Constificator User Manual

Institute for Software IFS
https://www.cevelop.com
https://ifs.hsr.ch

This manual provides an introduction to **Constificator**, a plug-in for the Cevelop IDE that aims to help programmers write const-correct code. The plug-in has first been released in version 1.5 of Cevelop and has since undergone a lot of changes, both in the user interface as well as the analysis subsystem. This document focuses on the version of the plug-in that is shipped with **Cevelop 1.6**.

Contents

Note: *Keybindings*

Any keybindings described and used throughout this document are the **default** keybindings for a fresh installation of the Cevelop IDE. They might be subject to change between releases and can be configured in the workspace-wide preferences.

1 Introduction

The main purpose of **Constificator** is to help you write const-correct code starting from the first line of your project. Of course, **Constificator** also supports you retroactively when reviewing or extending an existing codebase. Since it makes use of the existing problem reporting infrastructure of Cevelop, it integrates nicely into the existing workflow by providing standard problem markers and *QuickFixes* as you type.

In the following sections we take a look at how to activate and configure **Constificator**. We also show some examples to demonstrate some of the situations in which the plug-in will help you write better code as you type.

1.1 Benefits of using Constificator

You might be wondering why you should care about const-correctness. In this section, we give a short overview of the benefits you get from consequently using **Constificator** and try to convince you to make your code as `const` as possible.

First and foremost, writing const-correct code helps you to make your intent clear. Since in C++ we often care about efficency, we also often use *pass-by-reference* to eliminate unnecessary copies. Unfortunately, variables in C++ are – as the name implies – variable by default. To see why this is an issue, consider the following code:

```cpp
#include <string>

void print(std::string & data) {
  // ... implementation goes here
}

int main() {
  std::string data{"I don't want this to be modified"};
  print(data);
}
```

Listing 1.1: It is not clear whether or not `print` modifies its argument.

Admittedly, the code in Listing 1.1 is somewhat contrived, but imagine the definition of `print` is located in a different translation unit. From the signature alone, we can not be sure whether the function modifies the string we provide it with. Of course the behavior could be documented in some form of API documentation, but why not let the compiler enforce the immutability on our behalf? Listing 1.2 shows an improved version of the function definition. With this signature, we enable the compiler to make sure that the function does not modify its argument.

Note: *const_cast<...>*

Of course, the implementation could always const_cast<> away the constness of its argument, but if it were to modify the values afterwards, the function itself would exhibit *undefined-behavior*.

```
#include <string>

void print(std::string const & data) {
  // ... implementation goes here
}

int main() {
  std::string const data{"I don't want this to be modified"};
  print(data);
}
```

Listing 1.2: Adding const makes the intent clear.

It is worth noting, that changing print's signature also enabled us to add const to the declaration of data, thus preventing it to be changed accidentally. This is only one of the areas, where **Constificator** can help you by interactively providing suggestions on how to improve your code.

Having function signatures clearly express if the arguments will be modified or not, allows you and other programmers that use your code to reason about what a given code segment can and can not do. In multithreaded code for instance, we can have any number of concurrent consumers of the same piece of data, as long as none of them modifies it. By consequently applying const, we can decide if a function is safe to use on a high level, while having the compiler enforce the immutability for us.

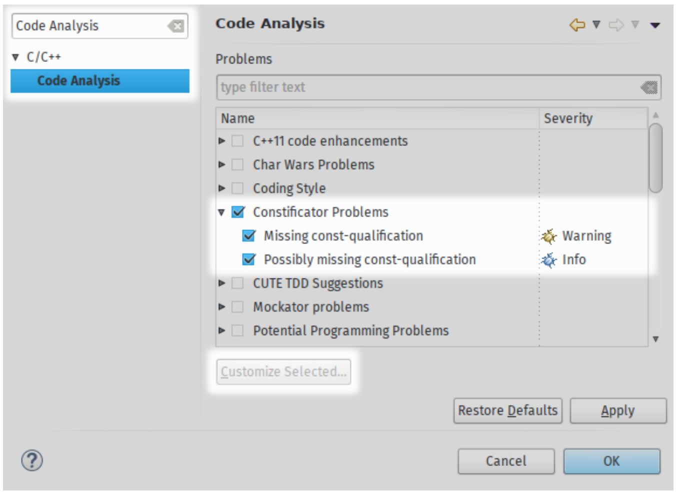
Besides additional *type-safety*, applying const where possible can, under some circumstances, have a profound impact on how much code the compiler generates. For an example of such an optimization arising from a single const, see Jason Turner's talk Rich Code for Tiny Computers which he gave at CppCon 2016.

If you want to learn more about *const-correctness*, we recomend you take a look at isocpp's FAQ entry on this very topic.

1.2 Enabling Constificator

Since its inclusion in version 1.5 of the Cevelop IDE, **Constificator** has been disabled by default. One reason for this is that, especially in large projects, the plug-in often places a large number of markers. Another reason is that running a complete *C++ Code Analysis* on an existing, large project will take a cosinderable amount of time.

Since **Constificator** integrates natively with the Cevelop IDE, enabling it is extremely easy. First you will want to make sure you are using at least version 1.6 of the Cevelop IDE. Open the preferences via Window » Preferences and navigate to the *Code Analysis* settings in the group *C/C++*.



Code Analysis

▾ C/C++
Code Analysis

Code Analysis

Problems

type filter text

Name	Severity
▸ ☐ C++11 code enhancements	
▸ ☐ Char Wars Problems	
▸ ☐ Coding Style	
▾ ☑ Constificator Problems	
☑ Missing const-qualification	Warning
☑ Possibly missing const-qualification	Info
▸ ☐ CUTE TDD Suggestions	
▸ ☐ Mockator problems	
▸ ☐ Potential Programming Problems	

Customize Selected...

Restore Defaults Apply

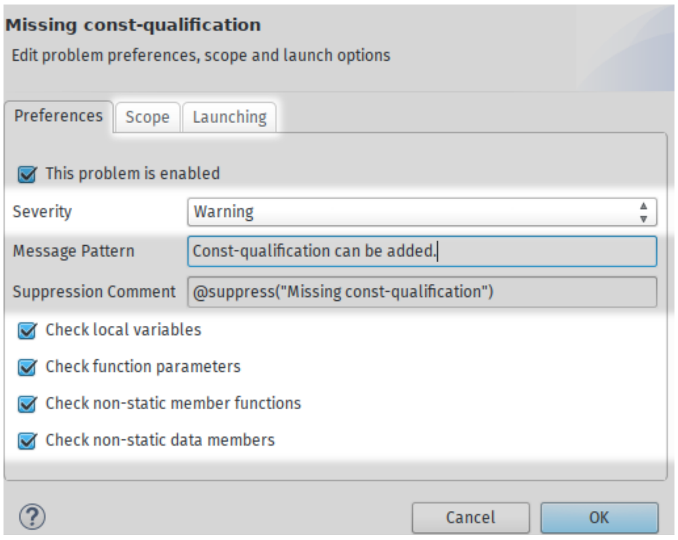
Cancel OK

Figure 1: The Code Analysis preferences

Figure 1 shows the preferences dialog for the *Code Analysis* preferences. To enable **all** checks supported by **Constificator**, check the row *Constificator problems*. You also have the option to only enable the *high-confidence* checks, by selecting only *Missing const-qualification*, or the *low-confidence* checks by only activating the checkbox for *Possibly missing const-qualification*. After you have chosen which checks to activate, you can save these setting by clicking the OK button. We will see the difference of the two checks in section 1.3 on the next page.

Note: *Per project preferences*

Like most preferences, you can also set the *Code Analysis* settings on a per project basis. To enable **Constificator** only for a specific project, open the *Project Properties* by either right-clicking the project and selecting Properties, or using Alt + ↵ on **Windows** and **Linux** or similiarly ⌥ + ↩ on **macOS**. Afterwards the process is identical to changing the workspace-wide settings.



Missing const-qualification
Edit problem preferences, scope and launch options

Preferences | Scope | Launching

☑ This problem is enabled

Severity	Warning
Message Pattern	Const-qualification can be added.
Suppression Comment	@suppress("Missing const-qualification")

☑ Check local variables
☑ Check function parameters
☑ Check non-static member functions
☑ Check non-static data members

Cancel OK

Figure 2: Customizing the Constificator checks

In addition to these coarse-grained settings, you can also customize each of the checks by selecting one of the two check categories and clicking the Customize Selected... button. In the newly opened window, you can define which specific checks you want **Constificator** to perform. The default is for all checks to be activated. You can also change the severity of the markers the plug-in will place. As with other *Code Analysis* plug-ins, you can also define on **which resources** the checks should be performed (via the *Scopes* tab) as well as **when** the plug-in's analysis should be executed.

1.3 Constificator Marker Types

In Figure 1 on the preceding page we saw that the plug-in features two top-level marker types, the *Missing const-qualification* marker and the *Possibly missing const-qualification* marker. The difference between the two is the confidence with which **Constificator** has determined that the `const` keyword could be inserted.

Since C++ is a fairly complex language, it is often difficult to decide whether or not a declaration can receive const-qualification. The plug-in works hard to try and make sure that no markers are placed where inserting the `const` keyword would produce ill-formed code. The plug-in also tries hard to avoid high-level semantic changes to the program being analyzed.

1.3.1 Missing const-qualification

Most of the time, adding const-qualification does not change the semantics of the program being analyzed. For example, if a variable is never used in a context that modifies its value, const-qualification can be added without changing any of the high-level behavior. In these cases, **Constificator** will put a *Missing const-qualification* marker at the position where the `const` keyword can be inserted.

1.3.2 Possibly missing const-qualification

Consider the following code, imagining the definition of `Base` being in a separate header file:

```
struct Base { virtual void do_stuff() {} };

struct Derived : Base {
  void do_stuff() {
    return;
  }
};
```

Listing 1.3: Adding `const` would change overriding to hiding.

Technically speaking, there is nothing that would prevent us from adding `const` to the definition of `Derived::do_stuff()`. The implementation does not touch any *non-static* data members of `Derived` or `Base` and also does not call any *non-const non-static* member functions of `Derived` or `Base`. However, applying `const` to the definition of `Derived::do_stuff()` would change the semantics of the code, since all of a sudden, `Derived::do_stuff()` would **hide** `Base::do_stuff()` and not **override** it. Since **Constificator** can not decide whether or not to apply `const` in this situation, it places a *Possibly missing const-qualification* marker on `Derived::do_stuff()`.

2 Usage

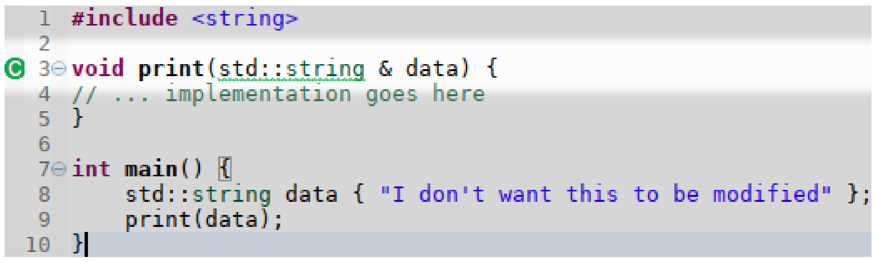
In this chapter, we take a look at how to work with **Constificator**. To keep the code examples small, we use somewhat contrived examples. We would also like to point out that there is an example project available at `https://github.com/IFS-HSR/constificator-demo`.

Note: *Constificator must be activated*

This section assumes that **Constificator** is activated, either for the whole workspace or the project you are working on. If you haven't done so already, please activate the plug-in. For information on how to do that, and how to configure **Constificator** to suite your needs, please see section 1.2 on page 2.

2.1 Applying Simple Fixes

Let's start with a simple example of how to apply code fixes (so-called *QuickFixes*) to your code using **Constificator**. Starting from the code seen in Listing 1.1 on page 1, you will notice that the plug-in places a problem marker on line 3 (see Figure 3 below).



```
 1 #include <string>
 2
 3 void print(std::string & data) {
 4 // ... implementation goes here
 5 }
 6
 7 int main() {
 8     std::string data { "I don't want this to be modified" };
 9     print(data);
10 }
```

Figure 3: Marked function parameter

Note the green and white **C** left of the line numbers as well as the green squiggly line under the type of `print`'s parameter. What **Constificator** is telling us here is that it detected that the `std::string` stored at the location referenced by data is not changed in the scope of `print`. You could of course go in and add the `const` yourself, but where is the fun in that? Instead, try placing your cursor on the line 3 and pressing Ctrl+1 on **Windows** and **Linux** or similiarly ⌘+1 on **macOS**. This will open a drop-down right next to the marked section. In this drop-down, click on *Add const-qualification*. The plug-in will insert the missing `const` keyword for you.

After a couple of seconds, you will notice that **Constificator** has placed a new marker on line 8, since it determined that the variable `data` is never used in a modifying context. Adding the `const` keyword here, is just as easy as above. If you ever apply a *QuickFix* when you did not mean to, just use Ctrl+z on **Windows** and **Linux** or similiarly ⌘+z on **macOS** to undo the change.

2.2 Applying Fixes Across Multiple Files

Besides handling fixes inside a single file, **Constificator** also supports adding constness across file boundaries. This functionality comes in handy when you want to apply a *QuickFix* to a function parameter and the function has one or more forward declarations in different files. The canonical example for this situation are `class`es or `struct`s that are often declared in a header file, and defined in a separate implementation file.

```
struct Structure {
  void doit();
private:
  int member{};
};
```

Listing 2.1: Structure.h - A simple C++ `struct`

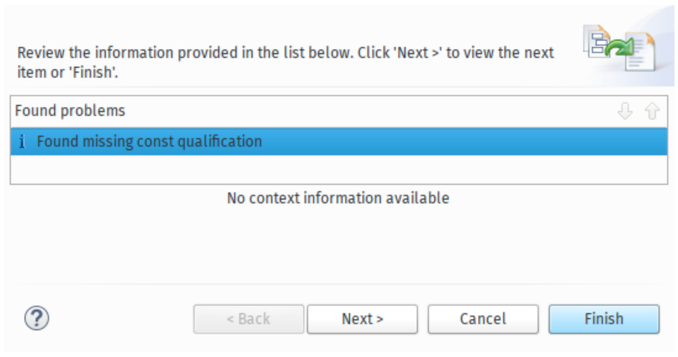
If you input the code in Listing 2.1 as a new file called *Structure.h* into the Cevelop IDE, you will notice that **Constificator** will **not** highlight the declaration of `Structure::doit()`. This is because of two reasons. First, the plug-in has not seen the definition for the function yet. Therefore it would be impossible to decide whether or not the function does something that would prevent it from receiving const-qualification. Secondly, the plug-in will currently only highlight definitions. This is because the definition is the *origin* of the "problem" and thus the fix should be applied from there.

```
#include "Structure.h"

void Structure::doit() {
  // Do nothing
}
```

Listing 2.2: Structure.cpp: The Implementation of `Structure::doit`

Now add the code in Listing 2.2 in a file called *Structure.cpp*. You will notice that **Constificator** will mark `Structure::doit` as eligible to receive const-qualification. This makes sense since `Structure::doit` neither modifies any *non-const non-static* data members of `Structure` nor calls any *non-const non-static* member functions. If you go ahead and apply the suggested fix like described in section 2.1 on the preceding page you will notice that **Constificator** now presents you with the dialog seen in Figure 4 on the next page.


Review the information provided in the list below. Click 'Next >' to view the next item or 'Finish'.

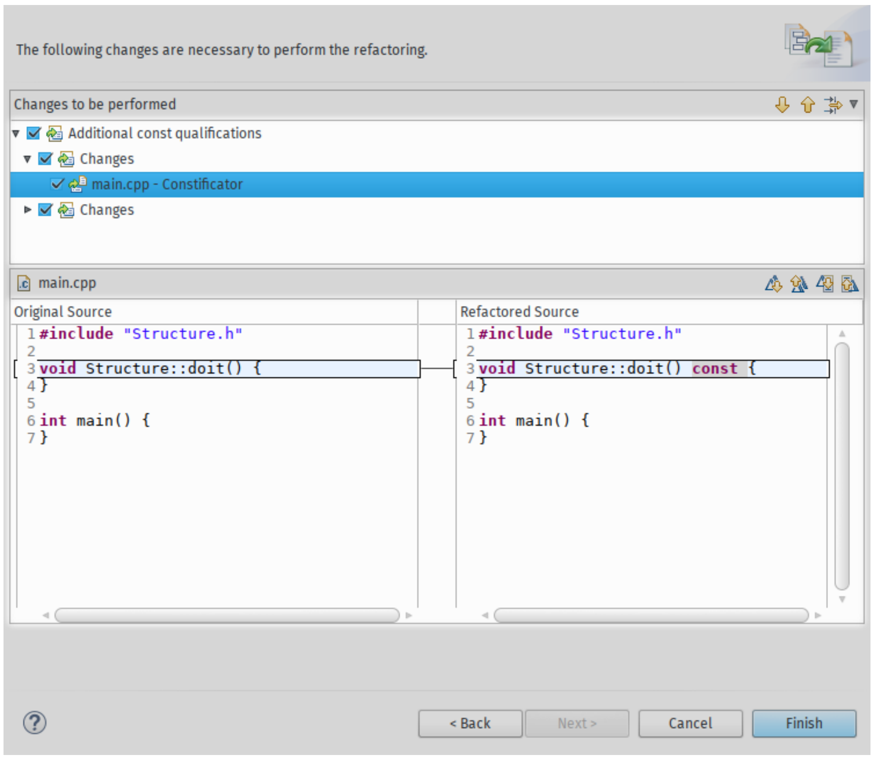
Found problems

Found missing const qualification

No context information available

Figure 4: Entry dialog for multiple changes

Every time a *QuickFix* modifies multiple locations in your project, even when all locations are in the same file, the plug-in presents you with a dialog like this one. Clicking on the Next > button will take you the next dialog as seen in Figure 5.


The following changes are necessary to perform the refactoring.

Changes to be performed

- ✓ Additional const qualifications
  - ✓ Changes
    - ✓ main.cpp - Constificator
  - ✓ Changes

main.cpp

Original Source

```
#include "Structure.h"

void Structure::doit() {
}

int main() {
}
```

Refactored Source

```
#include "Structure.h"

void Structure::doit() const {
}

int main() {
}
```

Figure 5: Overview of the changes to be applied

In addition to showing you a preview of the changes that will be applied, this dialog allows you to select which of them to actually apply. When you click on the Finish button, only the changes that are checked will be applied. You can cancel the process during each step using the Cancel button.

3 Revision History

Revision	Date	Author(s)	Description
1.0	16.01.2017	FM	Initial documentation