

GslAtrPtr

C++ Core Guidelines Pointer Checker and Support Library Refactorings

Bachelor Thesis

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF APPLIED SCIENCE RAPPERSWIL

Institute for Software

Spring Term 2016

Authors: Elias GEISSELER & Philipp MEIER
Advisor: Prof. Peter SOMMERLAD
External-Co-Examiner: Mr. Martin BOTZLER
Internal-Co-Examiner: Prof. Andreas STEFFEN

June 17, 2016

Abstract

The role of a raw pointer in legacy C++ is often ambiguous. It could represent a single object on the heap that must be deleted. It could represent an array of objects or it could represent a shared object or one not on the heap. In modern C++ smart pointers can help to alleviate that problem. However, raw pointers are still common in code, most notably within legacy systems or when programmers intend to stay resource efficient. This project wants to use the new ideas introduced with the C++ Core Guidelines [SS16] to help clear up this confusion of raw pointer roles in modern C++14 [ISO14] by giving developers easy to use refactoring tools.

The C++ Core Guidelines that got introduced at CppCon15 [mk15] aim to provide a set of guidelines on how to write better modern C++. They include several topics on how to handle memory and raw pointers. Following the guidelines allows for C++ code to be checked by static analysis tools and catch many semantic programming errors that are still common. Many of the guidelines make use of the Guidelines Support Library (GSL) [Mic16]. The GSL provides several types that should be used in context of raw pointers in code. The GSL types are recommended because they are more expressive than raw pointers and often contain additional functionalities that help prevent errors with minimal additional overhead.

The end product of this project is a plug-in for the Codeloop IDE [fS16] that helps developers to adhere to the C++ Core Guidelines in regards to raw pointers. The plug-in contains code checkers that mark problematic code using raw pointers and will then allow for refactorings to the GSL types marking their role. We have also tested our plug-in on a real world code base to ensure a better user experience and more reliable functionality.

Management Summary

The goal of this bachelor thesis is to build an Eclipse CDT [Fou16] plug-in that helps C++ developers to adhere to the C++ Core Guideline rules [SS16] related to raw pointers. Our thesis builds on the previous CharWars [GS14] thesis and their plug-in. CharWars was also concerned with refactoring raw pointers. In our plug-in we want to provide a set of checks and refactorings concerned with making raw pointers more expressive to programmers and therefore make code better.

Motivation

A raw pointer is a variable that holds the memory address of an object. Depending on the situation raw pointers can have different roles, however this role is not always clearly visible in code. The modern C++ standard [ISO14] introduced alternatives like smart pointers that can evade the problem.

However smart pointer can not be used in all situations, this is why the C++ Core Guidelines introduced rules to clearly mark what role a raw pointer has. To mark roles we use types from the Guidelines Support Library (GSL) [Mic16] that signify what role the raw pointer has. The long term goal of the guidelines is to modify the code so it can be checked by static analysis tools. These static analysis tools can then find complex logic errors that are hard to find manually.

Goal

For this thesis we focused on the C++ Core Guideline rules related to raw pointers. We want to provide a set of checks that find problematic code segments and a corresponding set of refactorings, helping programmers to move from legacy C++ code to safer and better code. We analyze the Guideline rules and types of the GSL, extracting the

core values and principles we can integrate into our plug-in. We extend the functionality of the CharWars plug-in by adding our checks and refactorings. Finally we test our checks and refactorings on a real life code base to improve their reliability and further test them.

Results

The end product of this bachelor thesis is an extension of the CharWars plug-in for the Cevelop IDE [fS16]. Our contribution can be divided into two parts: Ownership and nullability refactorings and span refactorings.

- **Ownership and nullability**

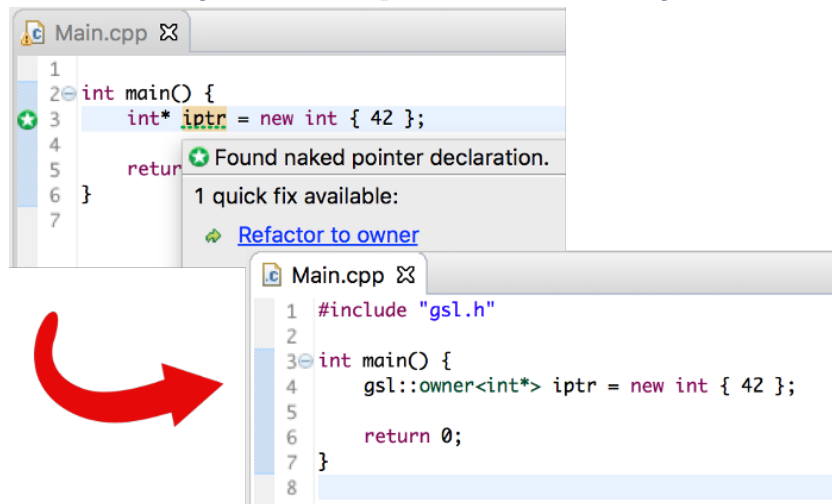
We implemented checks to effectively find raw pointers in code. When a raw pointer has been found, the programmer can then execute the appropriate refactoring to a GSL type, clarifying the role of the raw pointer. If the refactoring was made on a function definition then all the associated declarations will get searched and changed as well. By providing these reliable and easy to use refactorings, we help the programmer save time on repetitive tasks.

- **Span**

In C++, arrays are passed to a functions as raw pointers, pointing to the arrays first element. When doing so the size of the array gets lost and therefore needs to be passed to the function as a separate argument. We implemented checks that find these situations in code and offer a refactorings to the GSL `span<T>` type that encapsulates the raw pointer and its size. By putting together what belongs together code gets more readable, cohesive and safe.

In figure 0.1 we show how such a check marks problematic code and a refactoring resolves the situation.

Figure 0.1.: Example check and refactoring.



To make our plug-in fit for use we tested it on the Fish Shell project [fSh16]. As a result we improved our code to ensure a better user experience and more reliable functionality.

Further work

In this thesis we were able to extend the original CharWars plug-in [GS14]. Adding modern features to its catalog, while keeping the original functionality intact. However there are many cases where highly specific types need to be used, we were not able to cover all of them yet. Here are some additional features that could improve the plug-in:

- Improvement of the `span<T>` refactoring so it can handle more diverse function interfaces.
- Quick assist for `span<T>` in addition to the quick fix.
- Suppression of warnings via attributes.
- Checks and refactorings for `string_span`.

Declaration of Authorship

We declare that this bachelor 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 unauthorizedly used in this work.

Place and date

Elias Geisseler

Place and date

Philipp Meier

Contents

1. Introduction	5
1.1. Problem description	5
1.2. Solution	6
1.3. Previous work	7
1.4. Our goals	8
1.4.1. Planned Features	8
1.5. Time management	8
2. Analysis	9
2.1. Analysis of GSL types	9
2.1.1. <code>owner<T*></code>	10
2.1.2. <code>not_null<T*></code>	10
2.1.3. <code>span<T></code>	12
2.1.4. <code>string_span</code>	14
2.2. General concepts and challenges	15
2.3. Memory leaks and corruption	15
2.3.1. Ownership	17
2.3.2. Smart pointers	19
2.4. Analysis of C++ Core Guidelines	20
2.4.1. I.11: Never transfer ownership by a raw pointer (<code>T*</code>)	20
2.4.2. I.12: Declare a pointer that must not be null as <code>not_null</code>	23
2.4.3. I.13: Do not pass an array as a single pointer	25
2.4.4. I.24: Avoid adjacent unrelated parameters of the same type	26
2.4.5. F.22: Use <code>T*</code> or <code>owner<T*></code> to designate a single object	27
2.4.6. F.23: Use a <code>not_null<T></code> to indicate that "null" (meaning <code>nullptr</code>) is not a valid value	28
2.4.7. F.24: Use a <code>span<T></code> or a <code>span_p<T></code> to des- ignate a half-open sequence	28

Contents

2.5.	Additional considerations for our plug-in	29
2.5.1.	Adding the borrower type	29
2.5.2.	Including the GSL and gslrefactor.h in the CDT project	30
2.5.3.	Allowing the user to disable checks and warnings but still use our refactorings	31
2.5.4.	Configuring the GSL type names and other plug-in preferences	32
2.6.	Conclusion: How to proceed	33
2.6.1.	GSL types we focus on	33
2.6.2.	Checks to implement	33
2.6.3.	Refactorings to implement	34
2.6.4.	Configuration	34
3.	Implementation	35
3.1.	Overview of implemented features	35
3.1.1.	Checkers and problem markers	35
3.1.2.	Quick fixes	37
3.1.3.	Quick assists	39
3.1.4.	GSL project includer	41
3.1.5.	Preference page	42
3.2.	Abstract syntax tree	43
3.3.	Work flow when using checkers and quick fixes	44
3.4.	Our approach and solved challenges	45
3.4.1.	Ownership and nullability refactorings	45
3.4.2.	Architecture and the n^3 aspect problem	48
3.4.3.	Span refactorings	51
4.	Testing	53
4.1.	Automated testing	53
4.1.1.	Testing checkers	54
4.1.2.	Testing quick fixes	54
4.1.3.	Testing quick assists	55
4.2.	Manual testing	56
5.	Real world application	57
5.1.	Raw pointer problem statistic	57

5.2. Raw pointer + size problem statistics	59
5.3. Span in Fish Shell	59
6. Conclusion	61
6.1. Features	61
6.2. Relevance of our refactorings	61
6.3. Early redefining of task	62
6.4. Future work	62
A. User manual	63
A.1. Installation	63
A.2. Configuration	63
A.3. Refactorings	65
A.3.1. Ownership and nullability refactorings	65
A.3.2. Span refactorings	67
B. Developer manual	69
B.1. Version and software used	69
B.2. How to setup Eclipse PDE compile and run the tests	69

1. Introduction

This section outlines our bachelor thesis and our goals for it.

1.1. Problem description

In 2015, the Standard C++ Foundation, led by Bjarne Stroustrup, released an initial version of the C++ Core Guidelines document [SS16]. The document outlines a set of guidelines to help people use modern C++ more effectively. The guidelines are focused on issues such as interfaces, resource management, memory management and concurrency. They are designed to help the programmer write C++ code that is statically type safe and has no resource leaks, they can also help to catch common logic errors.

Alongside the guidelines, Microsoft released a small C++ library to aid the implementation of these guidelines, called the Guidelines Support Library (GSL) [Mic16]. Most notably for our thesis, the GSL provides types that help increase resource safety by implicitly annotating the intent of code. Meaning code can still use efficient raw pointers `T*` and references `T&`, while static analysis tools notify the developer when a misuse of pointers exists. The types GSL provides include `owner<T*>`, `not_null<T*>`, `span<T>` and `string_span`. Using these types instead of raw pointers and references also allows for static lifetime safety analysis of the code, as described in the paper Lifetimes I & II [SM15].

However, manually refactoring old C++ code to adhere to the C++ Core Guidelines and use the GSL types can be a time consuming and tedious task. The Institute for Software at HSR is contributor to the Cevelop IDE [fS16], this IDE is based on the Eclipse CDT IDE [Fou16]. There has been previous work creating various refactorings, but no GSL refactoring tools for Cevelop have been made yet.

1. Introduction

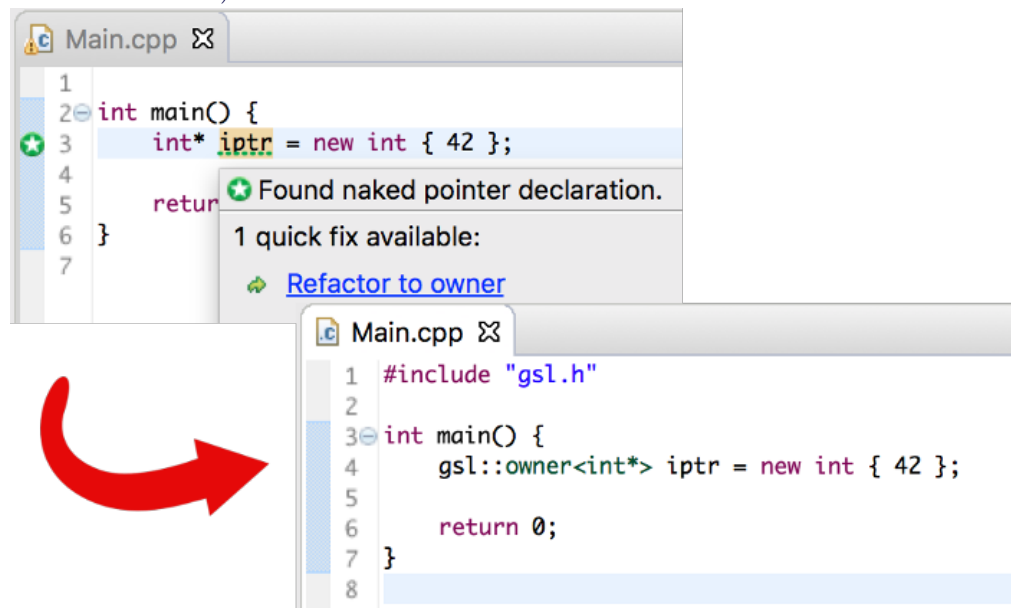
1.2. Solution

In order to make the task of refactoring old C++ code to adhere to the C++ Core Guidelines easier and less time consuming for the developer, we aim to develop a tool that finds problematic code not following the core guidelines and provides quick refactorings. This tool will be realized as a plug-in for Eclipse CDT.

The plug-in will perform static code analysis of C++ code on AST basis. The analysis can be triggered manually by the user, or can be set to run automatically whenever the code changes.

Problematic code snippets will be marked with a problem marker. The user can then choose to apply refactorings to the code, depending on the type of problem found. For example, a raw pointer declaration could be refactored to the GSL types `owner<T*>` or `not_null<T*>`.

Figure 1.1.: Example problem marker with quick fix refactoring. (Raw pointer to owner)



1.3. Previous work

There have been similar projects before. Specifically the term project Pointerterminator (2013) [GS13] by Fabian Gonzalez and Toni Suter, in which they developed an Eclipse CDT plug-in providing refactorings to get rid of pointers in C++. The plug-in is capable of the following refactorings:

- Replace C strings with `std::string` objects
- Replace C arrays with `std::array` objects
- Replace pointer parameters with reference parameters

In their bachelor thesis CharWars (2014) [GS14], Gonzalez and Suter have expanded the plug-in to allow better elimination of C strings, providing refactorings for many common C string functions to their corresponding `std::string` or `<algorithm>` counterparts.

The CharWars plug-in already has a lot of basic functionality that we can utilize in our plug-in. It uses problem markers, quick fixes and it already has an extensive array of functions for analyzing C++ code on AST basis and applying refactorings. Therefore, it makes sense for us to build upon the CharWars plug-in instead of starting from scratch.

It is also worth mentioning the Smartor (2013) [FM13] project by André Fröhlich and Christian Mollekopf. They developed a Eclipse CDT plug-in to refactor raw pointers to smart pointers such as `std::unique_ptr` and `std::shared_ptr`. Smart pointers often solve the same problems as the GSL types, such as leaks and dangling. However, sometimes the developer can not or does not want to use smart pointers. For example when dealing with legacy systems or when trying to be as resource efficient as possible. This is when our plug-in and the GSL types can be used.

1. Introduction

1.4. Our goals

Our goal is to advance the CharWars plug-in to include refactorings for the GSL types. We will first analyze the different types in the GSL as well as the corresponding C++ Core Guidelines. Our focus will be on CGL rules and GSL types regarding raw pointers. Then we will decide on refactorings that make sense to implement.

We will focus on simple refactorings that don't rely on complex flow analysis. This already provides value to users, helping them find problematic code such as raw pointers and quickly allowing them to choose an appropriate refactoring as a resolution.

1.4.1. Planned Features

- Find problematic raw pointers and mark them
- Provide refactoring from problematic raw pointers to GSL types
- Provide a solution to refactor a project step by step to adhere C++ Core Guidelines

1.5. Time management

Our project started on the 2016-02-22. It will end on 2016-06-17. The report and all of our other artifacts will be due at the end. This project is valued with 12 ECTS credits. 12 credits correspond to 360 hours of work per person and 720 hours for the team. In the last 2 weeks there are no classes, it is expected that we can work full time on this thesis during that time. This means we will work 40 hours per week and person on the last 2 weeks and an average of 18.6 hours per week and person on the remaining weeks.

2. Analysis

This chapter contains the analysis needed to understand the purpose of our project and to discover what should be implemented in our plug-in.

Since our plug-in aims to provide refactorings to GSLs [Mic16] pointer wrapper types, we will first analyze which GSL types exist and explore their purpose, usage and function.

We will then explain some of the general concepts and challenges that are important for understanding the problems associated with raw pointers.

We will also analyze some of the C++ Core Guideline [SS16] rules associated with raw pointers and the GSL types. We will evaluate which rules are suited to be considered in our plug-in and how they can be implemented as checks and refactorings.

Finally we have some additional considerations that we have to make for our plug-in.

The analysis concludes with a summary on how we want to proceed, giving the basis on what to implement in our plug-in.

2.1. Analysis of GSL types

In this section we will introduce the GSL types and explain their purpose and usage. The following GSL types will be covered:

- `owner<T*>`
- `not_null<T*>`
- `span<T>`
- `string_span` and its variations

2. Analysis

2.1.1. `owner<T*>`

GSLs `owner<T*>` is a pure marker/annotation type: It has no functionality whatsoever. (See listing 2.1) The purpose of `owner<T*>` is to mark pointers which have ownership of the indirected object, thus providing additional information that makes it easier to reason about how the pointer needs to be handled.

In order to follow some of the C++ Core Guidelines it is necessary to know if a pointer is an owner or not. Generally, a class or function that has an owner is responsible to free the allocated memory at the end. For example: If there is a class with an owner pointer as a member, one should make sure that `delete` is called in the destructor of that class. See section 2.3.1 for an explanation of the ownership concept.

Listing 2.1: Definition of GSLs owner type

```
template <class T>
using owner = T;
```

All raw pointers that are not marked with `owner<T*>` are considered non-owning per C++ Core Guidelines. This can be a problem when refactoring old code, since it is not possible to determine if a raw pointer has already been looked at. In order to solve that problem we introduce the `borrower<T*>` type to mark non owning pointers. It is the counterpart of `owner<T*>`. See section 2.5.1 for more information on the `borrower<T*>` type.

2.1.2. `not_null<T*>`

GSLs `not_null<T*>` type is used to declare pointers that should never be `nullptr` (`NULL / 0`). It is implemented in a way so that it is guaranteed to never be `nullptr`: Trying to set a `not_null<T*>` pointer to `nullptr` results in a compiler error. Note that `not_null<T*>` is not limited to raw pointers. It can be used for any value where `nullptr` is a relevant value, such as `std::unique_ptr` or `std::shared_ptr` [ISO14].

2.1. Analysis of GSL types

Typically, `not_null<T*>` is used for function parameter and return types in order to indicate that null is not a valid value. (See listing 2.2)

Listing 2.2: Typical usage of `not_null<T*>`

```
void justDoIt(not_null<Thing*> p); // justDoIt(nullptr) is invalid,  
    it is the callers job to make sure the parameter p is not null  
  
not_null<int*> calculateIt(); // calculateIt() is guaranteed to never  
    return null
```

Using `not_null<T*>` reduces overhead by eliminating the need to check against `nullptr` before using a pointer. It also reduces the possibility of a programming error where someone might forget such a check, as suggested by the rule F.23 in section 2.4.6. Smart pointers, described in section 2.3.2, can also be used with the `not_null<T*>` type.

See section 2.4.2 for more information of how `not_null<T*>` can be used to clarify ambiguity in interfaces.

We conclude that our plug-in should provide a refactoring for raw pointers to `not_null<T*>`.

References as an alternative to `not_null<T*>`

References (e.g. `T&`) are similar to `not_null<T*>` pointers. Both of them can not be null. The differences are that pointers can be repointed/rebinded (for example through pointer arithmetic), while references are fixed to point to the same object after initialization. Additionally, `not_null<T*>` can be used for other `nullptr`-assignable types, such as `std::unique_ptr` or `std::shared_ptr`, which could not be substituted with a reference that easily.

The CharWars plug-in already provides a checks and refactorings to transform pointer parameters to reference parameters. No additional refactorings for other pointer occurrences, such as return types or member variables, are available though.

2. Analysis

2.1.3. `span<T>`

The `span<T>` type is used to represent a non-owning range of contiguous memory, such as an array, pointer (with size) or `std::vector`. [Ban16a]

One of the problems of using plain arrays (e.g. `int[42]`) is that the size is lost when passing it as a function argument (array to pointer decay). Therefore, a second parameter for the size is needed. This can lead to errors when the size is incorrect. `span<T>` alleviates this issue by storing both a pointer to the data as well as the size information. It is bounds checked and safe. `span<T>` also removes the need to implement different function overloads for different container types and arrays/pointers, since they all can be converted to spans easily. Another advantage of `span<T>` is that it can be easily traversed using range based for loops. (e.g. `for(auto x : mySpan)`)

Creating a `span<T>` is easy. There are constructors for arrays, pointer + size, begin + end pointer and standard containers with contiguous memory like `std::vector`. (See listing 2.3)

Listing 2.3: A few examples of different ways to create a `span<T>`

```
void displayNumbers(span<int> numbers) {
    //...
}

int numArr[] = {1, 2, 3};
span<int> spanNumArr = span{numArr}; // = {1, 2, 3}, when constructing
    from a plain array, size is automatically deduced
span<int> spanNumArrWithSize = span{numArr, 2}; // = {1, 2}
displayNumbers(numArr); // = {1, 2, 3}, implicit call of constructor
    from plain array

vector<int> numVec = {1, 2, 3, 4};
span<int> spanNumVec = span{numVec}; // = {1, 2, 3, 4}
span<int> spanNumVecFromPointer = span{numVec.data(), 4}; // = {1, 2,
    3, 4}, when constructing from pointer, size is required
span<int> spanNumVecFromPointerRange = span{&numVec[1], &numVec[2]};
    // = {2, 3}
```

Our plug-in will try to find function definitions that consist of pointer + size parameters and will provide a refactoring to change them to `span<T>`.

2.1. Analysis of GSL types

The refactoring will just generate a trampoline function and adjust the call sites to use that new trampoline function. A trampoline function is a function overload, with `span<T>` parameter instead of pointer + size, that just calls the original function. This is much easier to implement than trying to completely refactor the original function and its body to `span`, which can be done in a later step once all the call sites have been adapted.

An example of such a refactoring with a trampoline function can be seen in listings 2.4 and 2.5.

Listing 2.4: Before refactoring

```
void displayNumbers(int* numbers,
                   int numberCount) {
    //...
}

int main() {
    int [] numArr = {13, 42, 777};
    displayNumbers(numArr, 3);

    return 0;
}
```

Listing 2.5: After refactoring

```
void displayNumbers(int* numbers,
                   int numberCount) {
    //...
}

// generated trampoline function
void displayNumbers(span<int>
                   numbers) {
    displayNumbers(numbers.data(),
                   numbers.size());
}

int main() {
    int [] numArr = {13, 42, 777};
    displayNumbers(gsl::span<int> {
        intarr, 3 });

    return 0;
}
```

Using `std::array<T>` as an alternative to `span<T>`

Instead of using `span<T>` to encapsulate plain arrays one could also use `std::array<T>` instead of plain arrays and pass a reference to it. CharWars already provides refactorings to change plain C-style arrays into `std::array<T>`.

2. Analysis

2.1.4. `string_span`

Although regular `span<T>` can be used with C-strings, the GSL provides different span implementations for various types of strings. [Ban16b] These `string_span` types provide the same benefits as `span<T>`, such as bounds checks, plus some additional string specific functionality.

The following `string_span` types are available:

- `string_span` for string spans of `char`
- `cstring_span` for string spans of `const char`
- `wstring_span` for string spans of `wchar_t`
- `cwstring_span` for string spans of `const wchar_t`
- `zstring_span` for nul-terminated string spans of `char`
- `czstring_span` for nul-terminated string spans of `const char`
- `wzstring_span` for nul-terminated string spans of `wchar_t`
- `czwstring_span` for nul-terminated string spans of `const wchar_t`

The terminology is:

- c = const
- w = wide (e.g. `wchar_t`)
- z = nul-terminated (C-style char array ending with `\0`)

When creating a `string_span` from a C-style char array/pointer, the function `gsl::ensure_z()` needs to be used. A `string_span` can be easily converted to a `std::string` using the `gsl::to_string()` function. (See listing 2.6)

Listing 2.6: Example usage of `ensure_z()` and `to_string()`

```
char * cstring = "Hello";

gsl::string_span<> stringspan = gsl::ensure_z(cstring);

std::string stdstring = gsl::to_string(stringspan);
```


2.2. General concepts and challenges

This chapter contains explanations of general concepts and challenges that are relevant for this thesis. We will explain memory leaks and memory corruption, that can occur when misusing pointers. Additionally we explain the concept of ownership and how it can help to prevent leaks and dangling. We will also take a quick look at smart pointers, as they are often a better alternative to solve the problems discussed in this paper.

2.3. Memory leaks and corruption

One of the oldest programming challenges is to manage heap memory. In languages like Java or C# a garbage collector takes care of freeing unused heap memory. Garbage collection is helpful but costs a lot of resources and is nearly unpredictable in a real time setting. In C++, programmers have to handle allocating and freeing memory manually or use types or libraries to help them, like smart pointers for example (part of the standard library). Manually managing the memory is not an easy task and the resulting bugs can be devastating and hard to track down.

Memory leaks

A memory leak occurs when allocated heap memory is not freed properly. As seen in the example in listing 2.7, memory is allocated by calling `new` but is never freed by calling `delete`.

Listing 2.7: Example code that can cause memory leaks

```
void doIt() {  
    int *i = new int{42}; // local pointer variable is lost at the end  
                        // of the function -> will probably never be deleted causing a memory  
                        // leak  
}
```

2. Analysis

Memory corruption

Memory corruption occurs when memory of the wrong object or no object is changed.

Common errors that can cause memory corruption include [Ipp16]:

- out of bounds access of an array, pointer, buffer etc.
- using a pointer that has already been freed (dangling pointer)
- freeing memory that has already been freed (double delete)
- using an address before memory is allocated and set
- invalid object access (wrong casts)
- exception errors (e.g. memory is never allocated because of an exception)

See listing 2.8 for examples of code that can cause memory corruption.

Listing 2.8: Example code that can cause memory corruption

```
void doIt(int *i){
    delete i;
}

void doArray(int *a){
    a[5]++; // out of bounds access: there is no int at this index,
           // there could be another object at this location
}

void foo(){
    int *i = new int{0};
    int *a = new int[2]{};
    doIt(i);
    doArray(a);
    *i++; // dangling pointer: i doesn't exist anymore, another object
         // could be at this address now
    delete i; // double delete
}
```

2.3.1. Ownership

Whenever we use heap memory and initialize a pointer using `new`, it becomes the question of who is responsible to delete that pointer once it is not needed anymore. Because we do need to delete it in order to prevent memory leaks, but we also should never delete it more than once to avoid causing undefined behavior [ISO14].

Take the code in listing 2.9 for example.

Listing 2.9: Ownership example

```
class CakeMachine {
public:
    CakeMachine();
    ~CakeMachine();

    Cake* bakeCake();
}

class CakeMonster {
public:
    CakeMonster();
    ~CakeMonster();

    void eatCake(Cake* cake);
}
```

The `CakeMachine.bakeCake()` function creates a `Cake` object and returns a pointer to the instance. This pointer can then be passed to the `eatCake()` method of the `CakeMonster` class for example.

The code in listing 2.10 shows an example of how these classes could be used.

Listing 2.10: Ownership example usage

```
CakeMachine cmachine{};
CakeMonster cmonster{};

Cake* cake = cmachine.bakeCake();
cmonster.eatCake(cake);
```

2. Analysis

The problem with raw pointers becomes apparent quickly. Who is responsible for deleting the pointer returned by `bakeCake()`?

- Does `CakeMachine` delete the pointer in its destructor? This could result in using a dangling pointer if the return value of `bakeCake()` is used after the generating `CakeMachine` instance is destroyed.
- May the call of `CakeMonster.eatCake()` delete it? This could cause problems if the pointer is not allocated on the heap or deleted twice. Or not deleted at all, resulting in a leak.
- Or maybe we have to delete it ourselves? If we forget it would cause a leak.

Here is where the concept of ownership comes in. The owner of the pointer is the one who is responsible for deleting it. If the code would be annotated using GSLs `owner<T*>` type for example, we could make out who is responsible for the deletion.

- If `bakeCake()` does not return an owner, it means that `CakeMachine` is responsible for deleting the pointer.
- If `bakeCake()` does return an owner, it means that ownership is transferred to us, the caller of the function, and that we are responsible for the deletion of the pointer.
- If then `CakeMonster.eatCake()` takes an owner as parameter, it means we are transferring ownership to the `CakeMonster` class and it is now responsible for the deletion. [Lav13]

The whole ownership problem could be avoided by using smart pointers `std::unique_ptr` (single ownership) and `std::shared_ptr` (shared ownership). See section 2.3.2 for a brief explanation on how these smart pointers work.

There are however cases where smart pointers can not be used, for example when dealing with legacy systems that have raw pointer APIs. This project aims to provide assistance in those cases. For smart pointer refactorings check out the Smartor project. [FM13]

2.3.2. Smart pointers

Modern C++ allows for easier memory management by using smart pointers. Smart pointers can help to prevent memory leaks, memory corruption and also help clarifying ownership. This chapter will explain the most commonly used smart pointers that have been introduced with C++11: `std::unique_ptr` and `std::shared_ptr`. [ISO14]

std::unique_ptr

`std::unique_ptr` is a container that encapsulates a raw pointer. It takes ownership over the encapsulated pointer, so it will make sure the encapsulated pointer gets deleted at the end of its lifetime. A `unique_ptr` cannot be copied. It can however be moved by using `std::move()`, this would signify a transfer of ownership to another `unique_ptr`. [Lav13]

The code from listing 2.9 could be improved to use `unique_ptr` to clearly indicate ownership. (See listing 2.11)

Listing 2.11: Ownership example with `unique_ptr`

```
class CakeMachine {
public:
    CakeMachine();
    ~CakeMachine();

    std::unique_ptr<Cake> bakeCake();
}
```

By returning a `unique_ptr`, it is clear that ownership is transferred to the caller of `bakeCake()` and it is now responsible for whatever happens to it. There should not be a copy stored in `CakeMachine` because it is unique and cannot be copied.

std::shared_ptr

`std::shared_ptr` is a container that encapsulates a raw pointer. It shares ownership of the encapsulated pointer with all the copies of the `shared_ptr`. It uses reference counting to ensure that the contained raw pointer is deleted once all the copies of the `shared_ptr` have been destroyed. This reference counting mechanism however causes a little bit of overhead but makes things very convenient for the developer, as he does not need to worry about manually deleting the pointer.

2. Analysis

2.4. Analysis of C++ Core Guidelines

In the following chapters we want to give our interpretation of the guidelines related to our project, show what problems they want to solve and how we can use this to create more robust refactorings. The chapters are linked to C++ Core Guidelines [SS16], we recommend the reader to be familiar with the respective guidelines as the aim of this chapter is not to repeat their content fully but rather to give our interpretation and the implications for our plug-in.

2.4.1. I.11: Never transfer ownership by a raw pointer (T*)

In a function interface with only a raw pointer it is ambiguous whether the caller or callee is the owner. This ambiguity leads to misuse of the interface and this in turn leads to memory leaks and memory corruption, as explained in section 2.3. For example the user might not delete a given pointer thinking it is taken care of by an other object, this leads to memory leaks. Consider the following code:

Listing 2.12: Bad interface

```
Result* doCalculation(ArithmeticExpression e){
    Result* result = new Result{e};
    ...
    return result;
}
```

Without looking at the body of the function it is not clear whether that returned raw pointer `Result*` needs to be deleted by the caller or if it is managed by a different object entirely. Since we can see the body of the function we can conclude that the ownership of `Result*` is transferred to the callee of the function and therefore needs to be deleted by the callee.

2.4. Analysis of C++ Core Guidelines

To get rid of the ambiguity there are several possibilities. (See listing 2.13)

Listing 2.13: Good interfaces

```
//return by value
Result doCalculation(ArithmeticExpression e);

//return by using a smart pointer
std::unique_ptr<Result> doCalculation(ArithmeticExpression e);
std::shared_ptr<Result> doCalculation(ArithmeticExpression e);

//return by marked pointer
gsl::owner<Result*> doCalculation(ArithmeticExpression e);
gsl::borrower<Result*> getResultFromLibrary(ArithmeticExpression e);
```

Returning by value is the most robust way, since without having pointers there is no need of managing memory manually. Smart pointer come with a built-in mechanism for managing memory and therefore handle the cleaning up for the user. Using `unique_ptr` makes it clear that the callee will be the owner of the object. With `shared_ptr` ownership will possibly be shared between multiple objects, where the last releasing the pointer will free the memory. The easy handling of `shared_ptr` comes at the cost of resources, it costs more memory and more computational resources than a raw pointer.

We still can and want to use raw pointers, for example in some environments we want to be as resource efficient as possible or we are stuck with the interface because of compatibility reasons. Here we can use the type aliases `owner<T*>` and `borrower<T*>` introduced in section 2.1.1. Since they are only aliases they are reduced to raw pointers at compile time, yet they help to clarify the intent of the interface. Using `owner<T*>` the interface signifies a transfer of ownership, meaning the callee will need to delete the returned object. When receiving a `borrower<T*>` there is no need to delete the object since it will be handled by someone else.

We conclude that the ambiguity of ownership is a key problem of raw pointers. By marking pointers as owner or borrower we can communicate the underlying ownership and help developers and static analysis tools to find misuse of pointers.

2. Analysis

Implications for plug-in

In order to make code more safe and less ambiguous it is recommended to mark every owner pointer in the code base as owner or borrower. We introduced the concept of borrower to make a incremental transition easier. To help the developers marking raw pointers with `owner<T*>` or `owner<T*>` we want to provide them with easy to use code checks and refactorings.

We have determined the following circumstances in which a raw pointer declaration might occur (See listing 2.14):

- as a local variable
- as a global variable
- as a member variable of a class
- as a parameter
- as a return type
- in a template argument
- in a function pointer
- in a lambda function

Listing 2.14: Examples of possible pointer declarations

```
T* gp; //global pointer

void bar(T* p); //parameter pointer

T* foo(){ //return type pointer
    T* p; //local pointer
    std::vector<T*> vp; //template argument pointer
    void (*fp)(T*) = bar; //function pointer containing pointer
    auto func = [] (int* a) { *a = *a + 1; }; //pointer in lambda
    function
    return p;
}

struct Barbarossa{
    T* p; //member pointer
};
```

We will not consider function pointers and lambdas in our refactorings since they are a rather advanced subject and we want to focus on the basics first.

We conclude that the plug-in should mark raw pointers and provide refactorings to either `owner<T*>` or `borrower<T*>`.

2.4.2. I.12: Declare a pointer that must not be null as `not_null`

Dereferencing a pointer that is a `nullptr` leads undefined behavior, which most of the time means that the application crashes, or even worse, that memory just gets modified or read out in its unknown state, causing memory corruption. To mitigate the problem, programmers check pointers before accessing them. But each such `nullptr` check costs a bit of processing time, so often we transfer the responsibility of checking for `nullptr` to the caller of our interface. The problem is that this transfer of responsibility is not natively clear in C++, meaning it is not possible to know whether passing a `nullptr` is allowed by just looking at the function signature.

Listing 2.15: Ambiguity if `nullptr` is allowed

```
int countDepth(Node const * n);
// ...
int main(){
    countDepth(nullptr);
}
```

Take the code in listing 2.15 for example. It is ambiguous if the call `countDepth(nullptr)` is within the specifications or not. As seen in section 2.4.1 ambiguity can lead to mistakes. This is why we use `not_null<T*>` to indicate clearly that the responsibility of not passing a `nullptr` is given the caller. If the `not_null<T*>` is the return type then the interface takes responsibility of never passing `nullptr` back. If there is no `not_null` used we can assume that passing `nullptr` is allowed.

Implications for plug-in

By giving refactorings for wrapping pointers in `not_null<T*>` we want to help developers transition to safer code that follows the core guidelines.

2. Analysis

Why there is no nullable

At the beginning of our thesis we expected there to be some sort of `nullable<T*>` type introduced by us accompanying `not_null<T*>`, analogue to `owner<T*>` and `borrower<T*>` where one excludes the other.

If we assume that a code base is moving from non guideline code to safe guideline code incrementally and that a developer is deciding whether or not a pointer is an owner or a borrower also decides if it is `not_null<T*>` at the same time, then we can conclude that a pointer will be left in one of the five states listed in listing 2.16.

Listing 2.16: Possible states of raw pointers before and after refactoring

```
//unrefactored -> was not looked at yet
void something(T* a);

//owner -> pointer could be nullptr, is owning
void something(gsl::owner<T*> a);

//borrower -> pointer could be nullptr, is not owning
void something(gsl::borrower<T*> a);

//not_null owner -> pointer is never nullptr and is owning
void something(gsl::not_null<gsl::owner<T*>> a);

//not_null borrower -> pointer is never nullptr and is not owning
void something(gsl::not_null<gsl::borrower<T*>> a);
```

Adding a nullable alias to the above code would add no additional information and therefore would just clutter up the code more.

If the assumption that a programmer will do both refactorings for a raw pointer at the same time is not holding up, then this scheme above would not work. However, we feel that a programmer getting introduced to `owner<T*>` also gets introduced to `not_null<T*>` since they are a big part of the C++ Core Guidelines, GSL and the accompanying talks.

2.4.3. I.13: Do not pass an array as a single pointer

One of the problems of C-style arrays is that they degenerate to pointers when passed to function as a parameter. When this happens the size of the array is lost. To circumvent this issue, developers usually add a second parameter for the size to the function interface.

Listing 2.17: Example of a function interfaces taking pointer + size parameters

```
void calcSum(int* arr, int size);
void copy_n(T* p, T* q, int n);
```

Function interfaces like this are prone to errors. What if there are less elements in the array than specified by the size parameter? We would probably read or overwrite unrelated memory, which leads to undefined behavior [ISO14].

A better alternative is to use GSLs `span<T>` or `string_span` types. That way it is clear that a range of elements is expected. `span<T>` also contains functionality to automatically deduce the element type and number of elements in its constructors. (See section 2.1.3)

Listing 2.18: Example of better interfaces using GSL `span<T>`

```
void calcSum(span<int> arr);
void copy_n(span<T> p, span<T> q);
int[5] iarr;
//...
calcSum(iarr); // automatically deduce type and size
```

Implications for plug-in

We will have to implement a check to find function interfaces that consist of pointer + size parameters. For this analysis we have to consider the different possibilities for pointer + size, such as multiple pointers but only a single size. (As seen in the above example in the `copy_n()` function)

2. Analysis

We will then have to implement refactorings to change those function interfaces to use `span<T>` or `string_span` instead. All the call sites to said functions need to be adjusted too. In order to avoid having to refactor the whole function body, we will first generate a trampoline function. (See listing 2.20) Later we will also try to provide the option the refactor the function body if possible.

Listing 2.19: Before refactoring

```
void displayNumbers(int* numbers,
                   int numberCount) {
    //...
}

int main() {
    int[] numArr = {13, 42, 777};
    displayNumbers(numArr, 3);

    return 0;
}
```

Listing 2.20: After refactoring

```
void displayNumbers(int* numbers,
                   int numberCount) {
    //...
}

// generated trampoline function
void displayNumbers(span<int>
                   numbers) {
    displayNumbers(numbers.data(),
                   numbers.size());
}

int main() {
    int[] numArr = {13, 42, 777};
    displayNumbers(gsl::span<int> {
        intarr, 3 });

    return 0;
}
```

2.4.4. 1.24: Avoid adjacent unrelated parameters of the same type

Adjacent parameters of the same type can be easily swapped by mistake. This is can be especially problematic when using pointers to ranges, since an out of bounds access will result in undefined behavior.

Listing 2.21: Problematic interface with adjacent parameters of the same type

```
void copy_n(T* p, T* q, int n); // copy from [p:p+n) to [q:q+n)
```

Consider the function interface in listing 2.21. It's easy to reverse the "to" and "from" arguments.

2.4. Analysis of C++ Core Guidelines

A better alternative would be to use GSLs `span<T>` type or to use a parameter struct with appropriately named parameters.

Implications for plug-in

Our plug-in will help implementing this rule by providing refactorings from pointer + size interfaces to GSLs `span<T>`. See section 2.4.3 about rule I. 13 for more information.

2.4.5. F.22: Use `T*` or `owner<T*>` to designate a single object

In traditional C and C++ code raw pointers (`T*`) are used for many purposes, such as to [SS16]:

- Identify a (single) object (not to be deleted by this function)
- Point to an object allocated on the free store (and delete it later)
- Hold the `nullptr`
- Identify a C-style string (nul-terminated array of characters)
- Identify an array with a length specified separately
- Identify a location in an array

This rule says to only use raw pointers to designate single objects. For ranges such as arrays, use `span<T>`. For C-style strings user `string_span<T>` or `zstring`.

Implications for plug-in

Our plug-in will help implementing this rule by providing refactorings from pointer + size interfaces to GSLs `span<T>`. See section 2.4.3 about rule I. 13 for more information.

2. Analysis

2.4.6. F.23: Use a `not_null<T>` to indicate that "null" (meaning `nullptr`) is not a valid value

This rule is similar to rule I.12 (See section 2.4.2). It says to use `not_null<T*>` to indicate when `nullptr` is not a valid value. That way it's clear to the user of an interface if he needs to check against `nullptr` before dereferencing a pointer for example.

Implications for plug-in

Our plug-in will help implementing this rule by providing refactorings for raw pointers to GSLs `not_null<T>`. See section 2.4.2 about rule I.12 for more information.

2.4.7. F.24: Use a `span<T>` or a `span_p<T>` to designate a half-open sequence

Similar to rules I. 13 and I. 24, this rule says to use `span<T>` instead of raw pointers (+ size) to designate ranges.

Implications for plug-in

Our plug-in will help implementing this rule by providing refactorings from pointer + size interfaces to GSLs `span<T>`. See section 2.4.3 about rule I. 13 for more information.

2.5. Additional considerations for our plug-in

In order to make our plug-in more robust and user friendly we had to make a couple of additional considerations that are not directly tied to a GSL type or CGL rule.

2.5.1. Adding the borrower type

The GSL provides types used for writing code according to the C++ Core Guidelines [SS16]. However, it tries to be as resource small and lightweight as possible. This means that only a type alias for owning pointers exists and that every raw pointer not declared as such will be regarded as non-owning. Although this is fine for a code base already following the guidelines, it is problematic when migrating code to follow the guidelines.

To apply the guidelines to old code, the developers need to revisit all the uses of raw pointers and decide whether or not it is an owner. To do these refactorings step by step, developers need to be able to differentiate already refactored code from code that has not been looked at yet. This is not possible with only the GSL types, since it is not possible to know if a raw pointer has already been decided to be non-owning or if it just has not been looked at yet.

To solve this problem we introduce the type alias `borrower<T*>` as counterpart of `owner<T*>`. The type is defined in our own header file called *gslrefactor.h*. The header file also includes the rest of the GSL so that `#include "gslrefactor.h"` is sufficient to have access to all of the GSL types.

2. Analysis

Listing 2.22: Content of `gslrefactor.h`

```
/*
 * gslrefactor.h
 *
 */

#ifndef GSLREFACTOR_H
#define GSLREFACTOR_H

#include "gsl.h"

namespace gsl
{
//
// GSL.borrower: non owning pointer
//
template <class T>
using borrower = T;
}

#endif // GSLREFACTOR_H
```

2.5.2. Including the GSL and `gslrefactor.h` in the CDT project

In order use the GSL types and `borrower<T*>`, the `gslrefactor.h` header file needs to be included with a `#include` statement. This statement should be automatically by our refactorings whenever one of them is applied.

For the code to compile, `gslrefactor.h` and the GSL files need to be lying somewhere on the file system and need to be added to the include path of the CDT C++ project, so the compiler knows where to find them. Ideally, our plug-in should be able to ensure that those needed files are available and automatically add the references to the C++ project whenever a refactoring is applied.

2.5. Additional considerations for our plug-in

In order to this we would have to bundle a recent version of the GSL with the plug-in or implement a mechanism to download it from the Internet. Downloading it from the Internet is rather sophisticated and could break our plug-in if it changed too much from the version we developed the plug-in for, so bundling a version with the plug-in appears to be the better option. However, we need to make sure that the developer is still be able to use his own GSL instead of the bundled one if he wants to.

2.5.3. Allowing the user to disable checks and warnings but still use our refactorings

In the talk "CppCon 2015:Static Analysis and C++" by Neil MacIntosh [Mac15], he explains that code checks being too noisy, by generation a lot of warnings, is one of the common causes a user might not use the code check at all and deactivate it. This is especially true when the checks are used in a legacy code project for the first time, because there would be a lot of problematic code sections.

For this reason we want to ensure that a user can disable the warnings and checkers but can still use the refactorings provided by our plug-in, like refactoring a raw pointer to `owner<T*>` for example.

2. Analysis

2.5.4. Configuring the GSL type names and other plug-in preferences

Since the GSL is relatively new and in an early stadium, the names of the types could still be subject to change in a future version. To circumvent this problem, our supervisor Prof. Sommerlad suggested that we make the type names configurable. This would also have the added benefit of allowing the user to use his own types instead of the GSLs if he desires.

It would also be nice if the include functionality outlined in the previous section is configurable as well. Like being able to disable the automatic includes or to change the name of the header file that is included. That way the user is able to use his own header file.

Another thing to be configured are the types that should be considered size types for the `span<T>` refactorings since the user might use special size types in his code and not just `int`, `long`, `char`, etc.

This whole plug-in configuration would be best realized as a preference page directly in the Eclipse preferences dialog.

2.6. Conclusion: How to proceed

Concluding our analysis we want to give a quick summary of the results and outline what features we want to implement in our plug-in.

2.6.1. GSL types we focus on

We have decided to focus on providing refactorings for the following types:

- `owner<T*>`
- `borrower<T*>`
- `not_null<T*>`
- `span<T>`

We have not included `string_span` because we think that the other types are more important and we want to focus on those in the limited time we have. CharWars already provides great refactorings for C-style strings to `std::string`, which can often be used as an alternative to `string_span`.

2.6.2. Checks to implement

To implement those refactorings we first need to implement checks that find raw pointers in code and mark them. For these checks we will consider the following circumstances where raw pointer might occur:

- as a local variable
- as a global variable
- as a member variable of a class
- as a parameter
- as a return type
- in a template argument

Additionally, we need checks that find raw pointer + size combinations in function parameters for the `span<T>` refactoring. We will first focus on simple function interfaces and then try to improve the checks to find more complex ones.

2. Analysis

2.6.3. Refactorings to implement

We need to implement the actual refactorings to convert those raw pointers into GSL types. Since we want to provide combined refactorings for `not_null<T*>` with either `owner<T*>` or `borrower<T*>` we get the following list of refactorings:

- raw pointer to `owner<T*>`
- raw pointer to `borrower<T*>`
- raw pointer to `not_null<T*>`
- raw pointer to `not_null<owner<T*>>`
- raw pointer to `not_null<borrower<T*>>`
- raw pointer + size parameter to `span<T>`

The following points need to be considered when implementing the refactorings:

- When refactoring function interfaces, make sure that all the declarations of the function and the call sites are also refactored.
- Allow the refactorings to be used even when checks and warnings are disabled.
- Make sure the *gslrefactor.h* file gets included when a refactoring is applied.
- Make sure the GSL files and *gslrefactor.h* are made available and are referenced in the projects include path when a refactoring is applied.

2.6.4. Configuration

We want to allow the user to configure certain aspects of the plug-in, such as the GSL type names and the include functionality. For this we need to implement our own preference page and add it to the Eclipse preference dialog.

3. Implementation

In this chapter we will explain how we implemented the checks and refactorings outlined in the analysis as checkers, quick fixes and quick assists. We show how we extended the existing CharWars [GS14] plug-in infrastructure and what considerations were made to achieve our goals. We also want to show the challenges that came up during implementation and our approach to solve them.

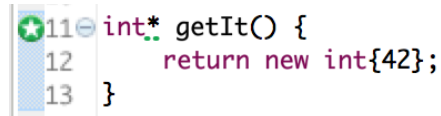
3.1. Overview of implemented features

Here we give an overview of the different features that have been implemented and how they work. When possible, we will also show how the implemented feature can help to adhere to the C++ Core Guidelines analyzed in the analysis chapter.

3.1.1. Checkers and problem markers

Static analysis of the code is done by checkers. Checkers in Eclipse CDT can be run on demand or whenever a change is made in the document. They traverse and analyze the code using the AST (See section 3.2). If a checker finds code that meets its criteria, it marks the corresponding AST node with a problem marker.

Figure 3.1.: A problem marker indicating a raw pointer in a return type.

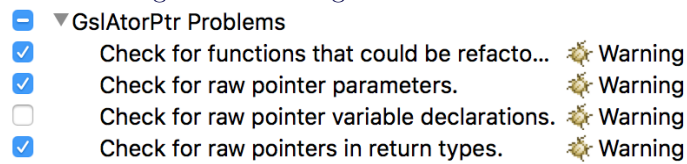


```
11 int* getInt() {  
12     return new int{42};  
13 }
```

Each checker is responsible for a specific problem (e.g. raw pointer variable declaration, raw pointer in return type, etc.). The checkers can be configured for the whole workspace or on a per project basis under the "Code Analysis" tab in the Eclipse preferences dialog, allowing the user to enable or disable each specific checker.

3. Implementation

Figure 3.2.: Configuration of checkers.



In the upcoming sections we will quickly go over every checker we have implemented. They all can find different positions as seen in the "Implications for plug-in" section of 2.4.1. By finding most positions we can provide a fast way to clear up ambiguous sections of code. We will also show how the checkers map to the positions described in the analysis.

Raw pointer variable declaration checker

This checker can find raw pointers in variable declarations. This means it can find raw pointers in the following situations:

- local pointer variable
- global pointer variable
- member pointer variable
- pointer in a template argument of a local, global or member variable

Raw pointer parameter checker

This checker can find raw pointers in parameters of function definitions. This includes template arguments of these types.

Raw pointer return type checker

This checker can find raw pointers in return types of function definitions. This includes template arguments of these types.

3.1. Overview of implemented features

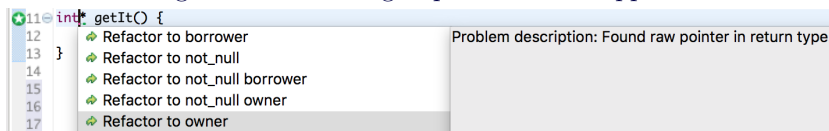
Raw pointer parameter + size to `span<T>` checker

This checker is searching for raw pointers in parameters and a follow up size parameter. It finds a set of the situations described in section 2.4.3. The current checker is fairly limited: It is only able to recognize functions interfaces where the size parameter comes directly after the raw pointer parameter, improving this checker would be a good topic for future work. This checker is used for the `span<T>` refactoring.

3.1.2. Quick fixes

Quick fixes are the actual refactorings that can be applied via a problem marker. Each problem reported by a checker can have several quick fixes associated with it (e.g. `owner`, `borrower`, etc.). When the user applies a quick fix the AST is modified to the newly wanted structure.

Figure 3.3.: Selecting a quick fix to be applied.



The quick fixes in our plug-in can be divided in 2 categories: Quick fixes for ownership and nullability and quick fixes for raw pointer + size parameters to `span<T>`.

Ownership and nullability quick fixes

This category includes the following quick fixes:

- raw pointer to `owner<T*>`
- raw pointer to `borrower<T*>`
- raw pointer to `not_null<T*>`
- raw pointer to `not_null<owner<T*>>`
- raw pointer to `not_null<borrower<T*>>`

3. Implementation

These quick fixes are available on the problem markers generated by these checkers:

- raw pointer variable declaration checker
- raw pointer parameter checker
- raw pointer return type checker

This category of quick fixes can be used to adhere to the following core guideline rules:

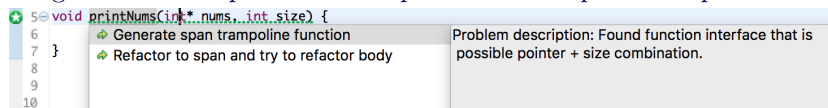
- I.11: Never transfer ownership by a raw pointer (T^*). (See section 2.4.1)
- I.12: Declare a pointer that must not be null as `not_null`. (See section 2.4.2)
- F.22: Use T^* or `owner<T*>` to designate a single object. (See section 2.4.5)
- F.23: Use a `not_null<T>` to indicate that "null" is not a valid value. (See section 2.4.6)

Raw pointer + size parameters to `span<T>` quick fixes

This category includes 2 quick fixes to refactor function interfaces that have raw pointer + size parameters to `span<T>`. One quick fix that generates a trampoline function and another quick fix that tries to refactor the functions body. (See section 3.4.3 for more information)

These quick fixes are available on problem markers generated by the "raw pointer parameter + size to `span<T>`" checker.

Figure 3.4.: Raw pointer + size parameters to `span<T>` quick fixes.



This category of quick fixes can be used to adhere to the following guideline rules:

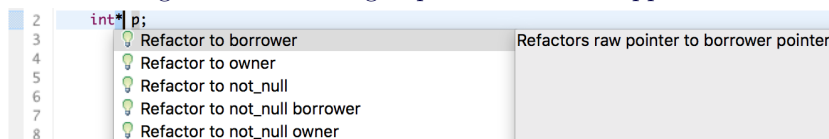
- I.13: Do not pass an array as a single pointer. (See section 2.4.3)
- F.24: Use a `span<T>` or a `span_p<T>` to designate a half-open sequence. (See section 2.4.7)

3.1.3. Quick assists

Quick fixes only work in combination with checkers and problem markers. However, some users might prefer to disable the checkers in order to not have their code riddled with warnings, but still want to use our refactorings. This is why we have decided to also offer refactorings in the form of quick assists in addition to quick fixes.

Quick assists do not require a problem marker to be present in order to be applied. To apply a quick assist the user just needs to select the code he wants to refactor in the editor and then open the quick assist menu (Default Ctrl+1/Cmd+1 on Windows/Mac). Eclipse then checks which quick assists are available for the selected code by using the quick assist processor.

Figure 3.5.: Selecting a quick assist to be applied.

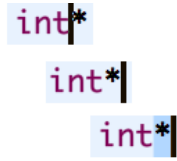


We have implemented quick assists for all the ownership and nullability refactorings outlined in section 3.1.2. Therefore However, we have not implemented quick assists for the `span<T>` refactorings because we ran out of time and had to focus on more important issues.

In order to use the quick assists provided by our plug-in the selection needs to be exactly the pointer (meaning the `*` symbol) that should be refactored. This can be achieved by putting the cursor directly to the left/right of the `*` or by having the selection enclosing just the `*`. We chose this approach because of its simplicity and reduced chance for ambiguity.

3. Implementation

Figure 3.6.: The 3 possible selections to use a quick assist on a raw pointer.



Additionally, the quick assists are only available if the corresponding checker is disabled. For example: The quick assists to refactor raw pointers in return types (to `owner<T>`, `borrower<T>`, etc.) only become available once the checker for raw pointers in return types has been disabled. This was done in order to avoid displaying both the quick fixes and the quick assists at once, because they would be redundant, and also to increase performance by not unnecessarily running the code analysis to determine which quick assists are available.

3.1. Overview of implemented features

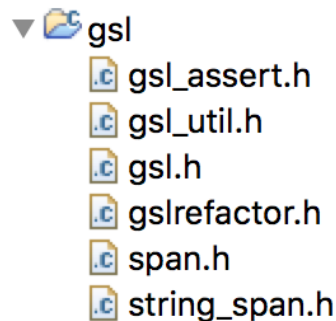
3.1.4. GSL project includer

In order to solve the issues outlined in section 2.5.2 we implemented a feature we call GSL project includer.

Whenever a refactoring (quick fix or quick assist) is applied, the GSL project includer checks if the *gsl* C++ project is present in the current Eclipse workspace. The *gsl* project is a project that contains all of the GSL files plus our own *gslrefactor.h* file.

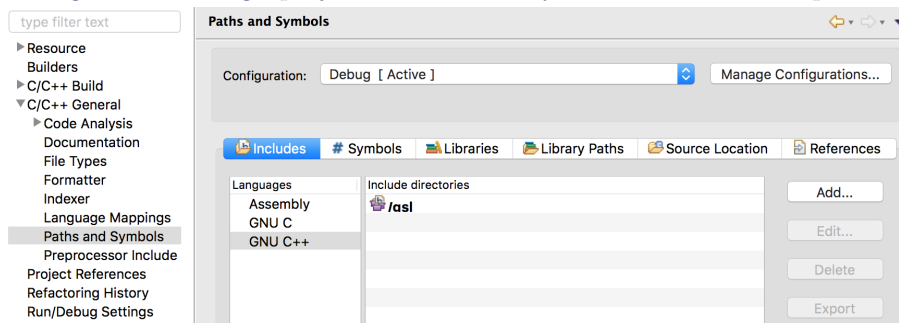
If the *gsl* project is not already present in the workspace, it is automatically generated using the files bundled with our plug-in.

Figure 3.7.: The *gsl* C++ project.



The *gsl* project is then added to the include path of the current C++ project (this is the project the refactoring is applied on) if it is not already there, so the compiler knows where to find it.

Figure 3.8.: The *gsl* project is automatically added to the include path.



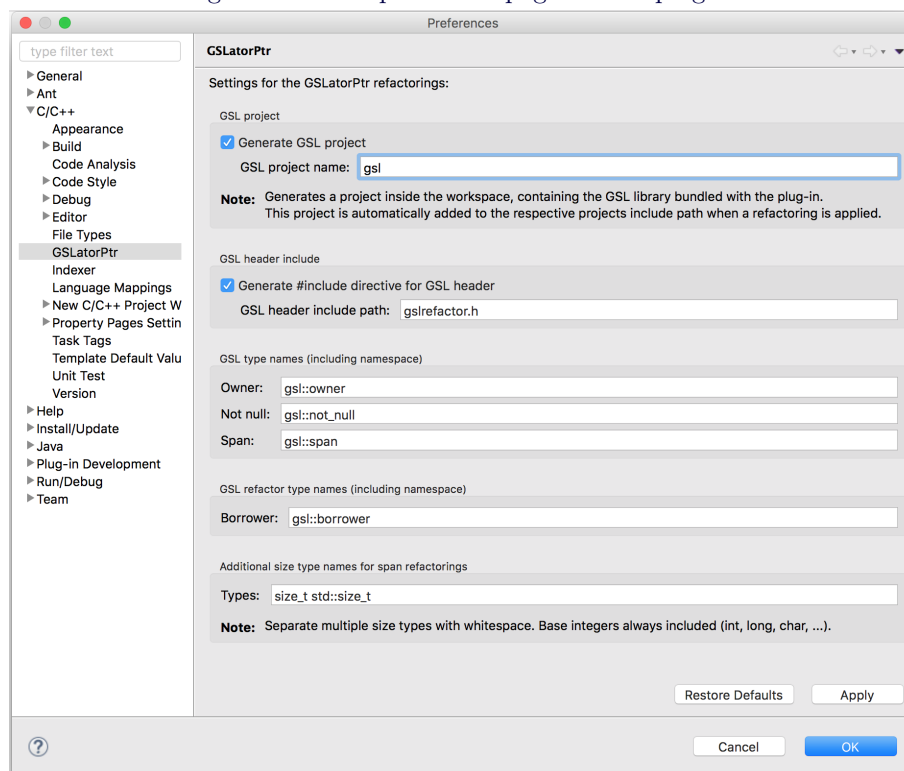
3. Implementation

This entire feature makes it very convenient to use our refactorings, as the user does not need to worry about downloading the GSL and adding it to his projects. If the user still prefers to manually manage the inclusion of the GSL this feature can be disabled on the plug-in preference page. (See section 3.1.5)

3.1.5. Preference page

In section 2.5.4 we concluded that we want to make certain aspects of our plug-in configurable. For this purpose we have created a preference page for our plug-in. The preference page is implemented using Eclipse SWT and is registered at an extension point so that it appears in the Eclipse preferences dialog as "GSLatorPtr" under the "C/C++" category.

Figure 3.9.: The preference page for our plug-in.



3.2. Abstract syntax tree

In the following sections we will briefly explain each of the preferences.

GSL project

Here the user can enable or disable the GSL project includer (See section 3.1.4). It is also possible to change the name of the *gsl* project to something different.

GSL header include

Here the user can chose to enable or disable the automatic generation of the `#include` directive. He can also change the path of included file from `"gslrefactor.h"` to something different.

GSL type names and GSL refactor type names

Here the user can configure the names of the GSL types the plug-in uses when applying refactorings. Owner, not null, span and our own borrower type can be configured. This is useful if the GSL types change in the future or if the user wants to use his own types instead.

Additional size type names for span refactorings

Here the user can specify additional types that should be considered as sizes in the raw pointer + size checker and the associated span refactorings.

3.2. Abstract syntax tree

The AST is a representation of source code that is used to modify, analyze and compile the program. The AST is generated by a parser using the context free grammar of C++. The relationship between the ASTs tree structure and the source code is bidirectional. This means that code can be parsed into a tree, then the tree can be modified and finally translated back to source code. We mostly modify or analyze code in its AST representation, only `#include` statements need to be added on text basis. See section 3.4.1 for an example of how such an AST could look like and for how CDT represents its AST.

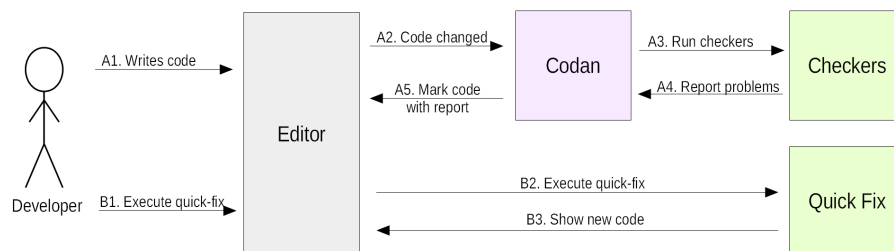
3. Implementation

3.3. Work flow when using checkers and quick fixes

Here we show the most common work flow, but it is also possible to disable all checks, let them run on demand or even just before a compilation.

- (A) Finding a problem
 1. The user writes some code
 2. Editor detects code change and notifies Codan
 3. CDT generates the AST and Codan runs activated checkers
 4. Our code checker traverses the AST and reports found problems
 5. Codan annotates code sections according to the problems with problem markers
 6. The users sees his newly written code underlined with a warning (problem marker)
- (B) Resolving a problem
 1. The user opens a context menu on the faulty code and executes a quick fix to solve the problem
 2. CDT starts our quick fix on the problem marker
 3. Our quick fix changes the AST and tries to solve the problem. Then returns a corrected AST.
 4. The changes are applied to the code and the user can see the results in the editor

Figure 3.10.: Work flow when using checkers and quick fixes



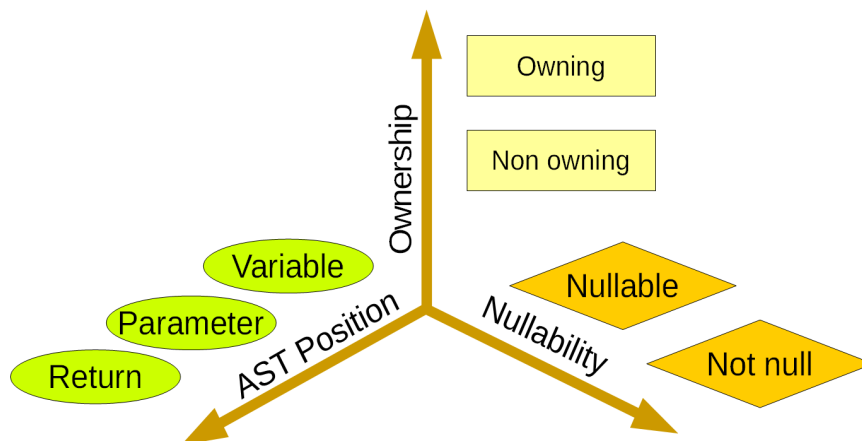
3.4. Our approach and solved challenges

Here we want to show our approach on how we solved certain challenges that came up during implementation.

3.4.1. Ownership and nullability refactorings

The ownership and nullability checkers and refactorings can be divided into 3 different aspects that are independent of each other. Figure 3.11 shows these aspects and the possible values for each. For the implementation we needed to work with the AST of C++ where the context of a pointer is important. A pointer in a parameter type, for example, needs to be refactored differently than a pointer in a function body.

Figure 3.11.: Different aspects that build the ownership refactorings and checks



3. Implementation

AST positons

It is important to understand that although the semantic meaning of a pointer in a declaration within a class body is different to one within a function body, they both can be manipulated the same way since they share the same grammatical possibilities and therefore AST nodes. For an in depth example see the sample code in listing 3.1 and how it translates into the AST seen in figure 3.12. Please note the following:

- The type of a variable consists of:
 - DeclSpecifier:** It holds the type name and eventual template argument and modifiers like `const`, `extern`, `unsigned`
 - Declarator:** It holds the name of the declared element and eventual pointer or reference modifiers, array and function pointer indication.
- The parent nodes of each pair of **DeclSpecifier** and **Declarator** are not always the same.
- The member and the function body variable share the same parent nodes and thus can be modified in the same way. This is also true for global variables (not shown in the figure).

We can do all the refactorings by handling these AST positions:

- Variable type, found within global variables, class bodies and function bodies
- Parameter type
- Return type

AST nodes in CDT

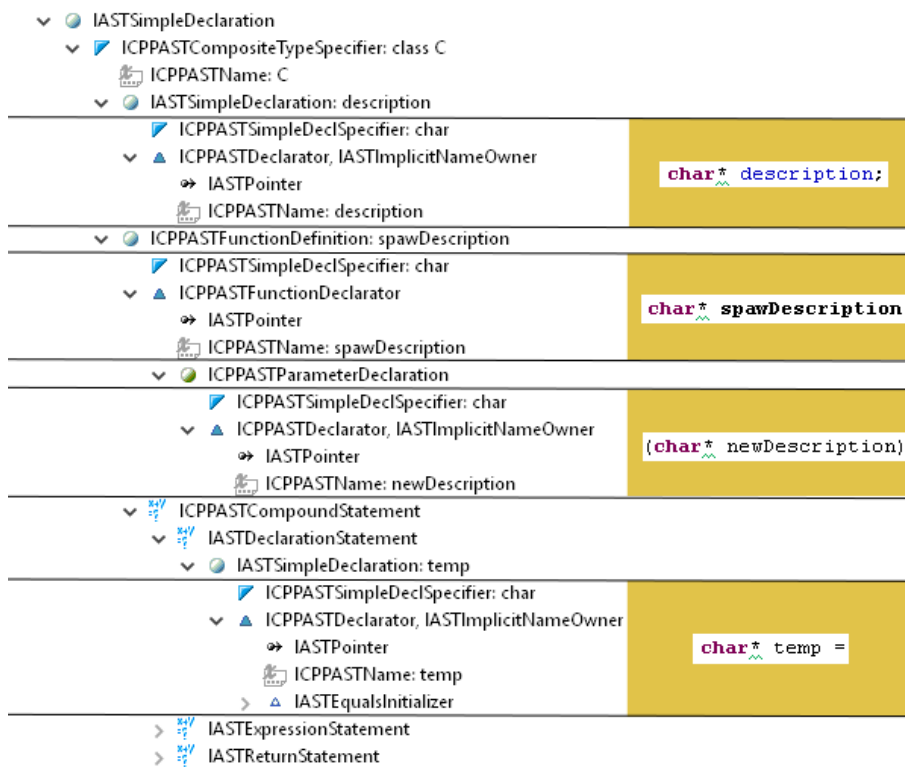
The CDT framework provides a lot of helpful functionality. Most notably the parsing of code files and translation into an AST. Depending on the AST node types found one can, for example, add or get different sub nodes. To find specific nodes in the AST, visitors can be used on it. Not all nodes are visitable but can be found by first searching the underling node via a visitor and then use introspection (reflection) and down-casting to navigate to the wanted node.

3.4. Our approach and solved challenges

Listing 3.1: A class with simple functionality using pointers

```
class C {
  char* description;
  char* spawDescription(char* newDescription) {
    char* temp = description;
    description = newDescription;
    return temp;
  }
};
```

Figure 3.12.: Abstract syntax tree representation of example code in listing 3.1



3. Implementation

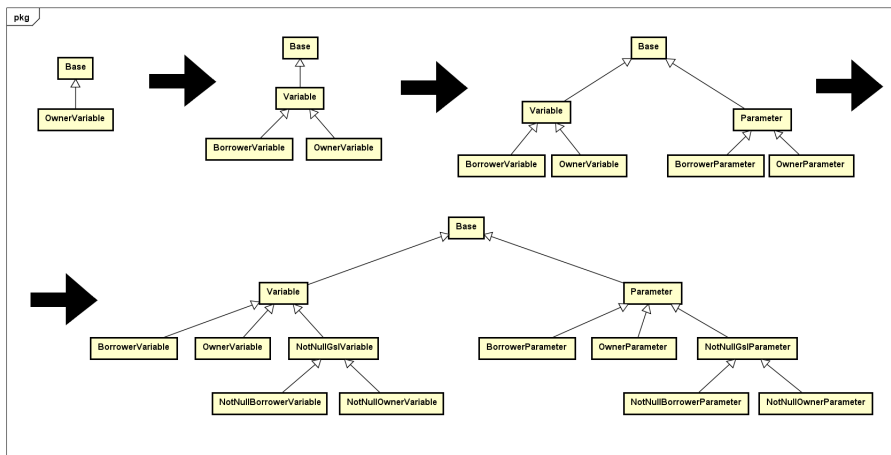
3.4.2. Architecture and the n^3 aspect problem

In this section we want to show how to incremental implementation of new refactorings bloated our architecture and code-base over time and how we fixed this issue with an architectural overhaul.

The creation and improvement of refactorings lends itself well to test driven development (TDD). TDD causes you to be focused on one task and not overbuild the architecture in advance. However during the construction of later quick fixes we noticed that we are not be able to reuse a lot of code and that but because of our architecture we were forced to repeat ourselves a lot.

Each aspect in figure 3.11 needs to be represented in code. We chose to build one refactoring and then add an additional changeable aspect into our system, one at the time. Figure 3.13 shows the the growth of the architecture in a simplified way. First we implemented the a Variable-Nullable-Owning refactoring (Base). Then we made the ownership aspect changeable. Then we made it possible to refactor at the parameter AST position. Then we created refactorings for not-null. Please note that over time, to incorporate one more value of an aspect, we had to make increasingly bigger changes.

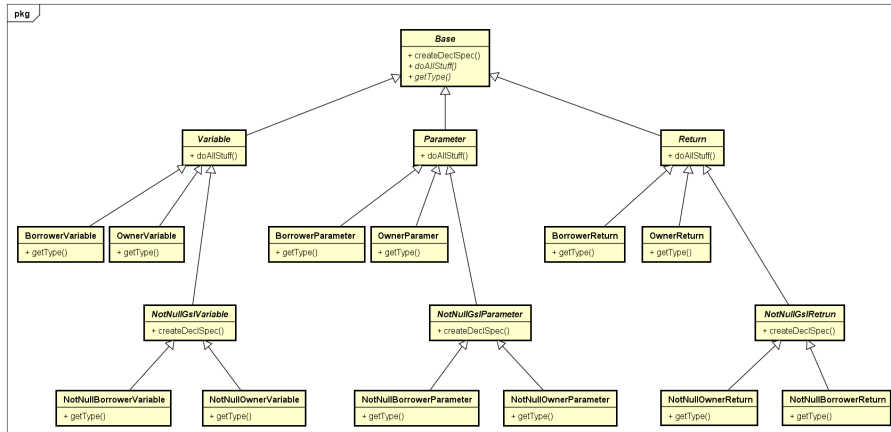
Figure 3.13.: Illustration on how the architecture grew over time



3.4. Our approach and solved challenges

Figure 3.14 shows the final stage of the architecture before we overhauled it. When using inheritance to realize the task we needed $3 * 2 * 2 = 12$ leaf classes, the product of each aspects values. (AST position * Ownership * Nullability).

Figure 3.14.: Resulting bad architecture



This architecture is functional and makes it possible to change behavior very precisely. However, this degree of freedom is not needed and the many classes needs a lot of organization. This is why we decided to refactor our architecture.

3. Implementation

Figure 3.15 shows how the core architecture has changed from a statically configured one to a dynamic one. There are now only 4 leaf classes as children of `GslBaseQuickFix`.

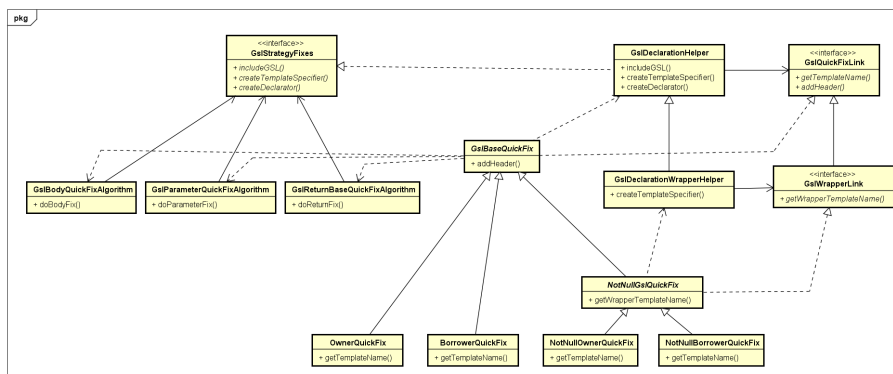
The checkers (not in figure) add information about where a problem was found to an argument list that later can be accessed by `GslBaseQuickFix`, with this information the correct algorithm can be invoked (left).

All algorithms depend on the creation and modification of **DeclSpecifiers** and **Declarators**, as seen in section 3.4.1, for this we use the `DeclarationHelper` (right).

For the not null refactorings we use the `DeclarationWrapperHelper` that can simply wrap a created `TemplateSpecifier` with `not_null<T*>`.

Some parts of the refactorings cause an include file to be added to the current files (`includeGSL()`), since this can not be done by modifying the AST we established a link to the `GslBaseQuickFix` class. The base class can generate includes on a text basis once all changes have been collected and applied.

Figure 3.15.: Better architecture after the overhaul



3.4.3. Span refactorings

This group of refactorings is concerned with the modification function signatures consisting of raw pointer + size parameters to `span<T>`. Checkers that report on this problem also need to save the parameter position where the pointer and length was found.

When a refactoring is executed we search for call sites to this function and change the call to use `span<T>`.

Because some situations need different flavors of delicacy when changing a function interface, we actually created two refactorings to solve this problem:

- Add a trampoline function using `span<T>`
- Change raw pointer + size parameter to `span<T>`

Adding a span-trampoline function

This refactoring is useful when for legacy reasons the original function interface can not be changed but developers want to use `span<T>` to call this function. The trampoline function simply calls the original function with unpacked `span<T>` contents. For an example we execute this refactoring on listing 3.3 and get listing 3.3.

Listing 3.2: Original code before span refactoring

```
int summ(int* p, int length){
    int summ{0};
    for(int i =0; i < length; i++){
        summ += p[i];
    }
    return summ;
}

int main(){
    int* p = new int [5];
    std::cout << summ(p, 5);
    delete[] p;
}
```

3. Implementation

Listing 3.3: Code after span trampoline refactoring

```
#include "gslrefactor.h"
int summ(int* p, int length) {
    int summ { 0 };
    for (int i = 0; i < length; i++) {
        summ += p[i];
    }
    return summ;
}

int summ(gsl::span<int> p) {
    return summ(p.data(), p.size());
}

int main(){
    int* p = new int [5];
    std::cout << summ(gsl::span<int> { p, 5 });
    delete[] p;
}
```

Changing the function

With this refactoring the function interface is changed and the code within the function body is adapted to use the new `span<T>` parameter. We adapt the body by accessing the encapsulated content in `span<T>` wherever one of the old parameters was used. It is important to note that while this approach tries to keep the same functionality it can not be guaranteed. Once the refactoring is done the programmer still needs to adapt the code within the function body to incorporate the full `span<T>` functionality. Listing 3.4 shows the code after this refactoring is applied on the code from listing 3.2.

Listing 3.4: Code after span refactoring with changing the body and interface

```
#include "gslrefactor.h"
int summ(gsl::span<int> p) {
    int summ{0};
    for (int i = 0; i < p.size(); i++) {
        summ += p[i];
    }
    return summ;
}

int main(){
    int* p = new int [5];
    std::cout << summ(gsl::span<int> { p, 5 });
    delete[] p;
}
```

4. Testing

Our plug-in was tested using a mixture of automated testing and manual testing.

4.1. Automated testing

We developed the majority of our plug-in using the test driven development (TDD) approach. Meaning that we would first write an automated test for a new feature that would fail initially and then implement the feature in order to get the test to run successfully. Thus it was important for us to write good automated tests.

CharWars already has a lot of tests for their checkers and quick fixes. We have adopted their style of testing for our project.

The tests are implemented using the CDT Testing [fS15] framework. They are integration tests, because we are not testing individual functions or classes, but rather entire checkers, quick fixes and quick assists.

The CDT Testing framework allows us to have the test cases and their C++ code in a so called *.rts* file, along with additional parameters needed to evaluate the test. When running the tests, the framework loads the code from the *.rts* file and runs our checkers, quick fixes, etc., on it. It then evaluates if the results correlate with what is expected.

In the following sections will briefly explain how the tests work for checkers, quick fixes and quick assists respectively.

4. Testing

4.1.1. Testing checkers

Checker tests are made up of C++ code and a parameter called **markerPositions**. When running the test, the code is checked by the checker and annotated with problem markers. It is then evaluated if the problem markers set by the checker are at the correct positions as specified by the **markerPositions** parameter and have the correct problem id. The problem id is defined per test class, there is a test class for each specific checker.

Listing 4.1: Test case for a checker

```
#!/ReturnPointer
//@.config
markerPositions=4
//@main.cpp
#include "gslrefactor.h"
int* foo(int i);

int* foo(int i) {
    return nullptr;
}

int main() {
    foo(7);
}
```

4.1.2. Testing quick fixes

The quick fix tests expand on the checker tests by applying a quick fix on the problem marker and then comparing the resulting code with the expected code. The "to be refactored" code and the expected code are separated by `//=`. By default the quick fix is applied on the first problem marker that has the problem id defined in the test class, this can be overridden with the **markerNumber** parameter.

Listing 4.2: Test case for a quick fix

```

//!SimpleParameterFix
//@.config
markerNumber=0
//@main.cpp
void foo(int *p);

void foo(int *p){
}

int main() {
}
//=
#include "gslrefactor.h"
void foo(gsl::not_null<gsl::borrower<int*> > p);

void foo(gsl::not_null<gsl::borrower<int*> > p) {
}

int main() {
}

```

4.1.3. Testing quick assists

For the quick assist tests we have tested if quick assists are available in C++ code based on the selection and whether the corresponding checker is enabled or disabled. We do not have automated tests for applying quick assists because that logic is identical to the quick fixes.

Each quick assist test consist of C++ code with a selection. The selection is marked with `/*$*/ ... /*$$*/`. There are parameters to control which checkers should be enabled for the test (e.g. **nakedPointerDeclarationProblemEnabled**) and a parameter called **shouldHaveQuickAssists** to specify if the selection should have quick assists in that case.

Listing 4.3: A test case for the "raw pointer in variable declaration" quick assist

```

//!LocalPointerHasQuickAssistsWhenProblemDisabled
//@.config
nakedPointerDeclarationProblemEnabled=false
nakedPointerParameterProblemEnabled=true
nakedPointerReturnProblemEnabled=true
shouldHaveQuickAssists=true
//@main.cpp
int main(){
    int/*$*//*$$*/ x = new int {42};
}

```

4. Testing

4.2. Manual testing

All of our features were tested manually in an Eclipse CDT instance hooked up with our plug-in. When testing manually we have put extra emphasis on testing the features that are hard to test automated, such as the user interface. When testing we used our own C++ code according to whatever we wanted to test. We also tested our plug-in in a real world scenario, see chapter 5 for that.

We have tested on both the Windows 10 and Mac OS X operating systems and have used Eclipse Mars version 4.5.2 with CDT version 8.8.1.

5. Real world application

Here we try out our plug-in on an open source project and see where it performs and where it falls short. We chose to use the Fish Shell repository from Github [fSh16] (26 of May 2016), as it is a small to mid size open source application with automated tests that does not rely on highly specific build tools. It does however rely on Autotools [mk16] and is therefore not easily buildable directly with Eclipse CDT. We can however use our plug-in to refactor the code and then manually build it and run tests with Autotools. This means that we need to manually include the GSL header files in the source directory since the automatically created C++ project will not be linked in Autotools. Fish Shell does not rely on the C++14 code standard [ISO14] but the GSL library does, so we also had to change the build process to accommodate the newer standard.

5.1. Raw pointer problem statistic

To measure how many problems of a certain type can be found by our plug-in we let the analysis tool run over the whole source code and then see how many warning markers have been produced. Table 5.1 shows the number of locations where a refactoring could be used.

Raw pointers in parameter types	929
Raw pointers in return types	123
Raw pointers in variable types	968
Total	2020

Table 5.1.: Possible situations for owner, borrower and not_null refactoring found

Raw pointer parameter problem

These problems hint at a missing ownership indication in the parameters of a function. These are really important to refactor since here

5. Real world application

misunderstandings could arise when programmers just use the API of the functions without looking at their implementation.

Raw pointer return problem

In the Fish Shell project there are many functions that query a char array for certain properties. Many results of these queries are positions returned as non-owning raw pointers, our plug-in can help the programmer to declare the non owning relationship with `borrower<T*>`. In addition the programmer can clarify if the returned value needs to be null checked or not by refactoring to `not_null<T*>`.

Raw pointer variable declaration problem

These problems show missing indications of ownership in the body of a function, class or in global scope. As seen in the example in listing 5.1, we can not be sure if the description could be null and if it needs to be deleted by just looking at the code.

Listing 5.1: Missing ownership indication in struct found in Fish Shell code

```
/// Struct describing a resource limit.
struct resource_t {
    int resource;           // resource ID
    const wchar_t *desc;  // description of resource
    wchar_t switch_char;  // switch used on commandline to specify
                          // resource
    int multiplier;       // the implicit multiplier used when setting
                          // getting values
};
```

5.2. Raw pointer + size problem statistics

In table 5.2 we show how many raw pointer + size situations were found in the Fish Shell source code by our plug-in. We also show what clear improvement was made by advancing the span refactoring so it is not only applicable to functions with 2 parameters. (See section 5.3 for more information)

Raw pointer + size, just 2 Parameters	57
Raw pointer + size, many Parameters	184
Improvement	127

Table 5.2.: Possible situations for span refactoring found

5.3. Span in Fish Shell

In the Fish Shell project there are many operations made on the user input. These functions often need a pointer to an array and a size as seen in the excerpt of Fish Shell code in listing 5.2. These function signatures then can be changed by our `span<T>` refactoring as seen in listing 5.3.

During our test of refactoring several different code snippets, we discovered that would need span but did not get marked. The biggest reason for this was that our checker and refactoring was limited to only 2 parameters in a functions signature, but many places also needed some additional parameters as seen in the example in listing 5.4. This realization was the reason to go back to the drawing board and try to improve the refactoring by lifting the 2 parameters only constraint.

5. Real world application

Listing 5.2: Raw pointer + size problem found in Fish Shell

```
static wstring str2wcs_internal(const char *in, const size_t in_len)
{
    if (in_len == 0) return wstring();
    ...
}
```

Listing 5.3: Problem resolved using the span refactoring

```
static wstring str2wcs_internal(gsl::span<const char> in) {
    if (in.size() == 0)
        return wstring();
    ...
}
```

Listing 5.4: Pointer + size occurrence not detected by first checker implementation

```
static bool wildcard_has_impl(const wchar_t *str, size_t len,
                              bool internal) {
    assert(str != NULL);
    const wchar_t *end = str + len;
    ...
}
```

6. Conclusion

Finally we want to look back on what we achieved with this bachelor thesis and what compromises we had to take. We also want to describe how our plug-in could be expanded upon in future.

6.1. Features

With this bachelor thesis we were able to realize the following features:

- Refactoring of raw pointers in a variety of different positions into `borrower<T*>` or `owner<T*>` and `not_null<T*>`.
- Refactoring of raw pointer + size function parameters into the more cohesive `span<T>` type.
- Automatically create GSL library project in workspace and link it up with the project where a refactoring is made.
- Give the user the ability to configure and customize his GSL preferences, this will help staying compatible with future changes to the GSL library.

We also tested the plug-in against a popular code base and improved functionality and stability of our refactorings accordingly.

6.2. Relevance of our refactorings

Our refactorings pave the way for the static code analysis as described in the Lifetime I & II paper [SM15]. Many checks can only work on code that correctly uses `not_null<T*>`, `owner<T*>` and `span<T*>`, so this has to be done first.

6. Conclusion

6.3. Early redefining of task

At the start of our project, we set our goal to implement parts of the Lifetime I & II paper [SM15] as a static code analysis tool. We were able to build a simple prototype to find aliasing problems like the one in listing Listing 6.1. However, after corresponding with our supervisor Prof. Peter Sommerlad, who was injured at the start of the project and thus unavailable, we concluded that it would be unfeasibly hard to implement a useful tool before the end of our thesis. As a result, our supervisor gave us the more realistic task of refactoring raw pointers to adhere with the C++ Core Guidelines [SS16].

The time invested in the original task was not wasted though, since we were able to understand the core issues and challenges with raw pointers and their static analysis.

Listing 6.1: Lifetime Safety Checker: Simple pointer aliasing and dangling example

```
int main() {
int *p;
{
int i = 7;
p = &i; // p points to i
} // i goes out of scope, p is now a dangling pointer
*p = 42; // ERROR: Dereferencing dangling pointer
}
```

6.4. Future work

In this thesis we were able to extend the original CharWars plug-in [GS14]. Adding modern features to its catalog, while keeping the original functionality intact. However there are many cases where highly specific types need to be used, we were not able to cover all of them yet. Here are some additional features that could improve the plug-in:

- Improvement of the `span<T>` refactoring so it can handle more diverse function interfaces.
- Quick assist for `span<T>` in addition to the quick fix.
- Suppression of warnings via attributes.
- Checks and refactorings for `string_span`.

A. User manual

A short overview of important plug-in features and how to use them. Our plug-in is built on top of CharWars [GS14], for features of the previous work please see the CharWars manual instead.

This plug-in wants to make it easier to adhere to the C++ Core Guidelines [SS16] by providing refactorings converting raw pointers into GSL [Mic16] types.

A.1. Installation

Or plug-in can be installed via the Eclipse "Install New Software" dialog found in the Help section.

A.2. Configuration

Configuration is done in two ways: Workspace specific (general plug-in preferences) and Project specific (enabling or disabling checkers) .

Workspace settings

These settings, shown in Figure A.1, let you configure how the plug-in maps with the GSL version [Mic16] you use. We included a GSL version (snapshot spring 2016) and the configuration for it. You can find the workspace settings in "Window - Preferences - C++ - GslA-torPtr".

1. **GSL project**

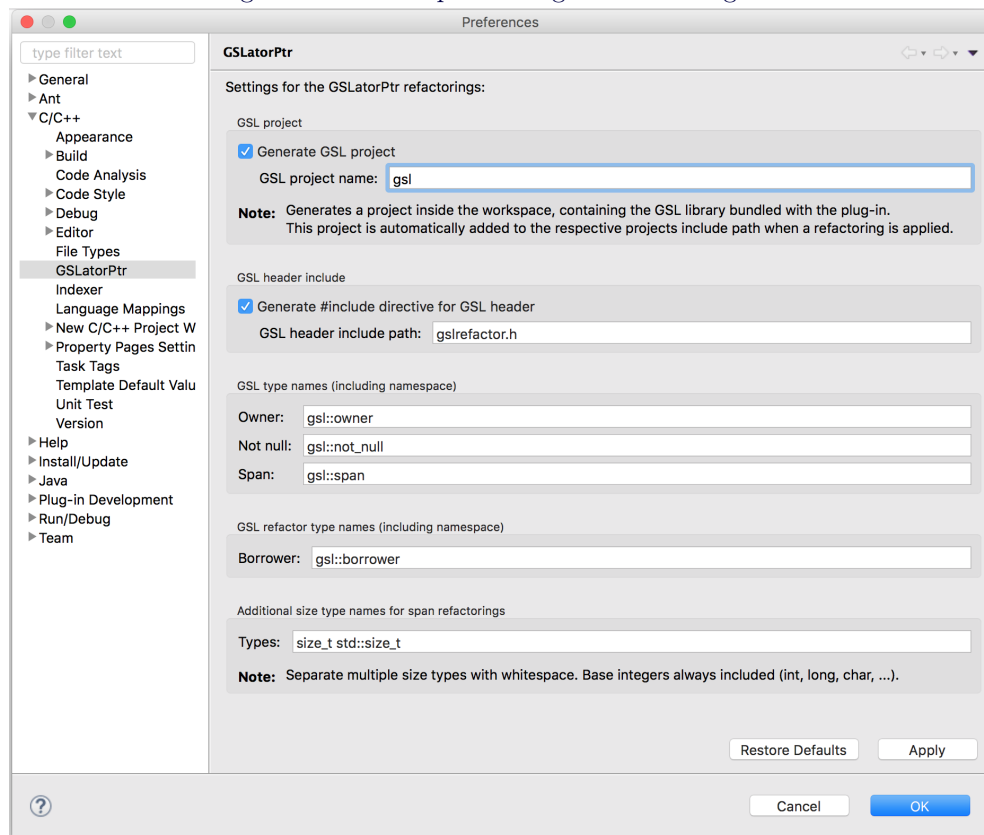
Name of the project that will be linked to the refactored one. If there is no GSL project in the workspace already there will be one generated.

2. **GSL header include**

Decide what GSL file should be included at the top of the refactored file.

A. User manual

Figure A.1.: Workspace configuration settings



3. GSL type names

These names correspond to the types found in GSL.

4. GSL refactor type name

This name corresponds to the type found within the "gslrefactor.h" file.

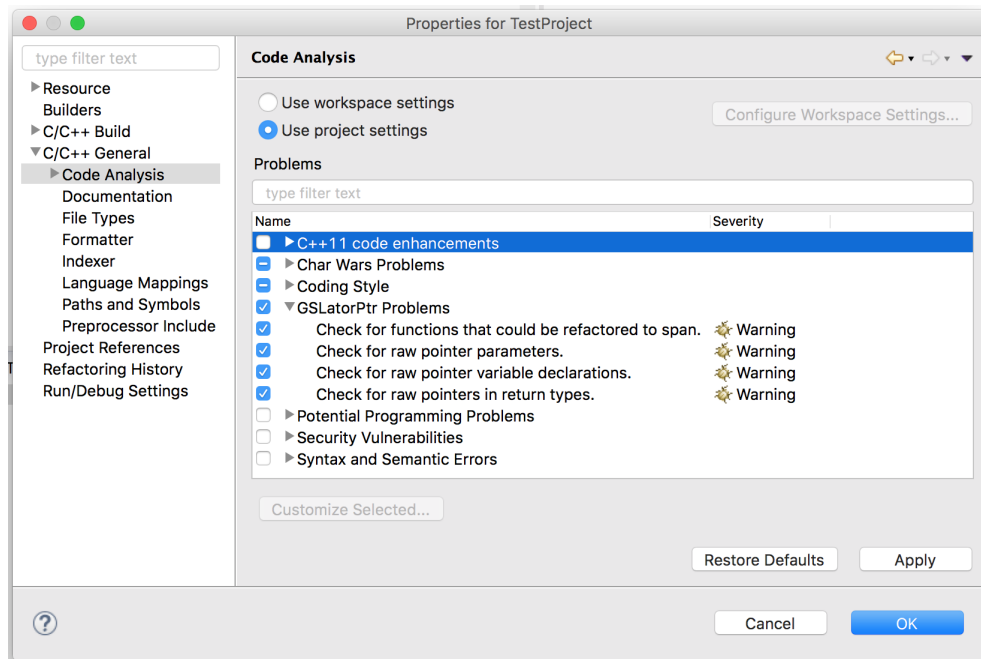
5. Additional size type names

This is a list of names to be considered as a size type for the pointer + size to span refactoring. If you use a specific name representing a size of an array you should put it here. Integers are always included and do not need to be added manually.

Project settings

The project settings are all about what problems should be searched for and how marked. The configuration, as seen in Figure A.2, can be found in "Project - Properties - C/C++ General - Code Analysis".

Figure A.2.: Project code analysis settings



A.3. Refactorings

There are two groups of refactorings we want to feature here. Ones considered with single raw pointers and ones where a array might be passed to a function.

A.3.1. Ownership and nullability refactorings

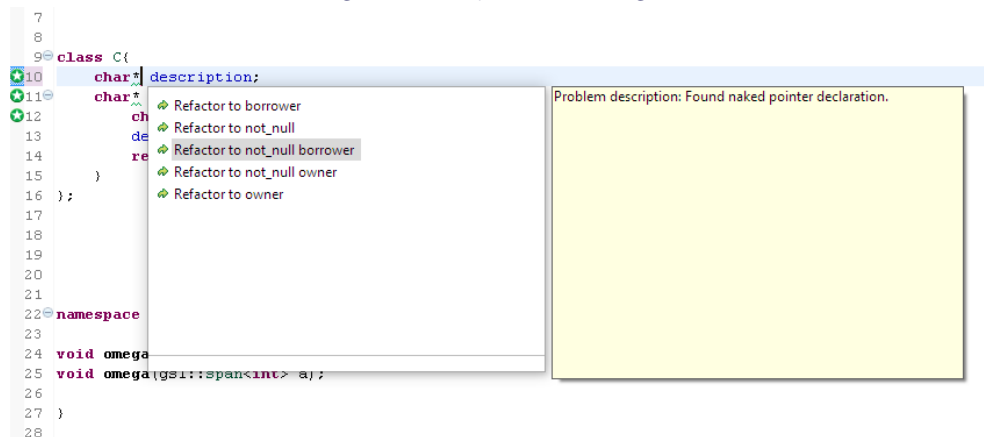
You can use these refactorings to refactor raw pointers into a more expressive type. The refactorings, as seen in Figure A.3, can be executed by moving your cursor over the pointer operator you wish to

A. User manual

clarify and then open the quick assist / quick fix menu (Mac: Cmd+1, Windows: Ctrl+1). You can choose one of the following:

- borrower
Use this to signify that the pointer is not owning and that you have been looking at it so others in your developer team do not need to repeat your work and look at the pointer again and again. Before accessing this pointer there is a null check needed.
- owner
Use this to signify a owning relationship. Before accessing this pointer there is a null check needed.
- not null borrower
Like borrower but no null check is need.
- not null owner
Like owner but no null check is need.
- not null
Use this if you do not want to use borrower.

Figure A.3.: Quick fix dialog



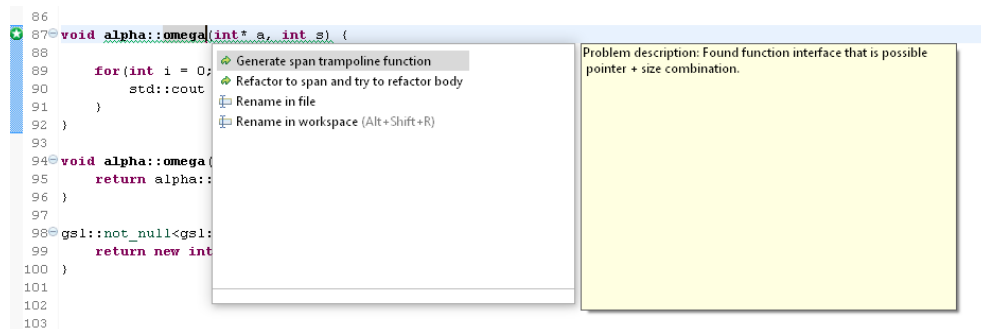
A.3.2. Span refactorings

When a function is passed a pointer and a follow up length, it is possible refactor the two to the GSL `span<T>` type. There are two types of span refactorings available:

- Generate a span trampoline function leaving the old function as is, but adding an additional function with span calling the original. This is useful when you want to keep the original for compatibility reasons.
- Change function signature and trying to adapt the body
This refactoring changes the signature of the function removing the old function. The body of the function will be refactored to use span, however the programmer will still need to change the function body to fully utilize the spans functionality.

Both refactorings search for callers of the function and change them to call the new function with the span type.

Figure A.4.: Span quick fix dialog



B. Developer manual

Here we address future developers of our plug-in who want to expand the functionality or just compile it and use it as reference.

B.1. Version and software used

This configuration is for developing the plug-in on a Windows 10 machine.

1. latex for the documentation
MiKTeX 2.9
2. Eclipse PDE
Mars.1 Release (4.5.1)
3. Maven for testing the build scripts
Apache Maven: 3.3.9
cygwin: 2.873(64 Bit)
4. Source Control
Git 2.6.0.windows.1
SourceTree 1.8.3.0

B.2. How to setup Eclipse PDE compile and run the tests

Here we give an explanation on how we set up the environment, however it is possible to do it differently.

1. Install Eclipse PDE (Plug-in Development Environment)
2. Pull current Cevelp repository
3. Create a workspace on the Cevelp folder, so the subdirectories are all "ch.ifs.cute..."
4. Via Eclipse import dialog, import all the projects found into the workspace
5. Find the "*.taget" file in the cute directory and open it in Eclipse PDE

B. Developer manual

6. Update the repository URLs of the ".target" file if needed
7. Set the ".target" file as target: "Set as Target Platform"
Now the includes that were missing in all the projects should be found.
8. Close all unrelated projects except for CharWars
9. You can now run the CharWars project as an plug-in project, but there will be exceptions thrown in the console.
10. To fix the exceptions you have to deactivate all plug-ins that are not related to your OS. Open the run configuration, then click the plug-ins tab, select "Launch with: plug-ins selected below only", now un tick all plug-ins that generated an exception. You can run again and un tick until no more exceptions occur.
11. If you want to run the CharWars tests open the "ch.hsr.ifs.cute.charwars.test" project and within it open the "ch.hsr.ifs.cute.charwars.test" package. Run the "TestSuiteAll" class as a JUnit plug-In Test.

Build server

If you want to use Jenkins as a build server you need to add some additional plug-ins and software.

- Git plug-in
To get the newest source
- Maven integration plug-in To run the POM files
- Xvnc plug-in
To run JUnit plug-in tests on the server (needs window manager installed on unix server to open eclipse window)

Bibliography

- [Apa04] Apache. Maven - build automation tool, jul 2004. <https://maven.apache.org/> Accessed: 2016-06-15.
- [Atl] Atlassian. Sourcetree- git client. <https://www.sourcetreeapp.com/> Accessed: 2016-06-15.
- [Ban16a] Marius Bancila. Codexpert blog: Guidelines support library review: `span<T>`. <http://codexpert.ro/blog/2016/03/07/guidelines-support-library-review-span/> Accessed: 2016-06-07, 2016.
- [Ban16b] Marius Bancila. Codexpert blog: Guidelines support library review: `string_span<T>`. http://codexpert.ro/blog/2016/03/21/guidelines-support-library-review-string_span/ Accessed: 2016-06-07, 2016.
- [FM13] André Fröhlich and Christian Mollekopf. Smartor, 2013. <https://eprints.hsr.ch/318/1/Smartor%20-%20Dress%20Naked%20C++%20Pointers%20to%20Smart%20Pointers.pdf> Accessed: 2016-06-12.
- [Fou16] The Eclipse Foundation. Eclipse CDT (C/C++ development tooling), 2016. <https://eclipse.org/cdt/> Accessed: 2016-06-07.
- [fS15] HSR Institute for Software. Cdt testing framework, 2015. <https://github.com/IFS-HSR/ch.hsr.ifs.cdttesting> Accessed: 2016-06-15.
- [fS16] HSR Institute for Software. Cvelop IDE, 2016. <https://www.cvelop.com/> Accessed: 2016-06-07.
- [fSh16] fish - the friendly interactive shell, 2016. <https://github.com/fish-shell/fish-shell> Accessed: 2016-06-07.

Bibliography

- [GS13] Fabian Gonzalez and Toni Suter. Pointerminator, 2013. https://eprints.hsr.ch/350/1/Pointerminator_eprints.pdf Accessed: 2016-06-07.
- [GS14] Fabian Gonzalez and Toni Suter. Charwars, 2014. https://eprints.hsr.ch/373/1/CharWars_eprint.pdf Accessed: 2016-06-07.
- [Haa] Krister Haav. toggl. www.toggl.com Accessed: 2016-06-15.
- [HT05] Junio Hamano and Linus Torvalds. Git - version control system, apr 2005. <https://git-scm.com/> Accessed: 2016-06-15.
- [Ipp16] Greg Ippolito. C/c++ memory corruption and memory leaks, 2016. <http://www.yolinux.com/TUTORIALS/C++MemoryCorruptionAndMemoryLeaks.html> Accessed: 2016-06-07.
- [ISO14] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 14882:2014: Information technology – Programming languages – C++*, 4th edition, dec 2014.
- [JKN11] Jenkins - open source continuous integration tool, feb 2011. <https://jenkins.io> Accessed: 2016-06-15.
- [Lav13] Eric Lavesson. C++ ownership semantics, 2013. <http://ericlavesson.blogspot.ch/2013/03/c-ownership-semantics.html> Accessed: 2016-06-08.
- [LD06] J Lang and E Davis. Redmine-open source project management web-application, jun 2006.
- [Mac15] Neil MacIntosh. Static analysis and c++: More than lint - CppCon 2015, sep 2015. CppCon: <http://sched.co/3vaZ> Talk: <https://www.youtube.com/watch?v=rK1HvAw1z50> Accessed: 2016-06-07.
- [Mic16] Microsoft. Gsl: Guidelines support library, 2015-2016. <https://github.com/Microsoft/GSL> Accessed: 2016-06-07.

Bibliography

- [mk15] Cppcon 2015. Standard C++ Foundation, September 2015. <https://cppcon2015.sched.org/> Accessed: 2016-06-08.
- [mk16] Gnu build system, 2016. https://en.wikipedia.org/wiki/GNU_Build_System.
- [SM15] Herb Sutter and Neil MacIntosh. Lifetime safety: Preventing leaks and dangling. <https://github.com/isocpp/CppCoreGuidelines/blob/master/docs/Lifetimes%20I%20and%20II%20-%20v0.9.1.pdf> Accessed: 2016-06-07, December 2015.
- [SS16] Bjarne Stroustrup and Herb Sutter. C++ core guidelines, 2015-2016. <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md> Accessed: 2016-06-07.
- [Sut15] Herb Sutter. Writing good c++ 14 by default - CppCon 2015, sep 2015. CppCon: <http://sched.co/41a3> Talk: <https://www.youtube.com/watch?v=hEx5DNLWGgA> Accessed: 2016-06-07.