects 17
2.7 Initial code 17
2.8 Expected quick-fix output 17
2.9 Initial code 18
2.10 Expected quick-fix output 18
2.11 Valid code due to C32 20
2.12 Invalid due to C32 20
2.13 Initial code for C32 example 21
2.14 Illegal assumption based on C32 21
2.15 Maximum const qualification for C32 example 21
2.16 Intuitive solution 22
2.17 Initial code for class types 22
2.18 Expected quickfix output for class types 22
2.19 Pass by value example 24
2.20 Pass by pointer example 25
2.21 Example for pointer argument 25
2.22 Duplicate definition of f 25
2.23 Pass by reference exception 26
2.24 Illegal refactoring for pass by reference parameters 26
2.25 Intial code for members of class types 26
2.26 Expected quick-fix output 26
2.27 Initial code for member functions 27
2.28 Expected quick-fix output 27
3.1 Visit example 30
3.2 Mapping a name to whether or not a condition is violated 33

3.3 Complex pointer dereference 34
3.4 Skip indexed headers for ast 35

Bibliography

[BM84] BOYER, Robert S. ; MOORE, J. S.: A Mechanical Proof of the Unsolvability of the Halting Problem. In: *Journal of the ACM (JACM)* 31 (1984), jul, S. 441–458

[Ecla] ECLIPSE, Foundation: *CDT/designs/StaticAnalysis*. Website, . – `http://wiki.eclipse.org/CDT/designs/StaticAnalysis`; visited 09/14/2015.

[Eclb] ECLIPSE, Foundation: *Eclipse CDT*. Website, . – `https://eclipse.org/cdt/`; visited 09/14/2015.

[Fou] FOUNDATION, Eclipse: *Tycho home*. Website, . – `https://eclipse.org/tycho/`; visited 12/18/2015.

[Fre] FRENZEL, Leif: *The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs*. Website, . – `https://eclipse.org/articles/Article-LTK/ltk.html`; visited 12/18/2015.

[iso] ISOCPP.ORG: *Const Correctness*. Website, . – `https://isocpp.org/wiki/faq/const-correctness`; visited 09/14/2015.

[ISO14] ISO: *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. Geneva, Switzerland : International Organization for Standardization, 2014. – 1376 (est.) S. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=64029`

[Las] LASKAVAIA, Elena: *Codan*. Website, . – `http://www.infoq.com/presentations/codan`; visited 09/14/2015.

[Vog] VOGELLA: *Eclipse Tycho Article*. Website, . – `http://www.vogella.com/tutorials/EclipseTycho/article.html`; visited 09/14/2015.