

BACHELOR THESIS, SPRING TERM 2014

CharWars

Replace C-String Library calls with C++
std::string Operations



IFS

INSTITUTE FOR
SOFTWARE

AUTHORS

Toni Suter & Fabian Gonzalez

SUPERVISOR

Prof. Peter Sommerlad



HSR

HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Bachelor Thesis

CharWars

Rise of the fallen strings

Fabian Gonzalez, Toni Suter

Spring Term 2014

Supervised by Prof. Peter Sommerlad

Abstract

C strings are still in heavy use in C++ programs. Additionally, standardized C functions such as `strcpy()` and `strstr()` are often used to modify or analyze the content of the strings. Unfortunately, because of the fact that a C string is just a pointer to a zero-terminated character array, those functions have a lot of drawbacks regarding performance, safety and readability.

The `std::string` class from the C++ standard library and its member functions provide a lot of the same functionality without these downsides. Building on previous work from our term project `Pointerminator` we extended the existing Eclipse CDT plug-in so that it helps a programmer to find and automatically refactor pieces of code, that use C strings in an unfavorable way.

We started with an analysis of the various ways C strings and their related C functions are used in practice. Based on that analysis we defined possible refactorings for a subset of the standardized C string functions. We then added this functionality to the existing plug-in, wrote corresponding unit tests and documented its architecture. Finally, we tested the plug-in in the code base of an open source C++ application called `XBMC`. The results of these tests allowed us to optimize the plug-in and to fix some of the problems that we discovered during testing.

Management Summary

This bachelor thesis builds on the results of our term project Pointerminator [Gon13]. The main goal of the term project was to write an Eclipse CDT plug-in that is able to eliminate pointers in existing C++ code. In our bachelor thesis we want to extend the functionality of the Pointerminator plug-in to allow the replacement of C strings and their related C functions (`strcpy()`, `strcat()`, etc.) with `std::string` objects and their member functions.

Motivation

In C, a string is just a pointer to a zero-terminated array of characters. Many existing C++ projects still use C strings along with standard C functions such as `strcpy()` and `strstr()` that are used to manipulate and analyze the string contents. Unfortunately, extensive use of C strings can lead to unreadable, inefficient and unsafe code.

The `std::string` class from the C++ standard library is a modern alternative to C strings. Replacing C strings with `std::string` objects can improve the safety, performance and readability of the code. However, programmers often don't use `std::string` objects either because they don't know about the drawbacks of C strings or because they have to work with an existing code base that already uses C strings.

Goal

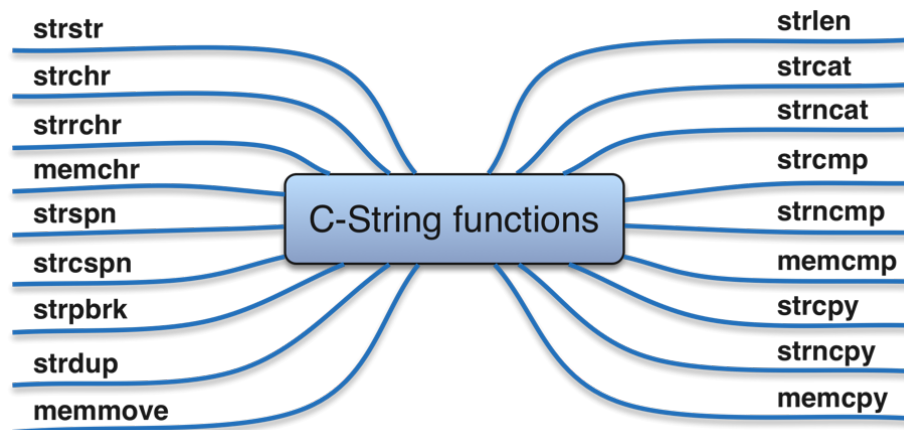
The main goal of this bachelor thesis is to extend the functionality of the Pointerminator plug-in so that C strings and their related C functions can be replaced with `std::string` objects and their member functions. We first analyze the various ways C strings are used in practice and define possible refactorings. It is important that these

refactorings cover all sorts of edge cases so that the tool is reliable enough to be used in an existing C++ code base.

In the implementation phase we add the new functionality to the Pointerterminator plug-in. Finally, the plug-in is tested with an existing C++ project. This helps us to find problems and optimize the refactorings.

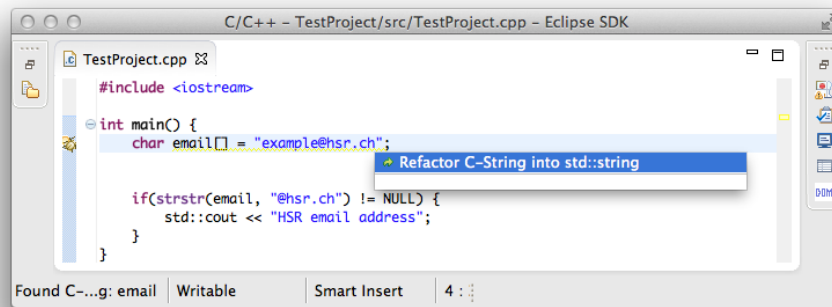
Results

The results of our bachelor thesis can roughly be divided into three parts. First, we analysed the different use cases of C strings and their related C functions. Based on these use cases we decided to put our focus on the C string functions shown in the following picture:



C string functions that can be refactored by the CharWars plug-in

In the second phase we extended the functionality of the Pointerterminator plug-in so that it can replace calls to those C string functions with calls to corresponding `std::string` member functions. The CharWars plug-in analyzes the code that is being written. If it finds a problem, it sets a marker in the editor. The programmer can then trigger an appropriate refactoring through the marker which causes the plug-in to apply this refactoring. The following page shows screen shots of the CharWars plug-in in action:



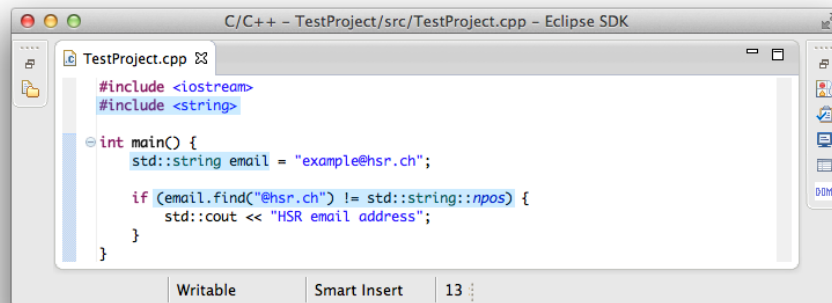
The screenshot shows the Eclipse IDE with a C++ file named TestProject.cpp. The code is as follows:

```
#include <iostream>

int main() {
    char email[] = "example@hsr.ch";

    if(strstr(email, "@hsr.ch") != NULL) {
        std::cout << "HSR email address";
    }
}
```

A tooltip is visible over the `email` variable, suggesting the refactor: "Refactor C-String into std::string". The status bar at the bottom indicates "Found C-...g: email", "Writable", "Smart Insert", and "4 : :".



The screenshot shows the Eclipse IDE with the same C++ file, but now refactored to use `std::string`. The code is as follows:

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

int main() {
    std::string email = "example@hsr.ch";

    if (email.find("@hsr.ch") != std::string::npos) {
        std::cout << "HSR email address";
    }
}
```

The status bar at the bottom indicates "Writable", "Smart Insert", and "13 : :".

Refactoring the C string function strstr()

Finally, to optimize the plug-in, we tested it with an existing open source C++ project called XBMC[xG14]. In total, the CharWars plug-in found 776 C strings and marked them accordingly. To check if the plug-in works correctly, we applied the refactoring for 150 of those C strings and verified the results. The CharWars plug-in was able to correctly refactor 65% of the C strings as shown in the following table:

Markers set	Markers tested	Solved	Unsolved
776	150	98 (65%)	52 (35%)

Further work

The CharWars plug-in is a nice improvement over the existing Point-terminator plug-in but there is still room for improvement. Further optimization would be worthwhile and there are other refactorings that could be added in addition to the existing ones such as:

- Refactoring of strings that are allocated on the heap
- Refactoring of string parameters
- Refactoring of string return values

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

Toni Suter

Place and date

Fabian Gonzalez

Contents

1. Task description	4
1.1. Previous work	4
1.2. Problem	4
1.3. Solution	5
1.4. Our goals	5
1.4.1. Features	5
1.4.2. Additional refactorings	6
1.5. Time management	7
1.6. Final release	7
2. Analysis	8
2.1. The structure of C strings	8
2.1.1. Const string literal	8
2.1.2. Char array on the stack	9
2.1.3. Char buffer on the heap	10
2.2. C strings vs. std::string	11
2.2.1. Memory management	11
2.2.2. Performance	12
2.2.3. Readability	13
2.3. Pointers vs. iterators	13
2.4. Analyzing C string functions	15
2.4.1. strlen	15
2.4.2. strchr / strrchr	16
2.4.3. strstr	19
2.4.4. strcmp	22
2.4.5. strncmp	23
2.4.6. memcmp	24
2.4.7. strpbrk	24
2.4.8. strcspn	25
2.4.9. strspn	26
2.4.10. memchr	27

Contents

2.5. Modifying C string functions	28
2.5.1. strcat / strncat	28
2.5.2. strdup	29
2.5.3. strcpy	30
2.5.4. strncpy	31
2.5.5. memmove	32
2.5.6. memcpy	33
2.6. Converting C string functions	34
2.6.1. atof	34
2.6.2. atoi / atol / atoll	35
2.6.3. strtol / strtoll	36
2.6.4. strtoul / strtoull	37
2.6.5. strtod / strtod / strtold	37
2.6.6. strtoul / strtoull	38
2.7. Refactoring example	39
3. Implementation	46
3.1. Overall architecture and functionality	46
3.1.1. The refactoring cycle	46
3.1.2. Parser and Abstract Syntax Tree (AST)	47
3.1.3. Bindings	49
3.1.4. The index	50
3.1.5. The plug-in components	51
3.1.6. Traversing the AST	51
3.1.7. Modifying and Rewriting the AST	52
3.1.8. Dealing with global variables	53
3.1.9. Two-step transformation	54
3.1.10. Default-Refactoring	59
3.1.11. Extracting common code	61
3.2. Problems and Decisions	62
3.2.1. std::string vs. const std::string	62
3.2.2. std::string member functions vs. algorithm func- tions	63
3.2.3. Multiple rewrites in the same AST subtree	65
3.2.4. Testing	66
3.2.5. Checking if a variable name exists	69
3.2.6. Exception and error handling	70

Contents

3.2.7. Marker position calculation	71
4. Refactoring real-life code	74
4.1. Statistics	74
4.2. Refactoring XBMC	75
4.2.1. First real-life test	76
4.2.2. Second real-life test	83
4.3. Where the plug-in needs manual corrections	83
4.3.1. How to refactor C string definitions	83
4.3.2. How to refactor C string assignments	84
4.3.3. How to refactor C string parameters	84
4.3.4. Known issues	85
5. Conclusion	87
5.1. Achievements	87
5.2. Future Work	88
A. User manual	89
A.1. Installation	89
A.2. Usage and configuration	91
A.2.1. Usage	91
A.2.2. Configuration	93
A.3. De-installation	95

1. Task description

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

1.1. Previous work

This bachelor thesis builds on the results of our term project Pointerterminator [Gon13]. The main goal of that project was to improve the quality of existing C++ code by getting rid of pointers. First, we did an analysis of the various ways pointers can be used in C++. Then we developed an Eclipse CDT plug-in that refactors and replaces pointers automatically. Specifically, the plug-in is capable of doing the following refactorings:

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

1.2. Problem

The Pointerterminator plug-in refactors C-style strings to `std::string` objects. However, it doesn't do much more than that. There are several standard C functions that are commonly used to analyze and modify C strings. For example, the function `strcat()` can be used to append one C string to another. These functions tend to have bad performance. This is because C strings are just pointers to an array of ASCII characters that is terminated with a `'\0'` character and the size of the string isn't stored anywhere. Because of that the size has to be recalculated each time such a function is called. Additionally, these functions have difficult to understand names such as `strpbrk()` and `strchr()` which lead to code that is hard to understand. The Pointerterminator plug-in did not improve that situation. Instead of replacing

1. Task description

the string functions it just tries to make the new `std::string` object work with the existing code.

1.3. Solution

Objects of the class `std::string` store the size of the string in internal state. Therefore, it should be possible to improve the performance and the readability of the code by replacing C string functions with a combination of `std::string` member functions and functions from the standard header `<algorithm>`.

1.4. Our goals

In our bachelor thesis we will first analyze the various C string functions and how they are used in existing C++ code. Then we try to define refactorings for each function that allow us to replace the C string function with a `std::string` member function or a function from the standard header `<algorithm>`. After that we extend the existing Pointerterminator[Gon13] Eclipse CDT plug-in to add the new functionality. The overall goal is to develop a plug-in that can improve the quality of existing C++ code by performing a set of well-defined refactorings. In the end we test the plug-in with a well-known C++ open source project and try to optimize it as much as possible.

1.4.1. Features

The plug-in will replace the following C string functions using a combination of `std::string` member functions and functions from the standard header `<algorithm>`:

Analyzing C string functions

- `strlen()` : Determines the length of a C string.
- `strcmp()` : Compares two C strings.
- `strncmp()` : Compares n characters of two C strings.
- `memcmp()` : Compares two blocks of memory.
- `strstr()` : Searches a substring inside a C string.

1. Task description

- **memchr()** : Searches a byte inside a block of memory.
- **strchr()** : Searches a character inside a C string.
- **strrchr()** : Searches a character inside a C string in reverse order.
- **strpbrk()** : Returns a pointer to the first occurrence of any character from the second C string inside the first C string.
- **strcspn()** : Returns the length of the initial part of the first C string not containing any of the characters that are part of the second C string.
- **strspn()** : Returns the length of the maximum initial segment of the first C string that contains only characters from the second C string.

Modifying C string functions

- **strcat()** : Appends one C string to another.
- **strncat()** : Appends n characters of one C string to another.
- **strcpy()** : Copies a C string into an existing char buffer.
- **strncpy()** : Copies n characters of a C string into an existing char buffer.
- **memcpy()** : Copies one block of memory into another. If the blocks overlap, the behaviour is undefined.
- **memmove()** : Copies one block of memory into another. The blocks may overlap.
- **strdup()** : Allocates a new buffer and copies a C string into that buffer.

1.4.2. Additional refactorings

If there is enough time at the end of the project the plug-in will also include the following refactorings:

- **atof()** : Converts a C string into a double.
- **atoi()** : Converts a C string into an int.
- **atol()** : Converts a C string into a long.
- **atoll()** : Converts a C string into a long long.
- **strtol()** : Converts a byte string into a long.
- **strtoll()** : Converts a byte string into a long long.

1. Task description

- **strtoul()** : Converts a byte string into an unsigned long.
- **strtoull()** : Converts a byte string into an unsigned long long .
- **strtof()** : Converts a byte string into a float.
- **strtod()** : Converts a byte string into a double.
- **strtold()** : Converts a byte string into a long double.
- **strtoimax()** : Converts a byte string into std::intmax_t.
- **strtoumax()** : Converts a byte string into std::uintmax_t.

1.5. Time management

Our project started on the 17th of February, 2014. It will end on June the 13th, 2014 at 12.00 p.m. which is when the final release has to be submitted completely.

1.6. Final release

The following items will be included in the final release of the project:

- 4 printed exemplars of the documentation (1 colored)
- Poster for presentation
- Management Summary and Abstract
- 2 CD/DVD with update site that contains the plug-in, project resources, documentation, virtual machine with operational Eclipse CDT with plug-in installed
- 1 CD for archive with the documentation and abstract without personal informations

2. Analysis

This chapter contains an analysis of C strings and shows their drawbacks in comparison to `std::string` objects. It also contains a description of several standard functions that are often used to analyze or manipulate C strings and demonstrates different refactorings that could be applied by the plug-in.

2.1. The structure of C strings

In C, a string is just a pointer to an array of characters that is terminated by a `'\0'` character. No additional information about the length of the string is stored anywhere. There are several ways to create a C string which have different effects on the mutability and the memory location of the string:

2.1.1. Const string literal

One way to create a C string is to initialize a char pointer with the address of a string literal as shown in Listing 2.1:

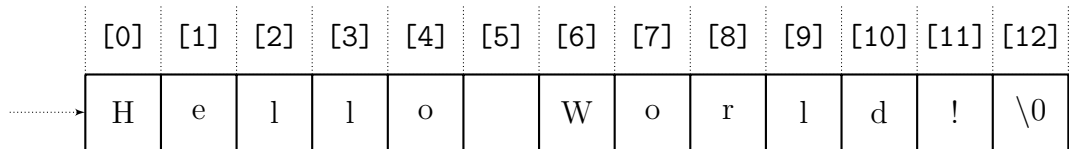
Listing 2.1: Const string literal

```
int main() {
    const char *str = "Hello, World!";
    //do something with str
}
```

By default the GCC compiler allocates 14 bytes (13 ASCII characters + one `'\0'` character) in the global/static section of the memory. This is shown in Figure 2.1:

2. Analysis

Figure 2.1.: Structure of a C string



In addition, the string is read-only. This allows the compiler to do an optimization called string pooling. Listing 2.2 shows an example:

Listing 2.2: String pooling

```
const char *str1 = "Hello, World!";

int main() {
    const char *str2 = "Hello, World!";
    std::cout << std::boolalpha
                << (str1 == str2)
                << std::endl;
}
```

The above program outputs “true”. Because the strings are immutable and stored in global/static memory, the compiler can optimize by storing strings that have the same value only once. All char pointers that are initialized with the same string literal then point to the same location in memory.

However, GCC does have an option `-fwritable-strings` to disable string pooling. This option also makes the strings mutable.

2.1.2. Char array on the stack

To create a mutable C string the programmer can declare a char array and initialize it with a string literal as shown in Listing 2.3:

Listing 2.3: Char array on the stack

```
int main() {
    char str[] = "Hello, World!";
    //do something with str
}
```

This string has the same representation as shown in Figure 2.1. However, the string is mutable and stored on the stack. Therefore, the

2. Analysis

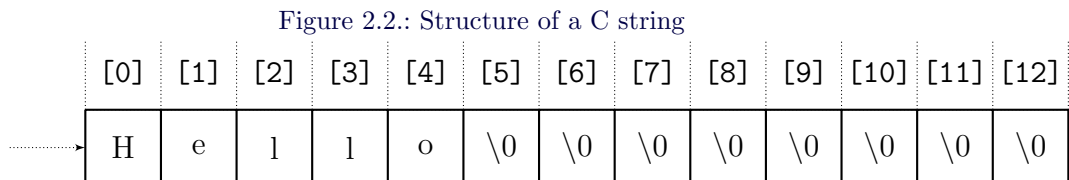
allocated memory automatically gets freed at the end of the array's scope.

Char arrays can also be partially initialized, leaving room to append another string to the first one as shown in Listing 2.4:

Listing 2.4: Char array on the stack

```
int main() {
    char str[13] = "Hello";
    strcat(str, ", World!");
    //do something with str
}
```

Before the call to the function `strcat()`, the array buffer looks like this:



After the concatenation it again looks like Figure 2.1.

2.1.3. Char buffer on the heap

Sometimes the size of a string is not known at compile time. Such strings can be dynamically allocated on the heap using `malloc()` as shown in Listing 2.5:

Listing 2.5: String allocation on the heap

```
char *duplicateString(const char *str) {
    char *copy = (char *)malloc(strlen(str)+1);
    strcpy(copy, str);
    return copy;
}

int main() {
    char *str = duplicateString("A string");
    //do something with str
    free(str);
}
```

2. Analysis

In this case clients of the function `duplicateString()` have to free the resulting string after they are done with it because strings that are allocated with `malloc()` aren't freed automatically.

2.2. C strings vs. `std::string`

2.2.1. Memory management

If a programmer wants to concatenate two C strings, he or she has to make sure, that there is enough space reserved in the destination buffer to hold the contents of both strings as well as the terminating `'\0'` character. If the sizes of the strings are known at compile time, this can be done by defining a char array on the stack as shown in Listing 2.6.

Listing 2.6: Concatenation of two C strings

```
int main() {
    const char *str1 = "Hello, ";
    const char *str2 = "world!";
    char str3[14];

    strcpy(str3, str1);
    strcat(str3, str2);

    //do something with str3
}
```

However, often the sizes are unknown at compile time. In the book *The C++ Programming Language* by Bjarne Stroustrup[Str97], there is a good example that shows how much code can be involved to achieve a relatively simple thing. The example is shown in Listing 2.7:

2. Analysis

Listing 2.7: Before the refactoring

```
char *address(const char *iden, const char *dom) {
    int iden_len = strlen(iden);
    int dom_len = strlen(dom);
    char *addr = (char *)malloc(iden_len+dom_len+2);
    strcpy(addr, iden);
    addr[iden_len] = '@';
    strcpy(addr+iden_len+1, dom);
    return addr;
}

int main() {
    char *email = address("someone", "gmail.com");
    //do something with email
    free(email);
}
```

The function `address()` returns a new C string that contains the email address built from the identifier and the domain part. If the programmer uses `std::strings` instead, the code becomes much more elegant and readable. This is shown in Listing 2.8:

Listing 2.8: After the refactoring

```
std::string address(const std::string& iden, const std::string& dom) {
    return iden + '@' + dom;
}

int main() {
    std::string email = address("someone", "gmail.com");
    //do something with email
}
```

The class `std::string` takes care of memory management and releases the memory once the variable “email” goes out of scope. Therefore, the call to the function `free()` is not necessary anymore.

2.2.2. Performance

As shown in section 2.1, C strings have a compact structure and take up very little space. While this can be an advantage in computing environments where memory is scarce (e.g., in embedded systems), it also comes with a performance penalty. String functions like `strlen()` or `strcat()` have to find out the length of the string to perform their task. This is shown in a blog post by Joel Spolsky[Spo14] in which he

2. Analysis

shows how `strcat()`, the function which appends one string to another, may be implemented:

Listing 2.9: Example from Joel on Software - Back to Basics

```
void strcat(char* dest, char* src)
{
    while(*dest) dest++;
    while(*dest++ = *src++);
}
```

It is easy to see that this code has $O(n)$ complexity and therefore isn't very efficient. Since the length isn't stored anywhere and there is no information about the buffer size, the function has to walk through the string looking for its null-terminator every time it is called. Sometimes compilers may be able to optimize performance for literals at compile time, but often this is not possible (e.g., if a string is read from `std::cin`).

The `std::string` class has a member function `size()` that has constant complexity according to the C++11 standard, indicating that the size of the string is stored in internal state.

2.2.3. Readability

The examples in the subsection 2.2.1 Memory management show how much the readability can be improved under certain circumstances. This not only makes the code easier to read but also lowers the risk for a programmer to introduce bugs when he or she has to modify the code.

2.3. Pointers vs. iterators

C strings are often used along with functions, that can be used to analyze or modify the string's contents. Some of those functions return a `char *` pointer that points to a position inside the string. For example, the function `strstr()` takes two C strings and returns a pointer to the first occurrence of the second string inside the first string. Listing 2.10 shows an example:

2. Analysis

Listing 2.10: C string function that returns a pointer

```
int main() {
    char url[100];
    std::cin >> url;
    char *found = strstr(url, ".ch");

    if(found) {
        *(found + 1) = 'd';
        *(found + 2) = 'e';
    }
    std::cout << url << std::endl;
}
```

Once the C string has been refactored to a `std::string`, the function `strstr()` also needs to be replaced by some other means. One way is to use one of `std::string`'s member functions as shown in Listing 2.11:

Listing 2.11: Example with `std::string` member function

```
int main() {
    std::string url;
    std::cin >> url;

    std::size_t found = url.find(".ch");
    if(found != std::string::npos) {
        url[found + 1] = 'd';
        url[found + 2] = 'e';
    }
    std::cout << url << std::endl;
}
```

Sometimes it is better to use one of the functions from the standard header `<algorithm>` because they often return an iterator which is conceptually similar to a pointer. Listing 2.12 shows an example using the `search()` function:

Listing 2.12: Example with function from standard header `<algorithm>`

```
int main() {
    std::string url;
    std::cin >> url;

    std::string searchStr = ".ch";
    auto found = std::search(url.begin(), url.end(), searchStr.begin(),
                             searchStr.end());
    if(found != url.end()) {
        *(found + 1) = 'd';
        *(found + 2) = 'e';
    }
    std::cout << url << std::endl;
}
```

2. Analysis

Whether it is better to use a `std::string` member function or a function from the standard header `<algorithm>` depends on what the `char *` pointer is used for in the original code.

2.4. Analyzing C string functions

This section contains the analysis of different C string functions. Most of the analyzed refactorings can also be used to refactor `wchar_t` strings.

2.4.1. `strlen`

The function `strlen()` has the following signature:

Listing 2.13: Signature of function `strlen()`

```
size_t strlen ( const char * str );
```

This function returns the length of a C string. The length is calculated from the beginning of the string to the null character, without including it. All C strings are terminated with a null character.

The class `std::string` has a member function called `size()` that also calculates the length. The signature of this member function can be found in Listing 2.14:

Listing 2.14: Signature of member function `size()`

```
std::string::size_type size() const;
```

Most of the time `size_type` is the same as `size_t`, so the two functions are very similar.

The following example shows how a simple use of the `strlen()` function could be replaced.

Listing 2.15: Before refactoring

```
int main() {
    char s[] = "Hello!";
    size_t l = strlen(s);
    std::cout << l;
}
```

Listing 2.16: After refactoring

```
int main() {
    std::string s = "Hello!";
    size_t l = s.size();
    std::cout << l;
}
```

2. Analysis

2.4.2. strchr / strrchr

The functions `strchr()` and `strrchr()` have the following signatures:

Listing 2.17: Signatures of the functions `strchr()` and `strrchr()`

```
const char * strchr(const char *str, int character);
char * strchr(char *str, int character);
const char * strrchr(const char *str, int character);
char * strrchr(char *str, int character);
```

They return a pointer to the first (`strchr`) or last (`strrchr`) occurrence of a given character in the C string “str”. If the character could not be found in this string both functions return a null pointer.

The functions can be replaced with the member functions `find_first_of()` and `find_last_of()` of the `std::string` class. Both functions are overloaded several times. Listing 2.33 shows the versions that best match the signatures of the `strchr` and `strrchr` function.

Listing 2.18: Signatures of the member functions `find_first_of()` and `find_last_of()`

```
size_type find_first_of(CharT ch, size_type pos = 0) const;
size_type find_last_of(CharT ch, size_type pos = npos) const;
```

These `std::string` member functions have a different return type. Instead of a pointer they return an index (of type `size_type`) that denotes the position of the character.

A simple way would be to convert the index back to a pointer and leave the rest of the program unchanged. An example can be found in the listing below.

Listing 2.19: Before the refactoring

```
int main() {
    char s[] = "Hello";
    const char *p =
        strchr(s, 'l');
}
```

Listing 2.20: After the refactoring

```
int main() {
    std::string s = "Hello";
    const char *p = s.c_str()
        + s.find_first_of('l');
}
```

By calling the member function `c_str()` a `const` pointer to the first char of the string is returned. By adding the index to the pointer it points to the correct position of the character. However, this refactoring doesn't take into account that it may be possible that the character is not part of the string in which case this calculation would be wrong.

2. Analysis

Instead of using a `std::string` member function it is also possible to use the `std::find` function of the standard header `<algorithm>` to find the first or last position of the located character. This function uses iterators as input and returns an iterator. The following listing shows its signature:

Listing 2.21: Signature of member function `std::find()`

```
InputIt find(InputIt first, InputIt last, const T& value);
```

Using this function we benefit from the iterator return type that allows us to do a simpler conversion to a pointer. An example can be found in the listings below.

Listing 2.22: Before the refactoring

```
int main() {
    char s[] = "World";
    char *ptr =
        strchr(s, 'o');
    *ptr = 'A';
    std::cout << ptr;
}
```

Listing 2.23: After the refactoring

```
int main() {
    std::string s = "World";
    auto ptr = std::find
        (s.begin(), s.end(), 'o');
    *ptr = 'A';
    std::cout << &*ptr;
}
```

The reverse iterators “`rbegin()`” and “`rend()`” can be used instead of the normal iterators to get the same behavior as the “`strchr`” function.

There would be more benefit if the plug-in refactors also the resulting char pointer. This could be difficult because pointers can be used in a lot of different ways.

Task 1: Handling Null-Values

If a programmer uses the `strchr()` or `strrchr()` function to find out whether a character is inside a string or not, he or she will check if the result is a null pointer or not. The corresponding `std::string` member function returns `std::string::npos` if the given character was not found in the string. So the plug-in should scan the code for corresponding null-checks and change them. For more details see the listings below.

2. Analysis

Listing 2.24: Before the refactoring

```
int main() {
    char s[] = "@mail";
    if (strchr(s, '@'))
    {
        //contains @ sign
    }
}
```

Listing 2.25: After the refactoring

```
int main() {
    std::string s = "@mail";
    if (s.find_first_of('@') !=
        std::string::npos){
        //contains @ sign
    }
}
```

This refactoring can also be done with the `std::find` function. This function returns an iterator to the end of the string if the character is not found:

Listing 2.26: Before the refactoring

```
int main() {
    char s[] = "@mail";
    if (strchr(s, '@')) {

        //contains @ sign
    }
}
```

Listing 2.27: After the refactoring

```
int main() {
    std::string s = "@mail";
    if (std::find(s.begin(), s.end(),
        '@') != s.end()) {
        //contains @ sign
    }
}
```

If the pointer is passed to a function or in other special cases where the pointer can not be replaced the plug-in should still be able to produce a valid pointer. The first example shows how this is done with the “`find_first_of`” member function of the class `std::string`:

Listing 2.28: Before the refactoring

```
int main() {
    char s[] = "@mail";

    const char *p = strchr(s, '@');

    print(p);
}
```

Listing 2.29: After the refactoring

```
int main() {
    std::string s = "@mail";
    size_t pos =s.find_first_of('@');
    const char *p = pos !=
        std::string::npos ? s.c_str() +
        pos : nullptr;
    print(p);
}
```

The following example uses the “`std::find`” function to refactor the same code.

2. Analysis

Listing 2.30: Before the refactoring

```
int main() {
    char s[] = "@mail";

    const char *p = strchr(s, '@');

    print(p);
}
```

Listing 2.31: After the refactoring

```
int main() {
    std::string s = "@mail";
    auto pos = std::find(s.begin(),
        s.end(), '@');
    const char *p = pos != s.end() ?
        &*pos : nullptr;
    print(p);
}
```

2.4.3. strstr

The function `strstr()` has the following signature:

Listing 2.32: Signature of function `strstr()`

```
const char* strstr(const char* str1, const char* str2);
```

It returns a pointer to the first occurrence of the substring `str2` in the string `str1`. If `str2` is not a substring of `str1`, the function returns a null pointer.

The class `std::string` has several overloads of a member function called `find()` that does a similar thing. The signature of the overload that is the closest match to `strstr()` is shown in Listing 2.33:

Listing 2.33: Signature of member function `find()`

```
size_type find(const CharT* s, size_type pos = 0) const;
```

The main difference between the two functions is the type of the return value. While `strstr()` returns a pointer, `find()` returns the index of the substring within `str1`.

A conservative way of dealing with this problem would be to immediately convert the index back to a pointer and leave the rest of the program unchanged. Listing 2.34 and Listing 2.35 show an example:

2. Analysis

Listing 2.34: Before the refactoring

```
int main() {
    char s[100];
    std::cin >> s;

    const char *p =
        strstr(s, "@");
    //do something with p
}
```

Listing 2.35: After the refactoring

```
int main() {
    std::string s;
    std::cin >> s;

    const char *p =
        s.c_str() + s.find("@");
    //do something with p
}
```

The index can be converted back to a pointer by adding it to the char pointer returned by the member function `c_str()`. However, because the pointer returned by `c_str()` is `const`, this only works if the pointer is not used to modify the contents of the string.

Ideally, the plug-in would refactor not only the call to `strstr()` but also the resulting char pointer and the subsequent code that uses this pointer. This can be difficult because pointers can be used to do a lot of different things. Often, it is easier to use a function from the standard header `<algorithm>` that returns an iterator as described in section 2.3 Pointers vs. iterators.

In the context of the `strstr()` function the pointer is often used to perform one or more of the following tasks:

Task 1: Performing a Null-Check

Often the programmer uses the `strstr()` function to find out whether `str2` is a substring of `str1`. The exact value of the pointer is of no interest. All the code does, is to check whether it is null or not. Listing 2.36 shows an example:

Listing 2.36: Before the refactoring

```
int main() {
    char url[100];
    std::cin >> url;

    if(strstr(url, ".com")) {
        //url is a .com
    }
}
```

Listing 2.37: After the refactoring

```
int main() {
    std::string url;
    std::cin >> url;

    if(url.find(".com")
        != std::string::npos) {
        //url is a .com
    }
}
```

2. Analysis

The same thing can be achieved using the `find()` member function but because it returns an index and not a pointer, the return value has to be compared with the constant `std::string::npos` instead of `null`.

Task 2: Calculating the index

Sometimes the programmer is interested in the index of substring `str2` inside of `str1`. This value can be calculated by doing pointer arithmetic as shown in Listing 2.38:

Listing 2.38: Before the refactoring

```
int main() {
    char email[100];
    std::cin >> email;
    int prefix_len = strstr(email,
        "@gmail.com") - email;
    //do something with prefix_len
}
```

Listing 2.39: After the refactoring

```
int main() {
    std::string email;
    std::cin >> email;
    int prefix_length = email.find(
        "@gmail.com");
    //do something with prefix_len
}
```

The `find()` member function returns the index directly, so that there is no need to calculate it.

Task 3: Manipulating the string

If `str1` is not `const`, it is possible to modify it through the pointer returned by the function `strstr()`:

Listing 2.40: Before the refactoring

```
int main() {
    char url[100];
    std::cin >> url;

    char *tld_ptr =
        strstr(url, ".de");

    *(tld_ptr + 1) = 'c';
    *(tld_ptr + 2) = 'h';
    //do something with url
}
```

Listing 2.41: After the refactoring

```
int main() {
    std::string url;
    std::cin >> url;
    std::string s = ".de";
    auto tld_ptr = std::search(
        url.begin(), url.end(),
        s.begin(), s.end());
    *(tld_ptr + 1) = 'c';
    *(tld_ptr + 2) = 'h';
    //do something with url
}
```

Listing 2.41 shows how the same thing can be achieved using the `search()` function from the standard header `<algorithm>`. This function returns an iterator which can be used in the same way as the pointer.

2. Analysis

The subsequent code didn't have to be changed, because iterators can be used just like pointers to modify the contents of a string. However, an additional variable to hold the value of the search string had to be introduced.

Task 4: Passing the pointer to a function

Listing 2.42 shows how the pointer could also be passed to a function:

Listing 2.42: Before the refactoring

```
int main() {
    char email[100];
    std::cin >> email;
    char *domain_part = strstr(email,
        "@") + 1;

    //print domain part of email
    address
    print(domain_part);
}
```

Listing 2.43: After the refactoring

```
int main() {
    std::string email;
    std::cin >> email;
    auto const found=email.find("@");
    std::string domain_part =
        email.c_str() + found + 1;
    //print domain part of email
    address
    print(domain_part.c_str());
}
```

With a call to the member function `c_str()`, a `std::string` can be converted back to a C string. However, this C string is `const` and cannot be modified.

2.4.4. strcmp

The C string member function `strcmp()` has the following signature:

Listing 2.44: Signature of function `strcmp()`

```
int strcmp(const char *str1, const char *str2);
```

The function compares the strings “str1” and “str2”. If both strings are equal the return value is zero. If the return value is greater than zero it indicates that the first C string is alphabetically after the second string, otherwise the return value is lower than zero.

This function can be replaced with the `compare()` member function of the `std::string` class. The function signature that best matches can be found below.

Listing 2.45: Signature of member function `compare()`

```
int compare(const CharT* s) const;
```

2. Analysis

See an example of this refactoring in the code below.

Listing 2.46: Before the refactoring

```
int main() {
    char a[] = "Apple";
    char b[] = "Banana";
    std::cout << strcmp(a,b);
}
```

Listing 2.47: After the refactoring

```
int main() {
    std::string a = "Apple";
    char b[] = "Banana";
    std::cout << a.compare(b);
}
```

2.4.5. strncmp

The function `strncmp()` has the following signature:

Listing 2.48: Signature of function `strncmp()`

```
int strncmp(const char *str1, const char *str2, size_t num);
```

The function compares the first “num” characters of the strings “str1” and “str2”. If the compared characters are equal the return value is zero. Otherwise is the return value greater or lower than zero depending on the alphabetical order of the strings.

This function can also be replaced with the `compare()` member function of the `std::string` class. This function has a signature that takes arguments to define the characters that should be compared. The function signature can be found below:

Listing 2.49: Signature of member function `compare()`

```
int compare(size_type pos1, size_type count1, const basic_string& str,
            size_type pos2, size_type count2) const;
```

Both functions have the same return values so we just need to change the function call. The parameters “pos1” and “pos2” are always zero in this case. So the comparison starts from the beginning of the strings. An example is shown in the listings below:

Listing 2.50: Before the refactoring

```
int main() {
    char a[] = "google.co";
    char b[] = "google.ch";
    std::cout <<
        strncmp(a,b,6);
}
```

Listing 2.51: After the refactoring

```
int main() {
    std::string a = "google.co";
    char b[] = "google.ch";
    std::cout <<
        a.compare(0,6,b,0,6);
}
```

2. Analysis

2.4.6. memcmp

The function `memcmp()` has the following signature:

Listing 2.52: Signature of function `memcmp()`

```
int memcmp(const void *ptr1, const void *ptr2, size_t num);
```

The `memcmp()` member function compares the first “num” bytes of memory blocks of the two pointers. The function will return a zero if both blocks are identical. Otherwise it returns a greater or lower value than zero depending on the lexicographical order of the first value.

The `compare()` member function of the `std::string` class has the same behaviour. The function signature of Listing 2.49 can be used for this refactoring.

Because both functions have the same return value the refactoring just need to change the function call. An example can be found in the listings below.

Listing 2.53: Before the refactoring

```
int main() {
    char a[] = "google.co";
    char b[] = "google.ch";
    std::cout <<
        memcmp(a,b,6);
}
```

Listing 2.54: After the refactoring

```
int main() {
    std::string a = "google.co";
    char b[] = "google.ch";
    std::cout <<
        a.compare(0,6,b,0,6);
}
```

2.4.7. strpbrk

The function `strpbrk` has the following signature:

Listing 2.55: Signature of function `strpbrk()`

```
const char* strpbrk(const char* dest, const char* str);
char* strpbrk(char* dest, const char* str);
```

It finds the first character in the C string `dest`, that is also in C string `str` and then returns a pointer to that position in `dest`. If no such character exists, the functions returns `NULL`.

2. Analysis

In the standard header `<algorithm>` there is a function `find_first_of()` that works similarly:

Listing 2.56: Signature of function `find_first_of()`

```
template<class InputIt, class ForwardIt>
InputIt find_first_of(InputIt first, InputIt last,
                    ForwardIt s_first, ForwardIt s_last);
```

Instead of a pointer, it returns an iterator. Listing 2.57 and Listing 2.58 show an example refactoring:

Listing 2.57: Before the refactoring

```
int main() {
    char s[100];
    std::cin >> s;
    char *nr = strpbrk(s, "02468");

    if(nr) {
        std::cout << nr - s;
    }
}
```

Listing 2.58: After the refactoring

```
int main() {
    std::string s;
    std::cin >> s;
    std::string search = "02468";
    auto nr =
        std::find_first_of(s.begin(),
                          s.end(),
                          search.begin(),
                          search.end());

    if(nr != s.end()) {
        std::cout << nr - s.begin();
    }
}
```

In order to be able to use the `find_first_of()` function, the string “02468” needs to be assigned to a separate `std::string` variable. In practice, the plug-in needs to make sure that the name of that variable doesn’t interfere with other variables in the same scope.

2.4.8. `strcspn`

The function `strcspn` has the following signature:

Listing 2.59: Signature of function `strcspn()`

```
size_t strcspn(const char *dest, const char *src);
```

Its functionality is very similar to the one of `strpbrk()`. It returns the length of the initial segment of C string `dest`, that consists only of characters that are not in C string `src`.

2. Analysis

This C string function can be replaced by the `std::string` member function `find_first_of()` which does a similar thing. The signature of the member function `find_first_of()` is shown in Listing 2.60:

Listing 2.60: Signature of member function `find_first_of()`

```
size_type find_first_of(const CharT* s, size_type pos = 0) const;
```

There is a small difference in the return values of the two functions. When the string `dest` only consists of characters that are not contained in the string `src`, the function `strcspn()` returns the length of `dest`. The function `find_first_of()` returns the constant value `std::string::npos` instead.

Listing 2.61 and Listing 2.62 show how the refactoring could still be done:

Listing 2.61: Before the refactoring

```
int main() {
    char s[100];
    std::cin >> s;

    size_t n =
        strcspn(s, "01");

    //do something with n
}
```

Listing 2.62: After the refactoring

```
int main() {
    std::string s;
    std::cin >> s;

    size_t found =
        s.find_first_of("01");
    size_t n =
        (found == std::string::npos) ?
        s.size() : found;
    //do something with n
}
```

2.4.9. `strspn`

The function `strspn()` has the following signature:

Listing 2.63: Signature of function `strspn()`

```
size_t strspn(const char *dest, const char *src);
```

It searches for the first character in `dest` that isn't contained in `src` and then returns the length of the prefix up to that character. For example, if `dest` is "123hello" and `src` is "0123456789" then `strspn()` would return 3 because the first 3 characters in `dest` are all contained in `src`.

2. Analysis

The class `std::string` has several overloads of a member function called `find_first_not_of()` that does a similar thing. The signature of the overload that is the closest match to `strspn()` is shown in Listing 2.64:

Listing 2.64: Signature of member function `find_first_not_of()`

```
size_t find_first_not_of(const char* s, size_t pos = 0) const;
```

Unfortunately, there is a subtle but important difference in the return values of the two functions. When the string `dest` only consists of characters that are also contained in the string `src`, the function `strspn()` returns the length of `dest`. The function `find_first_not_of()` returns the constant value `std::string::npos` instead.

Listing 2.65 and Listing 2.66 show how the refactoring could still be done:

Listing 2.65: Before the refactoring

```
int main() {
    char s[100];
    std::cin >> s;

    size_t n =
        strspn(s, "01");

    //do something with n
}
```

Listing 2.66: After the refactoring

```
int main() {
    std::string s;
    std::cin >> s;

    size_t found =
        s.find_first_not_of("01");
    size_t n =
        (found == std::string::npos) ?
        s.size() : found;
    //do something with n
}
```

2.4.10. memchr

The function `memchr()` has the following signatures:

Listing 2.67: Signatures of function `memchr()`

```
const void * memchr(const void *ptr, int value, size_t num);
void * memchr(void *ptr, int value, size_t num);
```

The function `memchr()` searches through the first “num” bytes of the memory pointed by the “ptr” argument for occurrences of the given “value”. The function returns a pointer to the first occurrence of the value or a null pointer if the value is not found.

2. Analysis

With the “std::find” function a similar behaviour can be achieved. By adding the “num” value to the “begin()” iterator we make sure that only the given characters are passed to the function. For more details see the example below.

Listing 2.68: Before the refactoring

```
int main() {
    const char s[] = "World!";

    char *ptr =
        (char*) memchr(s, 'o', 3);
    print(ptr);
}
```

Listing 2.69: After the refactoring

```
int main() {
    std::string s = "World!";
    auto v = std::find(s.begin(),
        s.begin() + 3, 'o');
    char *ptr = v != s.begin()+3 ?
        &*v : nullptr;
    print(ptr);
}
```

2.5. Modifying C string functions

This section contains possible refactorings of C string functions that modify a string.

2.5.1. strcat / strncat

The functions `strcat()` and `strncat()` have the following signatures:

Listing 2.70: Signature of functions `strcat()` and `strncat()`

```
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, std::size_t count);
```

They append the content of C string `src` to C string `dest`. The buffer for `dest` must have enough space to hold `dest`, `src` and the terminating null character. Both functions return a pointer to `dest`. However, in practice the return value is often ignored.

The `std::string` class has an `append()` member function to concatenate strings but it also overloads the `+=` operator to do basic concatenation which leads to conciser code. See Listing 2.71 and Listing 2.72 for a simple refactoring example:

2. Analysis

Listing 2.71: Before the refactoring

```
int main() {
    char name[100];
    char last_name[100];
    std::cin >> name
        >> last_name;

    strcat(name, " ");
    strcat(name, last_name);
    //do something with name
}
```

Listing 2.72: After the refactoring

```
int main() {
    std::string name;
    std::string last_name;
    std::cin >> name
        >> last_name;

    name += " ";
    name += last_name;
    //do something with name
}
```

The function `strncat()` can be used to append just a part of `src` to `dest`. The programmer can specify the start index by adding a number to the argument for the `src` parameter and the number of characters using the `count` parameter. Listing 2.73 and Listing 2.74 show how the refactoring can be done using the `append()` member function:

Listing 2.73: Before the refactoring

```
int main() {
    const char *url =
        "www.google.com/";
    char s[100] = "TLD: ";
    strncat(s, url+10, 4);
    //do something with s
}
```

Listing 2.74: After the refactoring

```
int main() {
    const std::string url =
        "www.google.com/";
    std::string s = "TLD: ";
    s.append(url, 10, 4);
    //do something with s
}
```

2.5.2. `strdup`

The function `strdup()` creates a mutable copy of an existing C string. Listing 2.75 shows the signature of the function:

Listing 2.75: Signature of the function `strdup()`

```
char *strdup(const char *s);
```

First, it allocates enough memory to hold the contents of the C string `s` and the terminating “\0” character. Then it copies the contents of `s` to the new string and returns it. The code that uses this function has to make sure that the memory for the new string gets freed after it is not used anymore.

2. Analysis

Listing 2.76 shows how `strdup()` is used as a simple way of creating a mutable copy of a const C string. The same thing can be achieved by simply creating a `std::string` and initializing it with the const C string as shown in Listing 2.77. The call to the function `free()` at the end of the program is not necessary anymore.

Listing 2.76: Before the refactoring

```
int main() {
    char *str = strdup("Hello");
    //do something with str
    free(str);
}
```

Listing 2.77: After the refactoring

```
int main() {
    std::string str = "Hello";
    //do something with str
}
```

2.5.3. strcpy

The function `strcpy()` has the following signature:

Listing 2.78: Signature of function `strcpy()`

```
char * strcpy(char *destination, const char *source);
```

The `strcpy` member function copies the characters from a source string into a destination buffer. The destination buffer needs to be at least as large as the source string including its terminating “\0”-character.

One way to get the same behaviour with `std::string` is to initialize the destination string directly with the contents of the source string. A simple refactoring example is shown in Listing 2.79 and Listing 2.80:

Listing 2.79: Before the refactoring

```
int main() {
    char s[] = "HSR";
    char r[4];
    strcpy(r,s);
    std::cout << r;
}
```

Listing 2.80: After the refactoring

```
int main() {
    std::string s = "HSR";
    std::string r;
    r = s;
    std::cout << r;
}
```

It is also possible to use the “`std::copy`” function to refactor this code. Keep in mind that the function `std::back_inserter()` is inefficient when using it for inserting really long strings.

2. Analysis

Listing 2.81: Before the refactoring

```
int main() {
    char s[] = "HSR";
    char r[4];
    strcpy(r,s);

    std::cout << r;
}
```

Listing 2.82: After the refactoring

```
int main() {
    std::string s = "HSR";
    std::string r{};
    std::copy(s.begin(),s.end(),
             std::back_inserter(r));
    std::cout << r;
}
```

2.5.4. strncpy

The function `strncpy()` has the following signature:

Listing 2.83: Signature of function `strncpy()`

```
char * strncpy(char *destination, const char *source, size_t num);
```

It is similar to the `strcpy()` function. In addition, it takes a `num` argument that specifies the number of characters that should be copied from source into destination. The `strncpy()` function can best be replaced with the `std::string` member function `replace()`. The signature of this function is shown in Listing 2.84.

Listing 2.84: Signature of member function `replace()`

```
basic_string& replace(size_type pos, size_type count, const
                    basic_string& str, size_type pos2, size_type count2);
```

An example of how a call to `strncpy()` could be refactored into a call to `replace()` is shown in the following listings:

Listing 2.85: Before the refactoring

```
int main() {
    char a[] = "Hello";
    strncpy(a, "Ha", 2);
}
```

Listing 2.86: After the refactoring

```
int main() {
    std::string a = "Hello";
    a.replace(0, 2, "Ha", 0, 2);
}
```

Another way to refactor this code is to use the “`std::copy_n`” function:

2. Analysis

Listing 2.87: Before the refactoring

```
int main() {
    char s[] = "goal";
    char r[3];
    strncpy(r,s,2);
    r[2] = '\\0';
    std::cout << r << s;
}
```

Listing 2.88: After the refactoring

```
int main() {
    std::string s = "goal";
    std::string r{};
    std::copy_n(s.begin(),2,
        std::back_inserter(r));
    std::cout << r << s;
}
```

2.5.5. memmove

The function `memmove()` has the following signature:

Listing 2.89: Signature of function `memmove()`

```
void * memmove(void *destination, const void *source, size_t num);
```

This function copies the first “num” bytes from the source to the destination. Source and destination can be overlapping. The destination buffer has to be large enough to hold num bytes.

The `memmove()` function can be replaced with the `std::string` member function `replace()` which has the following signature:

Listing 2.90: Signature of member function `replace()`

```
basic_string& replace(size_type pos, size_type count, const
    basic_string& str, size_type pos2, size_type count2);
```

While using the `memmove()` function one has to manually make sure that a “\0” is also copied. The `replace` function always ensures that the resulting string is valid. An example of this refactoring can be found below.

Listing 2.91: Before the refactoring

```
int main() {
    char s[] = "good goal!";
    memmove(s,s+5,4);
    std::cout << s;
}
```

Listing 2.92: After the refactoring

```
int main() {
    std::string s= "good goal!";
    s.replace(0,4,s,5,4);
    std::cout << s;
}
```


2. Analysis

2.5.6. memcpy

The function `memcpy()` has the following signature:

Listing 2.93: Signature of function `memcpy()`

```
void * memcpy(void *destination, const void *source, size_t num);
```

This function copies the first “num” bytes of the source to the destination. Source and destination can not be overlapping otherwise it will lead to undefined behaviour and the size of each of them needs to be at least as big as the given parameter “num”.

There is a `replace()` member function in the `std::string` class that provides similar functionality. The signature of this function is shown in Listing 2.94:

Listing 2.94: Signature of member function `replace()`

```
basic_string& replace(size_type pos, size_type count, const  
    basic_string& str, size_type pos2, size_type count2);
```

Listing 2.95 and Listing 2.96 show how a call to the `memcpy()` function can be refactored into a call to the `replace()` member function:

Listing 2.95: Before the refactoring

```
int main() {  
    char a[] = "Hello";  
    memcpy(a, "Ha", 2);  
}
```

Listing 2.96: After the refactoring

```
int main() {  
    std::string a = "Hello";  
    a.replace(0, 2, "Ha", 0, 2);  
}
```

If `memcpy` is just used to copy a complete C string one can just initialize a new `std::string` with the same value as the source string. The example below demonstrates this case.

Listing 2.97: Before the refactoring

```
int main() {  
    char s[] = "copy";  
    char r[4];  
    memcpy(r,s,4);  
    std::cout << r;  
}
```

Listing 2.98: After the refactoring

```
int main() {  
    std::string s = "copy";  
    std::string r{s};  
  
    std::cout << r;  
}
```

2.6. Converting C string functions

This section contains possible refactorings for C string functions that convert a string into another data type. Because all of these functions use a “const char *” as parameter they can also be used with `std::string` objects because there is a member function called `c_str()` which converts the `std::string` into a “const char *”. Listing 2.99 shows an example:

Listing 2.99: Before the refactoring

```
int main() {
    char s[] = "0.01";
    double n =
        std::atof(s);
}
```

Listing 2.100: After the refactoring

```
int main() {
    std::string s = "0.01";
    double n =
        std::atof(s.c_str());
}
```

2.6.1. atof

The function `atof()` has the following signature:

Listing 2.101: Signature of function `atof()`

```
double atof(const char *str);
```

This function converts a given C string into a double. It will return the converted value. If the converted value is out of range the return value is undefined. If the string can't be converted into a double, the function returns “0.0”.

In the C++ standard library there is a function called “`stod`” that converts a `std::string` into a double. If no conversion can be done a “`std::invalid_argument`” exception will be thrown. A “`std::out_of_range`” exception is thrown if the converted value falls out of range. If a valid input value is provided, the function returns the converted double. The signature of this function can be found below:

Listing 2.102: Signature of function `stod()`

```
double stod(const std::string& str, size_t *pos = 0);
```

In the case of a successful conversion, the two functions behave the same. An example of a simple refactoring can be found below:

2. Analysis

Listing 2.103: Before the refactoring

```
int main() {
    char s[] = "0.01";
    double n = std::atof(s);
    std::cout << n;
}
```

Listing 2.104: After the refactoring

```
int main() {
    std::string s = "0.01";
    double n = std::stod(s);
    std::cout << n;
}
```

However, if the input value is invalid, they behave differently. Therefore, it may not be possible to simply replace the `std::atof()` function with the `std::stod()` function like that. For example, it may be necessary to catch the exception and adapt the error handling accordingly.

2.6.2. atoi / atol / atoll

The functions `atoi()`, `atol()` and `atoll()` are very similar. See their function signatures below.

Listing 2.105: Signature of function `atoi()`, `atol()` and `atoll()`

```
int      atoi(const char *str);
long     atol(const char *str);
long long atoll(const char *str);
```

These functions take a C string and convert it into the data type `int`, `long` or `long long`, respectively. The converted value is returned if the conversion was successful. If the conversion fails, the integer value '0' is returned. If the converted value is out of range the return value is undefined.

Similar functions can also be found in the `<string>` header. They are called `stoi()`, `stol()` and `stoll()`. The signatures of these functions are shown in Listing 2.106:

Listing 2.106: Signatures of member functions `stoi()`, `stol()` and `stoll()`

```
int      stoi(const std::string& str, size_t *pos = 0, int base = 10);
long     stol(const std::string& str, size_t *pos = 0, int base = 10);
long long stoll(const std::string& str, size_t *pos = 0, int base = 10);
```

Also these functions return the same value as their corresponding C string function if the conversion was successful. However, if the conversion could not be performed an “`std::invalid_argument`” exception

2. Analysis

is thrown. An “std::out_of_range” exception is thrown if the resulting value is out of range. The following listings show an example:

Listing 2.107: Before the refactoring

```
int main() {
    char s[] = "42";
    double n = std::atoi(s);
    std::cout << n;
}
```

Listing 2.108: After the refactoring

```
int main() {
    std::string s = "42";
    double n = std::stoi(s);
    std::cout << n;
}
```

2.6.3. strtol / strtoll

The function signatures of the strtol() and strtoll() functions are shown in the listing below.

Listing 2.109: Signatures of functions strtol() and strtoll()

```
long strtol(const char *str, char **str_end, int base);
long long strtoll(const char *str, char **str_end, int base);
```

The functions strtol() and strtoll() convert a byte string into a long or long long. The integer value '0' is returned if no conversion can be done. The out parameter “str_end” returns a pointer to the position in the string up to which the conversion could be performed successfully. For example, if the input string is “123abc” this pointer will be pointed to the position of the letter 'a'.

It is possible to refactor these functions with the stol() or stoll() functions from the <string> header. The signature of these functions can be found in Listing 2.106. In the listing below an example of this refactoring can be found.

Listing 2.110: Before the refactoring

```
int main() {
    char s[] = "42";
    char * pEnd;
    long n =
        std::strtol(s, &pEnd, 10);
    std::cout << n;
}
```

Listing 2.111: After the refactoring

```
int main() {
    std::string s = "42";

    long n =
        std::stol(s);
    std::cout << n;
}
```

2. Analysis

2.6.4. strtoul / strtoull

Both of these functions are similar to strtol and strtoll. They also set the out parameter “str_end” to the position up to which the conversion could be performed successfully. Only the return type is different:

Listing 2.112: Signature of function strtoul() and strtoull()

```
unsigned long strtoul(const char *str, char **str_end, int base);
unsigned long long strtoull(const char *str, char **str_end, int base);
```

These function can be refactored with the “stoul()” and “stoull” functions from the <string> header. The signatures of both functions are listed below.

Listing 2.113: Signature of function stoul() and stoull()

```
unsigned long stoul(const std::string& str, size_t *pos = 0,
                  int base = 10);
unsigned long long stoull(const std::string& str, size_t *pos = 0,
                          int base = 10);
```

The following listings show how the function strtoul() could be refactored:

Listing 2.114: Before the refactoring

```
int main() {
    char s[] = "42";
    char * pEnd;
    unsigned long n =
        std::strtoul(s, &pEnd, 10);
    std::cout << n;
}
```

Listing 2.115: After the refactoring

```
int main() {
    std::string s = "42";

    unsigned long n = std::stoul(s);

    std::cout << n;
}
```

2.6.5. strtod / strtod / strtold

The strtod, strtod and strtold functions have the following signatures:

Listing 2.116: Signatures of functions strtod() strtod() and strtold()

```
float strtod(const char *str, char **str_end);
double strtod(const char *str, char **str_end);
long double strtold(const char *str, char **str_end);
```

They convert a byte string into a corresponding floating point data type. If the conversion fails they return in case of an out of range value an error and in case no conversion can be performed the value

2. Analysis

'0'. The out parameter "str_end" returns a pointer to the position to which the conversion could be performed successfully.

These functions can be refactored with the corresponding conversion functions from the <string> header. Those are called stof(), stod() and stold():

Listing 2.117: Signatures of functions strtod(), strtold() and stof()

```
float stof(const std::string& str, size_t *pos = 0);
double stod(const std::string& str, size_t *pos = 0);
long double stold(const std::string& str, size_t *pos = 0);
```

While the return value of a successful conversion remains the same when using these functions, their behaviour differs if the conversion fails. See an example refactoring below:

Listing 2.118: Before the refactoring

```
int main() {
    char s[] = "3.6e12";
    char * pEnd;
    double n =
        std::strtod(s,&pEnd);
    std::cout << n;
}
```

Listing 2.119: After the refactoring

```
int main() {
    std::string s = "3.6e12";

    double n = std::stod(s);

    std::cout << n;
}
```

2.6.6. strtoumax / strtoumax

The C char functions strtoumax() and strtoumax() have the following signatures.

Listing 2.120: Signature of function strtoumax() and strtoumax()

```
std::intmax_t strtoumax(const char* nptr, char** endptr, int base);
std::uintmax_t strtoumax(const char* nptr, char** endptr, int base);
```

The functions take as many characters as possible from a byte string and convert them into an integer or unsigned integer number. With the base one can define the range of numbers that are used in the byte string to represent the integer. The out parameter "str_end" returns the position to which the conversion could be performed successfully.

2. Analysis

Both member functions can be refactored with `stoll()` or `stoull()`. The signature of these functions can be found in Listing 2.113 and Listing 2.106. An example of this refactoring can be found in the following listings.

Listing 2.121: Before the refactoring

```
int main() {}
char s[] = "123456";
char * pEnd;
std::intmax_t n =
    std::strtod(s,&pEnd);
std::cout << n;
}
```

Listing 2.122: After the refactoring

```
int main() {
    std::string s = "123456";

    long long n = std::stoll(s);

    std::cout << n;
}
```

2.7. Refactoring example

This section contains a possible refactoring of a function from the WebKit Open Source Project[Pro14b]. More information about this project can be found under www.webkit.org. This example shows how the C strings in this function could be refactored to `std::string` objects.

Listing 2.123: Example code to refactor

```
#include "config.h"
#include "EnvironmentUtilities.h"
#include <wtf/text/CString.h>

void stripValuesEndingWithString(const char* environmentVariable,
    const char* searchValue) {
    ASSERT(environmentVariable);
    ASSERT(searchValue);
}
```

The C string parameters can be replaced with `const` references to `std::string` objects since the parameters are not modified inside the function body. The `ASSERT()` statements can be removed because it is not possible to pass `NULL` as an argument to a function that expects a reference parameter.

2. Analysis

Listing 2.124: Possible refactoring

```
#include <cstdlib>
#include <string>
#include <algorithm>

void stripValuesEndingWithString(const std::string &
    environmentVariable, const std::string &searchValue) {
```

Listing 2.125: Example code to refactor

```
// Grab the current value of the environment variable.
char* environmentValue = getenv(environmentVariable);
if (!environmentValue || environmentValue[0] == '\\0')
    return;
```

The function `getenv()` can return `NULL`. In C++, constructing a `std::string` object with char pointer that is `NULL` is undefined behaviour. Therefore, the variable “environmentValue” can’t be directly converted into a `std::string` object:

Listing 2.126: Possible refactoring

```
char *tmp = getenv(environmentVariable.c_str());
if (!tmp || tmp[0] == '\\0')
    return;
std::string environmentValue = tmp;
```

Listing 2.127: Example code to refactor

```
// Set up the strings we'll be searching for.
size_t searchLength = strlen(searchValue);
if (!searchLength)
    return;
```

Because we changed the type of the “searchValue” variable the `size()` member function of the `std::string` class can be used to get the length of the string.

2. Analysis

Listing 2.128: Possible refactoring

```
auto searchLength = searchValue.size();
if (!searchLength)
    return;
```

Listing 2.129: Example code to refactor

```
Vector<char> searchValueWithColonVector;
searchValueWithColonVector.grow(searchLength + 2);
char* searchValueWithColon = searchValueWithColonVector.data();
size_t searchLengthWithColon = searchLength + 1;
memcpy(searchValueWithColon, searchValue, searchLength);
searchValueWithColon[searchLength] = ':';
searchValueWithColon[searchLengthWithColon] = '\\0';
```

Because the vector is just used for the initialization of a C string there is no need for it while using the class `std::string`. The whole content of the string “searchValue” is copied into this C string so a direct initialization of a `std::string` with the correct value does the same.

Listing 2.130: Possible refactoring

```
std::string searchValueWithColon = searchValue;
auto searchLengthWithColon = searchLength + 1;
searchValueWithColon.append(':');
```

Listing 2.131: Example code to refactor

```
// Loop over environmentValueBuffer, removing any components that
// match the search value ending with a colon.
char* componentStart = environmentValue;
char* match = strstr(componentStart, searchValueWithColon);
bool foundAnyMatches = match != NULL;
```

Because the “componentStart” pointer is used afterwards for iteration over the characters it can be replaced with an iterator. Also the “strstr” function call can be replaced with a `std::search` function call that takes iterators as arguments. The calculation of the bool

2. Analysis

value needs to be changed, because the `std::search` function returns an iterator and not a pointer.

Listing 2.132: Possible refactoring

```
auto componentStart = environmentValue.begin();
auto match = std::search(environmentValue.begin(), environmentValue.
    end(), searchValueWithColon.begin(), searchValueWithColon.end());
bool foundAnyMatches = match != environmentValue.end();
```

Listing 2.133: Example code to refactor

```
while (match != NULL) {
    // Update componentStart to point to the colon immediately
    // preceding the match.
    char* nextColon = strstr(componentStart, ":");
    while (nextColon && nextColon < match) {
        componentStart = nextColon;
        nextColon = strstr(componentStart + 1, ":");
    }
}
```

The `strstr()` function calls can be replaced with calls to the corresponding `std::find` function that takes iterators as arguments. Because the variables “match” and “nextColon” are now iterators and not pointers anymore, the checks have to be adapted accordingly as well.

Listing 2.134: Possible refactoring

```
while (match != environmentValue.end()) {
    auto nextColon = std::find(componentStart, environmentValue.end(),
        ':');
    while (nextColon != environmentValue.end() && nextColon < match) {
        componentStart = nextColon;
        nextColon = std::find(componentStart + 1, environmentValue.end(),
            ':');
    }
}
```

2. Analysis

Listing 2.135: Example code to refactor

```
// Copy over everything right of the match to the current
// component start, and search from there again.
if (componentStart[0] == ':') {
    // If componentStart points to a colon, go ahead and copy the
    // colon over.
    strcpy(componentStart, match + searchLength);
} else {
    // Otherwise, componentStart still points to the beginning of
    // environmentValueBuffer, so don't copy over the colon.
    // The edge case is if the colon is the last character in the
    // string, so "match + searchLengthWithoutColon + 1" is the
    // null terminator of the original input, in which case this is
    // still safe.
    strcpy(componentStart, match + searchLengthWithColon);
}

match = strstr(componentStart, searchValueWithColon);}
```

“Strcpy” calls can be replaced with the replace member function of the `std::string` class. The `std::search` function can be used for the “strstr” call.

Listing 2.136: Possible refactoring

```
if (componentStart[0] == ':') {
    environmentValue.replace(componentStart,
        environmentValue.end(),
        match + searchLength,
        environmentValue.end());
} else {
    environmentValue.replace(componentStart,
        environmentValue.end(),
        match + searchLengthWithColon,
        environmentValue.end());
}
match = std::search(componentStart, environmentValue.end(),
    searchValueWithColon.begin(), searchValueWithColon.end());}
```

2. Analysis

Listing 2.137: Example code to refactor

```
// Search for the value without a trailing colon, seeing if the
// original input ends with it.
match = strstr(componentStart, searchValue);
while (match != NULL) {
    if (match[searchLength] == '\0')
        break;
    match = strstr(match + 1, searchValue);
}
```

Again, the `strstr()` calls to search for the corresponding variable can be replaced with calls to the `std::search` function. The check in the while statement needs to be adapted as well.

Listing 2.138: Possible refactoring

```
match = std::search(componentStart, environmentValue.end(),
    searchValue.begin(), searchValue.end());
while (match != environmentValue.end()) {
    if (match[searchLength] == '\0')
        break;
    match = std::search(match + 1,
        environmentValue.end(),
        searchValue.begin(),
        searchValue.end());
}
```

Listing 2.139: Example code to refactor

```
// Since the original input ends with the search, strip out the last
// component.
if (match) {
    // Update componentStart to point to the colon immediately
    // preceding the match.
    char* nextColon = strstr(componentStart, ":");
    while (nextColon && nextColon < match) {
        componentStart = nextColon;
        nextColon = strstr(componentStart + 1, ":");
    }
    // Whether componentStart points to the original string or the
    // last colon, putting the null terminator there will get us the
    // desired result.
    componentStart[0] = '\0';
    foundAnyMatches = true;
}
```

In these two “`strstr`” calls only one character is searched inside the

2. Analysis

string. Therefore, it can be replaced with a `std::find` function call that searches for a single character. The corresponding conditions need to be adapted as well.

Listing 2.140: Possible refactoring

```
if (match != environmentValue.end()) {
    auto nextColon = std::find(componentStart, environmentValue.end(),
                              ':');

    while (nextColon != environmentValue.end() && nextColon < match) {
        componentStart = nextColon;
        nextColon = std::find(componentStart + 1, environmentValue.end()
                              , ':');
    }
    componentStart[0] = '\\0';
    foundAnyMatches = true;
}
```

Listing 2.141: Example code to refactor

```
// If we found no matches, don't change anything.
if (!foundAnyMatches)
    return;
// If we have nothing left, just unset the variable
if (environmentValue[0] == '\\0') {
    unsetenv(environmentVariable);
    return;
}
setenv(environmentVariable, environmentValue, 1);
}
```

Because “setenv” and “unsetenv” take C string parameters the `std::string` objects are converted back into C strings using the `c_str()` member function.

Listing 2.142: Possible refactoring

```
if (!foundAnyMatches)
    return;
if (environmentValue[0] == '\\0') {
    unsetenv(environmentVariable.c_str());
    return;
}
setenv(environmentVariable.c_str(), environmentValue.c_str(), 1);
}
```

3. Implementation

In the Analysis section we described the disadvantages and the use cases of C strings. We also looked at ways to refactor C strings and the standardized functions that are commonly used to analyze or modify them. In this section we write about how we built an Eclipse CDT plug-in that can apply those refactorings automatically and the problems we had to solve along the way.

3.1. Overall architecture and functionality

The following subsections describe the functionality and architecture of the CharWars plug-in. The subsections 3.1.1, 3.1.2, 3.1.3, 3.1.4, 3.1.5, 3.1.6, 3.1.7 and 3.1.8 have been taken out of the Pointerterminator [Gon13] documentation.

3.1.1. The refactoring cycle

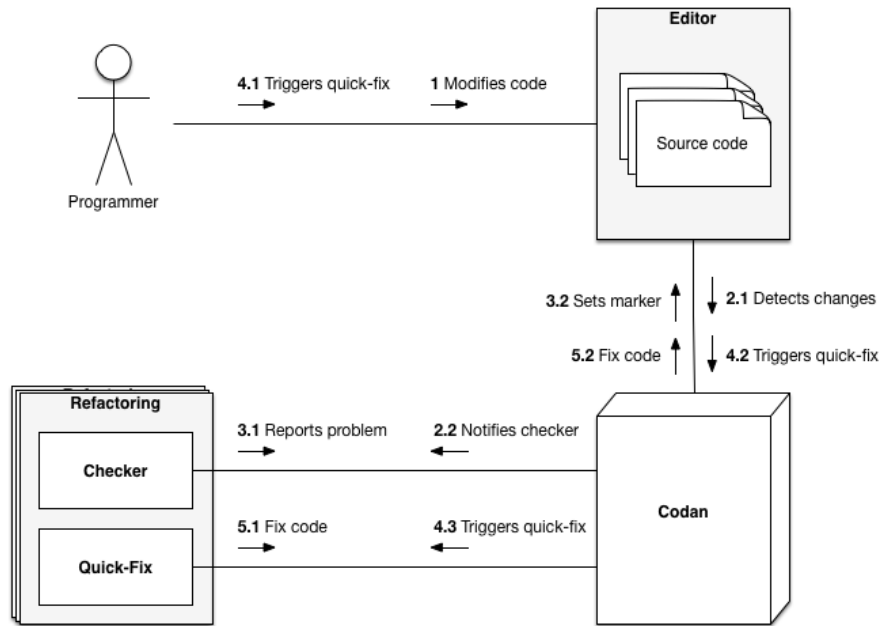
To implement its functionality, the CharWars plug-in relies heavily on Codan[fC14]. Codan is a C/C++ Static Analysis Framework for Eclipse CDT. It provides basic components to build and test a plug-in that does static analysis.

Each refactoring, in turn, consists of a checker and a quick-fix. The typical refactoring cycle is illustrated in Figure 3.1.

1. The programmer modifies the source code.
2. Codan[fC14] detects those changes and notifies all active checkers.
3. Each checker is responsible for a specific problem (e.g, unused variables). After a checker is notified by Codan, it analyzes the code. If it finds an occurrence of its problem, the checker reports it back to Codan. Codan, in turn, sets a marker in the editor to make the programmer aware of the problem.

3. Implementation

Figure 3.1.: Refactoring cycle



4. The programmer can then select the marker and trigger the corresponding quick-fix.
5. Finally, the triggered quick-fix modifies the code in order to fix the problem. Codan writes those changes back to the editor.

3.1.2. Parser and Abstract Syntax Tree (AST)

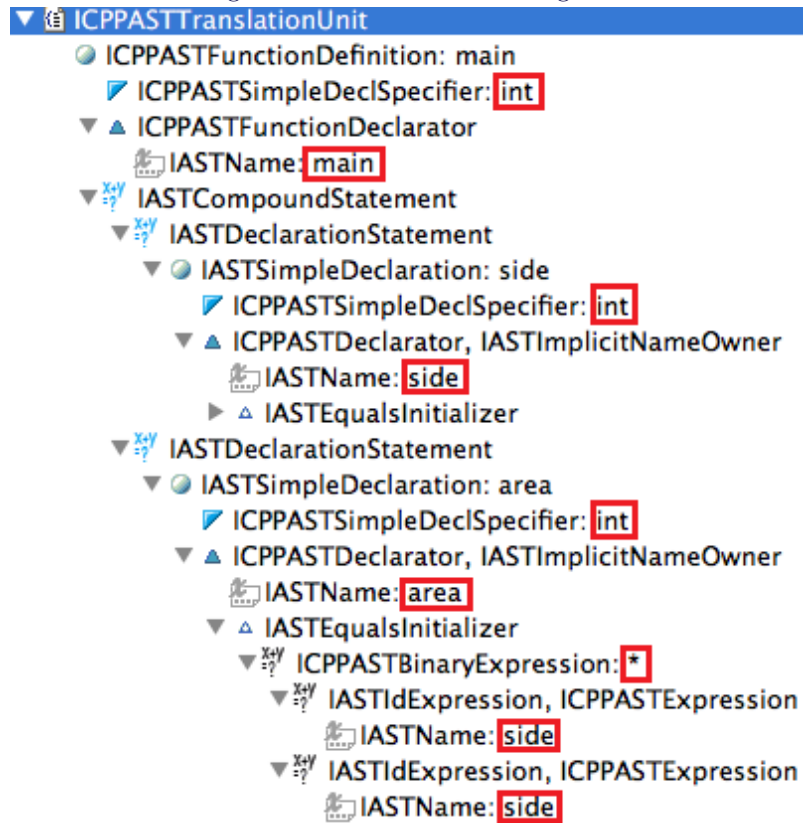
When a cpp-file is opened in an Eclipse CDT editor, the parser creates a tree-representation of the code, which is called the Abstract Syntax Tree (AST). The AST consists of nodes that all implement the IASTNode interface. Each node has one parent node and an array of child nodes. The AST can be used by static analysis tools such as the CharWars plug-in to traverse the code and find problems. Most refactorings can be done by simply modifying and rewriting the AST. Listing 3.1 and Figure 3.2 show an example of what the AST looks like for a short program.

3. Implementation

Listing 3.1: AST example

```
int main() {  
    int side = 2;  
    int area = side * side;  
}
```

Figure 3.2.: AST tree of Listing 3.1



3. Implementation

3.1.3. Bindings

Every C++ identifier (e.g., variable, function, class) is represented as a node of type “IASTName” in the Abstract Syntax Tree. Each such node has a reference to its binding object. Each occurrence of that identifier references the same binding object. For example, if a program has a function called `func()` then there will be a single binding object that represents `func()`. This binding object stores all the information about the `func` identifier, including the locations of the declaration, the definition and all the places where the function is called. The algorithm used to compute the bindings is called “Binding Resolution”. Binding resolution is performed on the AST after the code has been parsed.

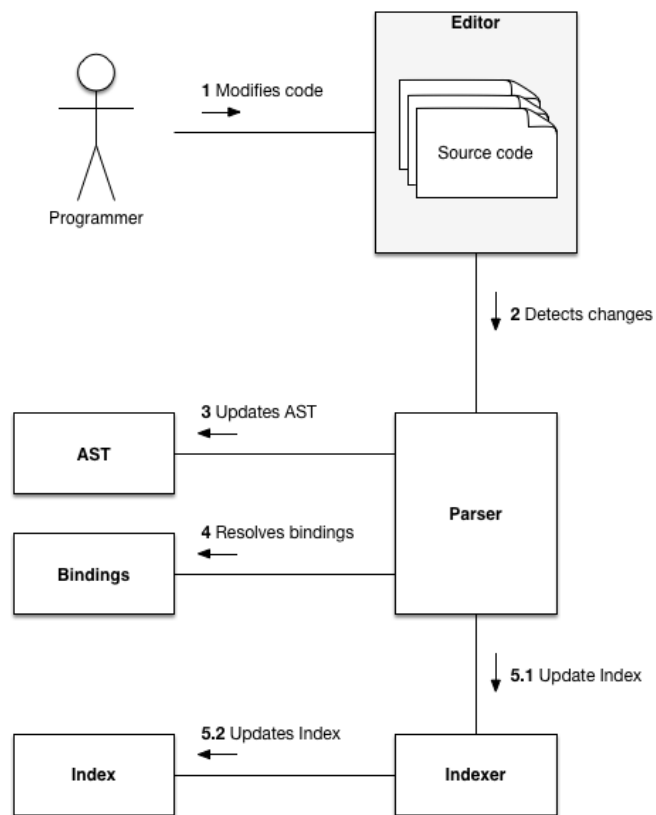
3. Implementation

3.1.4. The index

Parsing and binding resolution is a slow process. Therefore, Eclipse CDT stores the binding information in an on-disk cache called “the index”. To build the index, all the code has to be parsed and all the bindings have to be resolved. The index is then updated every time the programmer edits a file.

Figure 3.3 shows how everything fits together [oP14].

Figure 3.3.: How everything fits together

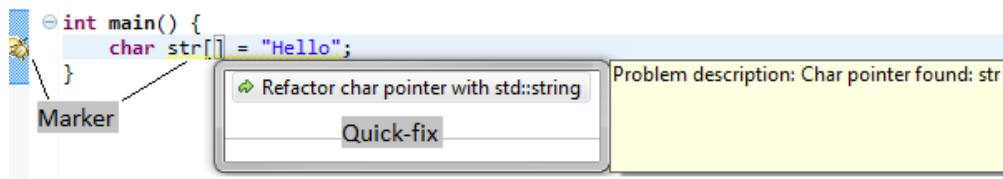


3. Implementation

3.1.5. The plug-in components

The CharWars plug-in consists of a set of checkers and quick-fixes. Each time a file is changed by the programmer, Codan starts the checkers. Each checker traverses through the AST and searches for a specific problem. For example, there is a CharPointerChecker, that searches for C strings that could be refactored to std::string. If a checker reports a problem, a marker is placed in the editor. When the programmer hovers over the marker with the mouse, a description of the problem appears.

Figure 3.4.: Plug-in components



The programmer can choose to apply the refactoring or ignore it. If the programmer applies the refactoring, Codan triggers the corresponding quick-fix in the CharWars plug-in. The quick-fix is then responsible to solve the problem by modifying and rewriting the AST. After the refactoring is done, the quick-fix deletes the marker and returns.

3.1.6. Traversing the AST

Checkers need to be able to traverse the AST in order to find problems in the code. Similarly, quick-fixes traverse the AST to find all occurrences of the refactored variable to do additional adjustments.

The AST is built to be easily traversable using the **Visitor pattern** [Gam94]. Eclipse CDT comes with a few predefined visitors that can be sub-classed to override the visit methods. Only the visit methods that differ from the subclass need to be overridden. Here is an example of a simple checker that uses a visitor to find variables with the name “test” and marks them with a marker. When the user edits a file, Codan automatically calls the checker’s processAst()-method, which

3. Implementation

starts the traversal of the AST using the visitor implemented as an inner class. For more details see the example in Listing 3.2:

Listing 3.2: Visitor example

```
class MyChecker extends AbstractIndexAstChecker {
    public final static String PROBLEM_ID =
        "ch.hsr.pointerterminator.problems.ExampleProblem";

    @Override
    public void processAst(IASTTranslationUnit ast) {
        ast.accept(new ExampleVisitor());
    }

    class ExampleVisitor extends ASTVisitor {
        public ExampleVisitor() {
            shouldVisitNames = true;
        }

        @Override
        public int visit(IASTName name) {
            if(name.toString().equals("test")) {
                reportProblem(PROBLEM_ID, name);
            }
            return PROCESS_CONTINUE;
        }
    }
}
```

3.1.7. Modifying and Rewriting the AST

Eclipse CDT comes with a set of classes that build the infrastructure for modifying code by describing changes to AST nodes. The AST rewriter collects descriptions of modifications to nodes and translates these descriptions into text edits that can then be applied to the original source. It is important to note, that this does not actually modify the original AST. That allows to, for example, show the programmer the changes that will be made by a quick-fix. Listing 3.3 shows a bit of sample code, that replaces a node in the AST, collects the description of the changes in a Change-object and finally performs the change on the original AST[AST14].

3. Implementation

Listing 3.3: AST rewrite example

```
ASTRewrite rewrite = ASTRewrite.create(ast);

rewrite.replace(oldNode, newNode, null);

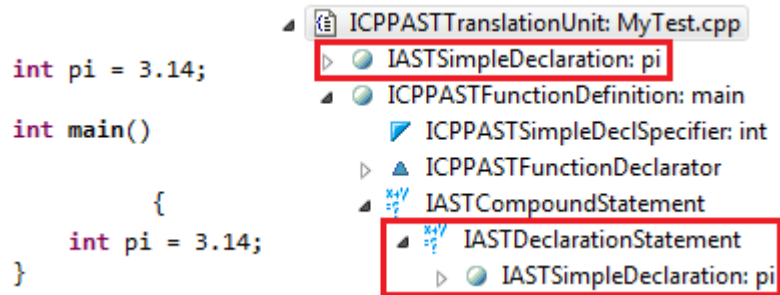
Change c = rewrite.rewriteAST();
try {
    c.perform(new NullProgressMonitor());
    marker.delete();
} catch (CoreException e) {
    e.printStackTrace();
}
```

3.1.8. Dealing with global variables

The C string refactoring has to be able to deal with global variables. Those do have a node structure in the Abstract Syntax Tree that is different from the node structure of local variables. A local variable is defined as a “DeclarationStatement” node in the AST. Inside this “DeclarationStatement” is a nested “SimpleDeclaration” node.

Global variables do not have a “DeclarationStatement” node. Their “SimpleDeclaration” node is a direct child of the root node (TranslationUnit). See Figure 3.5 for an example.

Figure 3.5.: AST structure - Global vs. local variable



3. Implementation

3.1.9. Two-step transformation

Consider the code in Listing 3.4:

Listing 3.4: Before refactoring

```
int main() {
    const char *str = "my string";

    char *found = strstr(str, "ing");
    if(found != nullptr) {
        int index = found - str;
        std::cout << "Found substring at: " << index << std::endl;
    }
}
```

When a programmer uses the plug-in in order to convert the C string `str` into a `std::string` object, this would ideally result in the code shown in Listing 3.5:

Listing 3.5: After refactoring

```
int main() {
    const std::string str = "my string";

    std::string::size_type found_pos = str.find("ing");
    if(found_pos != std::string::npos) {
        int index = found_pos;
        std::cout << "Found substring at: " << index << std::endl;
    }
}
```

This refactoring would involve a lot of changes, some of which the programmer might not expect. For example, the refactoring of the `strstr()` function means that the type of the variable that holds the return value of that function call changes. Then the refactoring may also change the name of that variable in order to reflect its new type and adapt subsequent occurrences of that variable.

Since the programmer initially just wanted to convert the C string into a `std::string` object this can be confusing. Thus, the plug-in performs this refactoring in two steps, each of which have to be triggered by the programmer:

3. Implementation

Step 1 : Char pointer refactoring

In the first step the CharPointerChecker marks C string variables that can be refactored into std::string objects. When a programmer applies the refactoring through a marker, the CharPointerQuickFix starts by replacing the C string definition with the definition of a std::string variable. Then it uses an ASTVisitor to find subsequent occurrences of the variable.

In order to handle the different refactoring cases there is a set of subclasses of the abstract StringRefactoring class. Each subclass can perform a different refactoring. For example, there is a StrlenRefactoring class that can replace a call to the strlen() function with a call to the size() member function. Table 3.1 shows all the StringRefactoring subclasses and how the C string functions are mapped into functions from the <string> / <algorithm> headers.

For each occurrence of the variable, the visitor tries to find an instance of an applicable StringRefactoring subclass and then uses it to refactor that occurrence. Finally, after all occurrences have been refactored, the quick-fix adds the necessary include statements and completes the refactoring by performing a rewrite of the AST.

The process of the Char pointer refactoring is shown in Figure 3.6 in the form of a flow chart.

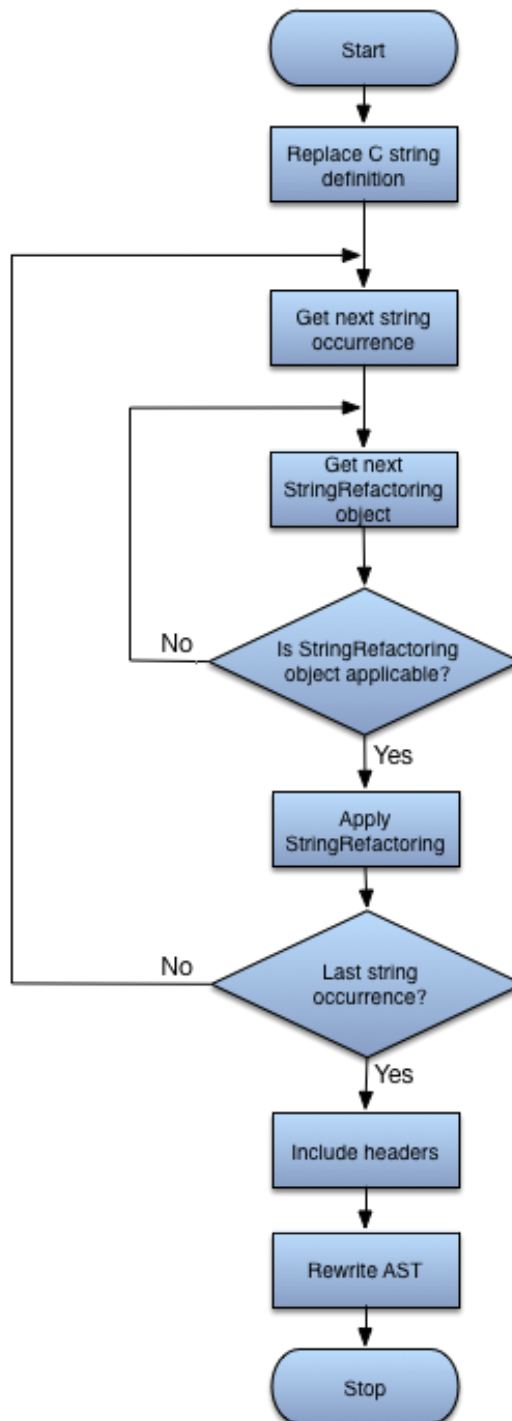
3. Implementation

Table 3.1.: StringRefactoring subclasses

StringRefactoring subclass	C string function	<string> / <algorithm> function
StrlenRefactoring	strlen()	size()
StrcmpRefactoring	strcmp()	== / compare()
StrncmpRefactoring	strncmp()	compare()
MemcmpRefactoring	memcmp()	compare()
StrcatRefactoring	strcat()	+=
StrncatRefactoring	strncat()	append()
StrcpyRefactoring	strcpy()	= / replace()
StrncpyRefactoring	strncpy()	replace()
MemcpyRefactoring	memcpy()	replace()
MemmoveRefactoring	memmove()	0
StrstrRefactoring	strstr()	find()
StrchrRefactoring	strchr()	find_first_of()
StrrchrRefactoring	strrchr()	find_last_of()
MemchrRefactoring	memchr()	std::find()
StrcspnRefactoring	strcspn()	find_first_of()
StrspnRefactoring	strspn()	find_first_not_of()
StrdupRefactoring	strdup()	=
StrpbrkRefactoring	strpbrk()	find_first_of()
ConvertingFunction-Refactoring	atof() / atoi() / atol() / atoll()	std::stod() / std::stoi() / std::stol() / std::stoll()
NullRefactoring	-	-
DefaultRefactoring	-	-

3. Implementation

Figure 3.6.: Flow chart of the Char pointer refactoring



3. Implementation

Applying the Char pointer refactoring to the code in Listing 3.4 results in the code shown in Listing 3.6:

Listing 3.6: After step 1

```
int main() {
    const std::string str = "my string";

    char* found = strstr(&*str.begin(), "ing");
    if(found != nullptr) {
        int index = found - str.c_str();
        std::cout << "Found substring at: " << index << std::endl;
    }
}
```

Step 2 : Char pointer cleanup refactoring

In the second step the Char pointer cleanup refactoring searches C string function calls such as `strstr()`, `strchr()`, etc. that are executed on `std::string` objects. These calls should mostly be the result from executing the Char pointer refactoring as in Listing 3.6. The `CharPointerCleanupChecker` marks such function calls. The programmer can then trigger the corresponding quick-fix via the marker which starts the Char pointer cleanup refactoring. The main job of the refactoring is to replace the C string function with a suitable `std::string` member function. Often, the member function doesn't have the same return type as the C string function. Thus, the variable that holds the return value of the function call and its subsequent occurrences have to be refactored as well. In the case of Listing 3.6 applying the Char pointer cleanup refactoring would lead to the code shown in Listing 3.5.

Sometimes the Char pointer cleanup refactoring isn't as straightforward as in this example. For example, consider the code in Listing 3.7:

Listing 3.7: After step 1

```
int main() {
    const std::string str = "my string";

    char* found = strstr(&*str.begin(), "ing");
    func(found);
}
```

3. Implementation

The main problem is that the `strstr()` function and the `find()` member function behave differently when the second string is not a substring of the first one. While the `strstr()` function returns a `nullptr`, the `find()` member function returns the constant `std::string::npos`. In Listing 3.6 the code had an if statement that verified that the return value captured in the variable `found` was not `NULL`. This meant that the refactoring was able to directly convert from the index returned by the `find()` member function back to a pointer that is equivalent to the pointer returned by `strstr()`. Unfortunately, the code in Listing 3.7 doesn't contain such a `NULL`-check. Therefore, the refactoring has to make sure that the pointer passed to the function `func()` stays the same after the refactoring even if the second string is not a substring of the first one. This leads to the code shown in Listing 3.8:

Listing 3.8: After step 2

```
int main() {
    const std::string str = "my string";

    std::string::size_type found_pos = str.find("ing");
    char* found = found_pos != std::string::npos ? &str[found_pos] :
        nullptr;
    func(found);
}
```

The refactoring added a temporary variable that holds the result of the `find()` function call and uses it to immediately convert back to a pointer. Thus, the subsequent code can be left unchanged because there still is a pointer-variable named `found`.

3.1.10. Default-Refactoring

As described in 3.1.9 the Char pointer refactoring tries to find a `StringRefactoring` subclass that is applicable for every occurrence of the string variable. More precisely, there is a for-loop that loops through an array that contains an instance of each `StringRefactoring` subclass. The method `isApplicable()` is called on each instance. The corresponding `StringRefactoring` then checks whether it is able to handle the occurrence of the string variable and returns an integer. The reason why the return value is an integer and not a boolean has to do with the fact that a single `StringRefactoring` can have multiple

3. Implementation

sub-refactorings each of which would then be denoted with a different integer value. Internally, each class defines an enum which describes the specific sub-refactorings. However, since the StringRefactoring classes have different enums they return an integer instead. A return value of 0 means that the StringRefactoring is not applicable. Every other value means that the StringRefactoring can be applied. Once the for-loop has found an applicable StringRefactoring it calls its apply() method and breaks out of the loop. The order in which the StringRefactoring subclasses are tested doesn't matter because they are mutually exclusive. That means that it isn't possible for two StringRefactoring subclasses to be applicable for the same occurrence of the string variable.

However, there is one exception. The DefaultRefactoring is a special StringRefactoring subclass that should always be the last one to check in the for-loop. It never returns 0 from the isApplicable() method and therefore acts as a fallback refactoring for string variable occurrences that can't be refactored by any of the other StringRefactoring subclasses. In those cases the DefaultRefactoring has to convert the std::string variable back to either a char pointer or a const char pointer depending on the context in which the variable is used. For example, in Listing 3.9 the string variable is passed as an argument to two custom functions. The print() function simply prints the string on the standard output. The makeUppercase() function on the other hand modifies the contents of the string:

Listing 3.9: Before refactoring

```
void print(const char* s) {
    std::cout << s << std::endl;
}

void makeUppercase(char *s) {
    for(int i = 0; i < strlen(s); ++i) {
        s[i] = std::toupper(s[i]);
    }
}

int main() {
    char str[] = "Hello, world!";
    print(str);
    makeUppercase(str);
    print(str);
}
```

3. Implementation

The DefaultRefactoring checks whether the function to which the string variable is passed as an argument expects a char pointer or a const char pointer and adapts the variable accordingly. If the corresponding parameter is a const char pointer the std::string variable can be converted by calling its c_str() member function. Otherwise it uses the iterator returned by the begin() member function and converts it to a char pointer. Therefore, refactoring the str variable in Listing 3.9 leads to the code in Listing 3.10:

Listing 3.10: After refactoring

```
void print(const char* s) {
    std::cout << s << std::endl;
}

void makeUppercase(char *s) {
    for(int i = 0; i < strlen(s); ++i) {
        s[i] = std::toupper(s[i]);
    }
}

int main() {
    std::string str = "Hello, world!";
    print(str.c_str());
    makeUppercase(&*str.begin());
    print(str.c_str());
}
```

3.1.11. Extracting common code

The checkers, quick-fixes and the StringRefactoring classes of the CharWars plug-in require a lot of common code. This code can be divided into three main categories. For each of those categories there is a separate class, that consists solely of public static methods:

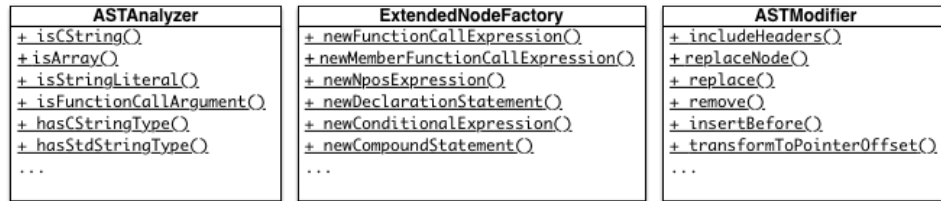
- ***ASTAnalyzer*** to analyze a node or a subtree of the AST.
- ***ExtendedNodeFactory*** to create new nodes or trees of nodes.
- ***ASTModifier*** to modify the AST.

Figure 3.7 is a class diagram of those three classes with some of their methods:

Since a lot of these methods are used both by checkers and quick-fixes which don't belong into the same class hierarchy, it wasn't possible to just put them in a common base class.

3. Implementation

Figure 3.7.: Class diagram



3.2. Problems and Decisions

This section lists the various problems that occurred during the implementation of the refactorings and describes how we solved them.

3.2.1. `std::string` vs. `const std::string`

Whenever the plug-in replaces a C string definition with a `std::string` definition it has to decide whether to make the variable `const` or not. The main goal is to preserve the constness of the original code. Since C strings are actually pointers, they can have four states of constness:

`char *` - strings

A C string variable that is defined as `char *` is not `const` in any way. The characters of the string can be changed and the variable can be repointed to another array of characters. Thus, it only makes sense to make the variable a non-`const` `std::string`.

`const char *` / `char const *` - strings

On the other hand, if a variable is defined to be either a `const char *` or a `char const *` this means that the pointer can be repointed to another array of characters but that the characters themselves can't be changed. Therefore, the decision whether to make the `std::string` `const` or not is not as straightforward as before. However, consider the code in Listing 3.11:

3. Implementation

Listing 3.11: Example of a const char * string

```
int main() {
    const char *email = "example1@hsr.ch";
    //...
    email = "example2@hsr.ch";
}
```

This is valid code which makes it clear, that the resulting `std::string` object can't be const because the reassignment of a const `std::string` is not possible.

char * const - strings

If a variable is defined as *char * const* this means that the variable cannot be pointed to another array of characters. However, the characters within the string can be changed because the variable is a const pointer to char. Therefore, the resulting `std::string` object can't be const because it is not possible to change the characters of a const `std::string`.

const char * const / char const * const - strings

Lastly, a C string that is defined as *const char * const* or *char const * const* cannot be repointed to another string and its characters can't be changed either. Therefore, this is the only situation in which the variable can safely be refactored into a const `std::string`.

3.2.2. `std::string` member functions vs. algorithm functions

As described in section 2.3 both `std::string` member functions and functions from the standard header `<algorithm>` could be used to refactor C string functions. However, during the implementation it became clear that `std::string` member functions are usually the better choice. For example, consider the code in Listing 3.12:

Listing 3.12: Before refactoring

```
int main() {
    const char *email = "example@hsr.ch";
    if(strstr(email, "@hsr.ch")) {
        std::cout << "HSR email address" << std::endl;
    }
}
```

3. Implementation

One possibility to refactor this code would be to use the `search()` function from the standard header `<algorithm>`. This function takes 4 iterators. The first two iterators delimit the string to be searched through while the other two define the string to search after. In most cases the second argument to `strstr()` will either be a C string variable or a literal as in Listing 3.12. Therefore, the plug-in would have to either refactor that C string variable into a `std::string` object or create a new `std::string` variable from the literal that is passed to `strstr()`. The resulting code is shown in Listing 3.13.

Listing 3.13: After refactoring with `search()`

```
int main() {
    const std::string email = "example@hsr.ch";
    const std::string str = "@hsr.ch";
    if (search(email.begin(), email.end(), str.begin(), str.end()) !=
        email.end()) {
        std::cout << "HSR email address" << std::endl;
    }
}
```

In contrast, the same refactoring could be accomplished in a much simpler way using the `std::string` member function `find()`. This is shown in Listing 3.14:

Listing 3.14: After refactoring with `find()`

```
int main() {
    const std::string email = "example@hsr.ch";
    if (email.find("@hsr.ch") != std::string::npos) {
        std::cout << "HSR email address" << std::endl;
    }
}
```

Because this second version of the refactoring is easier to read and easier to implement, the plug-in uses mostly `std::string` member functions to refactor C string functions. As shown in Table 3.1, the only refactoring that uses functions from the `<algorithm>` header instead is the `MemchrRefactoring` which replaces calls to the `memchr()` function with calls to `std::find()`.

3. Implementation

3.2.3. Multiple rewrites in the same AST subtree

As mentioned above, after the Char pointer refactoring replaces the C string definition, it loops through all the occurrences of the variable and tries to find an applicable StringRefactoring for each occurrence. However, this sometimes led to an issue if there were multiple occurrences in the same AST subtree. For example, consider the code in Listing 3.15:

Listing 3.15: Before refactoring

```
int main() {  
    char filename[] = "myfile.txt";  
    strncpy(filename + strlen(filename) - 3, "doc", 3);  
}
```

Figure 3.8 shows a compact version of the Abstract Syntax Tree of the second statement in Listing 3.15:

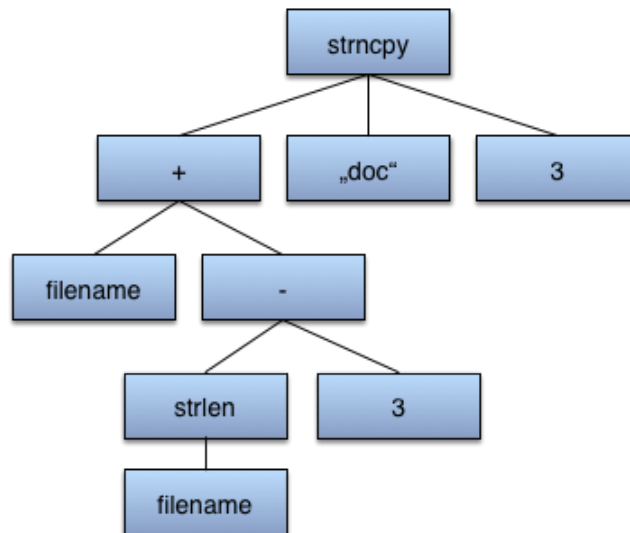


Figure 3.8.: Abstract Syntax Tree of Listing 3.15

The first occurrence of the string variable is handled by the `StrncpyRefactoring` and the second one is handled by the `StrlenRefactoring`. The plug-in uses the built-in `ASTRewrite` class to modify the Abstract Syntax Tree. The way this class works is that it lets you record changes

3. Implementation

to the AST and then performs them all at once when its `rewriteAST()` method is called. In the above example, the `StrlenRefactoring` would first record a change in which the call to `strlen()` is replaced with a call to the `size()` member function. Then the `StrncpyRefactoring` would record a second change in which the call to `strncpy()` is replaced with a call to the `replace()` member function. Unfortunately, it turned out that the `ASTRewrite` class can't handle this refactoring correctly, because the subtree at the `strlen()` node is affected by both recorded changes which caused one change to overwrite the other.

In order to avoid this limitation the plug-in now changes the nodes in each statement directly without using the `ASTRewrite`. Once all occurrences of the variable in the statement have been refactored, the `ASTRewrite` class is used to replace the complete statement at once.

3.2.4. Testing

The Codan[fC14] testing framework has been used to test the Pointerminator plug-in which was the result of our term project. Unfortunately, there were problems with randomly failing tests even if no changes have been done to the code. This seems to happen due to race conditions in the Codan testing infrastructure. Because of that, an alternative testing framework called CDT Testing[cdt14] has been used to test the CharWars plug-in.

CDT Testing has the following benefits:

- The tests check the entire program code not just certain parts of it.
- The code that will be tested is separated from the unit test for better readability.
- In comparison to the Codan testing framework, CDT Testing seems to be more stable and reliable.

Testing checkers

All unit tests for the checkers inherit from an abstract base class that defines the two methods `configureTest()` and `runTest()`. The first method loads the value of the “markerPositions” property which is

3. Implementation

defined in a separate rts-file (see below). This property contains the positions of the markers that ought to be set by the checker. In the `runTest()` method the unit test checks whether the markers at this positions have actually been set. Listing 3.16 shows the implementation of the `runTest()` method:

Listing 3.16: A unit tests for a checker

```
@Override
@Test
public void runTest() throws Throwable {
    if(markerPositions != null) {
        assertProblemMarkerPositions(markerPositions.toArray(
            new Integer[markerPositions.size()]));
    } else {
        assertProblemMarkerPositions();
    }
}
```

The unit test classes load the corresponding rts-files which contain the actual unit-tests using a Java annotation. They also override the method `getProblemId()` to determine which checker should be tested. An example of a unit test class for a checker can be found below:

Listing 3.17: A unit tests class for testing a checker

```
@RunWith(rtsFile="/resources/Checkers/CharPointerChecker.rts")
public class CharPointerCheckerTest extends BaseCheckerTest {
    @Override
    protected String getProblemId() {
        return CharPointerChecker.PROBLEM_ID;
    }
}
```

Inside the rts-file one provides the code that will be used to test the checker. An entry is identified by its test name. First, there is a config section that is used to define the `markerPositions` property. Then there is a section that contains the actual code. Listing 3.18 contains an example:

Listing 3.18: A rts file entry for a checker test expecting a marker in line two

```
#!/CharPointerString
//@.config
markerPositions=2
//@main.cpp
int main() {
    const char *str = "Hello, World!"; //line 2
}
```

3. Implementation

Testing quick-fixes

The quick-fix unit tests also inherit from a base class. The base class contains a method that returns the first marker that was found in the code. It also has two methods to remove all line breaks from the actual and the expected code inside the assert call. This workaround is used because it's hard to configure the formatter for adding the line breaks at the correct position. Also if the project is imported into another Eclipse instance one would need to configure the formatter correctly before running the tests because otherwise some tests may fail.

The unit test classes have one method to get the problem id of the corresponding checker and another method that runs the test by executing the corresponding quick-fix with the marker. The path to the rts-file that contains the test cases is defined as well. In Listing 3.19 an example of a quick-fix unit test is shown:

Listing 3.19: A quick-fix unit test class

```
@RunWith(rtsFile="/resources/QuickFixes/CharPointerQuickFix.rts")
public class CharPointerQuickFixTest extends BaseTest {

    @Override
    protected String getProblemId() {
        return CharPointerChecker.PROBLEM_ID;
    }

    @Override
    @Test
    public void runTest() throws Throwable {
        IMarker firstMarker = getFirstMarker();
        runQuickFix(firstMarker, new CharPointerQuickFix());
        assertEquals(getNormalizedExpectedSource(),
            getNormalizedCurrentSource());
    }
}
```

All tests are defined inside the rts-file that is referenced in the quick-fix unit test class. A test is identified by its name. First, there is a section that contains the code before the refactoring. After that, there is a section with the code that is expected after the refactoring is done. An example is shown below in Listing 3.20:

3. Implementation

Listing 3.20: A quick-fix test

```
#!/CharPointerString
//@main.cpp
int main() {
    char *str = "Hello, World";
}
//=
#include <string>
int main() {
    std::string str = "Hello, World";
}
```

3.2.5. Checking if a variable name exists

In the description of the Char pointer cleanup refactoring (3.1.9) Listing 3.7 and Listing 3.8 showed that it is sometimes necessary to introduce a new variable. Since the new variables hold position values the plug-in takes the name of the original pointer variable and appends “_pos” to it. So for example, in Listing 3.7 the pointer variable is called “found” which means that in Listing 3.8 the name of the new variable is “found_pos”.

However, it could be that a variable with the same name in the same block already exists. This would cause an error to occur after the refactoring is done because two variables with the same name can’t be defined in the same block. If a variable with the same name is just used but not defined within the same block this would also lead to problems, because the new variable would shadow the old one. Therefore, the plug-in has to scan the current block to find out whether a variable with the same name is used or defined in it. It does so using a visitor as shown in subsection 3.1.6. If the variable name is already in use, the plug-in modifies the name by appending an index number to the name and then scans the block again. If the new name is taken as well, it increments the index number and tries again until it finds a free name for the variable. So for example, “found_pos” first becomes “found_pos2”, then “found_pos3” and so on.

3. Implementation

3.2.6. Exception and error handling

If a known exception occurs that can not be corrected by our plug-in it will be logged to the internal error log of Eclipse. This can be done with the built-in logger functionality. An example of such code can be found in Listing 3.21.

Listing 3.21: Logging to internal error log

```
Activator activator = Activator.getDefault();
activator.getLog().log(new Status(Status.ERROR, Activator.PLUGIN_ID,
    Status.OK, "Unable to delete marker", e));
```

If an exception doesn't impact the process of the refactoring like a failed removal of a marker only this logging will take place. An error dialog will be shown to the user for exceptions that cause the refactoring to fail, so the user knows that something went wrong. A screenshot of the dialog that is shown to the user can be found in Figure 3.9.

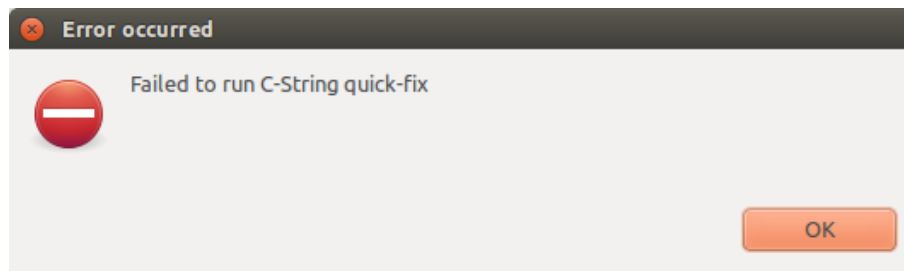


Figure 3.9.: Error dialog-box

Because quick-fixes don't have a way of showing a popup to the user the class Refactoring is used. This class shows user feedback automatically when an error is occurred. In our case the Refactoring class is only used to show the error dialog-box. Therefore, it only creates an error message during the initial condition check that will then automatically be shown to the user.

A Refactoring class can not be created without a RefactoringWizard. Because the RefactoringWizard will not be shown if the initial condition check of the refactoring fails it doesn't need to have any content. The RefactoringWizard can be started with a

3. Implementation

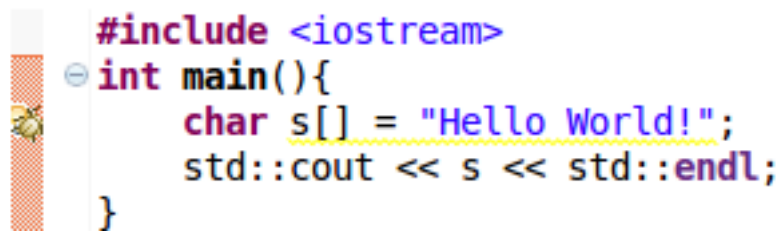
RefactoringWizardOpenOperation[Fel14]. The code that is used to create the error dialog-box can be found in Listing 3.22.

Listing 3.22: Show error dialog-box to user

```
ErrorRefactoring refactoring =
    new ErrorRefactoring(getErrorMsg());
ErrorRefactoringWizard refactoringWizard =
    new ErrorRefactoringWizard(refactoring, 0);
RefactoringWizardOpenOperation op =
    new RefactoringWizardOpenOperation(refactoringWizard);
try {
    op.run(null, "Error occurred");
} ...
```

3.2.7. Marker position calculation

To set a marker, a checker needs to pass a problem location back to Codan. Based on this location the problematic code will be marked in the editor. Get an example from Figure 3.10.



```
#include <iostream>
int main(){
    char s[] = "Hello World!";
    std::cout << s << std::endl;
}
```

Figure 3.10.: Problem marker

IASTNode objects have a method called “getNodeLocations()” that allows a programmer to get the location of a node. This method returns an array of IASTNodeLocation objects. Each IASTNodeLocation consists of an offset and a length. Normally, the array only contains one IASTNodeLocation object which fully describes the location of the node.

In special cases there are more than one IASTNodeLocation to describe the full location of the node. For example, if macros are used inside a node, there is one IASTNodeLocation object that describes the location of the code before the macro, another one that describes the location of the code after the macro and a third one to describe

3. Implementation

the location of the macro itself. Unfortunately, this last IASTNodeLocation object always has an offset of 1 and a length of 0. An example of this case is illustrated in Figure 3.11. It represents the locations of the node “s[] = HI” in Listing 3.23.

Listing 3.23: Example code with macro

```
#include <iostream>
#define HI "Hello World!"

int main() {
    char s[] = HI; //char pointer with macro
    std::cout << s << std::endl;
}
```

Name	Value
▼ locations	IASTNodeLocation[2] (id=153)
▼ ▲ [0]	ASTFileLocation (id=156)
fLength	5
fLocationCtx	LocationCtxFile (id=167)
fOffset	94
▼ ▲ [1]	ASTMacroExpansionLocation (id=157)
fContext	LocationCtxMacroExpansion (id=163)
fLength	1
fOffset	0

Figure 3.11.: IASTNodeLocation array of Listing 3.23

When a macro is used in the middle of a node one can just use the offset of the first IASTNodeLocation object to get the start position of the node. The end position of the node can be calculated by building the sum of the offset and the length of the last IASTNodeLocation object. But this calculation will not work if the macro is at the end of the node. In this case the last IASTNodeLocation object can not be used to calculate the correct end position because it has wrong offset and length values.

3. Implementation

A workaround to calculate the correct end position of the node is to take the offset of the first location and add to it the length of the node's "RawSignature". So the node will be marked and not the whole line that contains the node. The code for this workaround is shown in Listing 3.24.

Listing 3.24: Calculate positions of node

```
IASTNodeLocation[] nodeLocations = node.getNodeLocations();
IASTNodeLocation firstLoc = nodeLocations[0];
int start = firstLoc.getNodeOffset();
int end = firstLoc.getNodeOffset() + node.getRawSignature().length();
```

4. Refactoring real-life code

This section describes how the plug-in performs in real-life situations and which C string functions are frequently used. It also shows in which context the functions are normally used.

4.1. Statistics

The top 100 C++ repositories on Github[Git14c] have been used in May 2014 to create the statistics. The repositories have been sorted according their Github star rating. This list of repositories contains well-known projects such as “node-webkit”, “texmate”, “mongo db”, “xbmc” and “fish-shell”.

The repositories were scanned to find occurrences of the various C string functions that the plug-in supports. Afterwards, the context in which each function is used was analyzed and categorized according to certain patterns.

The CharWars plug-in only supports these functions if they are used with C string arguments. If a function like “memchr” is used to search a byte in something other than a C string, it can not be refactored.

As shown in Table 4.1 we differentiated between the following contexts:

- **If statement:** The function call happens directly inside an If statement condition.
- **Assignment:** The return value of the function call is assigned to a variable.
- **Return value:** The result of the function call is returned from another function.
- **Single statement:** The function is just called in a separate statement. The return value is not captured.
- **Other:** Everything that is not recognized by a pattern.

4. Refactoring real-life code

Table 4.1.: Ocurrence statistics

Function name	If statement	Assignment	Return value	Single statement	Other
<code>strlen</code>	164	155	4	0	349
<code>strcmp</code>	1507	39	105	0	283
<code>strncmp</code>	559	53	50	1	158
<code>memcmp</code>	447	90	137	36	387
<code>strcat</code>	6	1	0	383	23
<code>strncat</code>	1	0	0	67	1
<code>strdup</code>	8	349	34	0	85
<code>strcpy</code>	18	4	1	1168	56
<code>strncpy</code>	22	1	16	594	12
<code>memmove</code>	3	0	6	403	72
<code>memcpy</code>	8	7	7	1446	108
<code>strchr</code> *	133	613	17	0	192
<code>strrchr</code> *	3	254	0	0	24
<code>strstr</code> *	292	250	24	2	121
<code>strpbrk</code> *	9	27	0	0	11
<code>strcspn</code> *	0	13	0	2	5
<code>strspn</code> *	2	9	0	0	3
<code>memchr</code> *	7	59	4	8	42

For the functions that have a star next to their name in the table there exists a two-step refactoring as described in subsection 3.1.9.

4.2. Refactoring XBMC

The XBMC repository has been used to test the CharWars plug-in. We took a snapshot of the application's source code (in May 2014) from Github[xG14] and tried to apply as many C string refactorings as possible. More information about XBMC can be found under xbmc.org.

4. Refactoring real-life code

4.2.1. First real-life test

The plug-in added 776 `std::string` markers in total. Because the XBMC source code also contains C code and the plug-in can't differentiate between C and C++ code some markers can not be resolved. These markers were omitted for the creation of the statistics.

Due to the fact that resolving all markers would exceed the scope of this thesis only the first 150 have been checked. All markers have been tested without changing anything manually. If the code compiled afterwards without errors the marker counted as "solved" otherwise it counted as "unsolved". Table 4.2 shows the amount of resolved and unresolved markers.

Table 4.2.: Refactoring statistics

Markers set	Markers tested	Solved	Unsolved
776	150	72 (48%)	78 (52%)

In the following subsections there are some examples of C string functions that have been found inside the XBMC code and could be refactored correctly with the CharWars plug-in. To provide for as many functions as possible an example sometimes some small code changes have been taken before applying the refactoring.

strlen

The `strlen` function is used in a wide variety of contexts. Many calls are inside If-statement conditions and assignments. The function is also often used for index calculations, asserts and function arguments.

If `strlen` is used to calculate the length of a string literal it can not be refactored with our plug-in.

The code of the following example that could be successfully refactored can be found in the file `lib/UnrarXLib/pathfn.cpp` inside XBMC's code.

4. Refactoring real-life code

Listing 4.1: Before the refactoring

```
char cIllegalChars [] =
    "<>=?;\\"**+,/|";
unsigned int iIllegalCharSize =
    strlen(cIllegalChars);
```

Listing 4.2: After the refactoring

```
std::string cIllegalChars =
    "<>=?;\\"**+,/|";
unsigned int iIllegalCharSize =
    cIllegalChars.size();
```

strcmp

This function is mostly used inside If-statement conditions.

The following code that is located inside `xbmc/linux/PosixMount-Provider.cpp` contains several `strcmp` calls that can be refactored correctly with our plug-in.

Listing 4.3: Before the refactoring

```
const char* fs = "";
...
if (strcmp(fs, "fuseblk") == 0
|| strcmp(fs, "vfat") == 0
|| strcmp(fs, "ext2") == 0
|| strcmp(fs, "ext3") == 0
|| strcmp(fs, "reiserfs") == 0
|| strcmp(fs, "xfs") == 0
|| strcmp(fs, "ntfs-3g") == 0
|| strcmp(fs, "iso9660") == 0
|| strcmp(fs, "exfat") == 0
|| strcmp(fs, "fusefs") == 0
|| strcmp(fs, "hfs") == 0)
```

Listing 4.4: After the refactoring

```
std::string fs = fsStr;
...
if (fs == "fuseblk"
|| fs == "vfat"
|| fs == "ext2"
|| fs == "ext3"
|| fs == "reiserfs"
|| fs == "xfs"
|| fs == "ntfs-3g"
|| fs == "iso9660"
|| fs == "exfat"
|| fs == "fusefs"
|| fs == "hfs")
```

strncmp

Like “`strcmp`” this function is also used mostly inside If-statements. It is not used as frequently as `strcmp`.

Below is an example of a successfully refactored example that can be found inside the file `xbmc/guilib/XBTfReader.cpp`. To be able to refactor this code one needs to change the declaration of the C string into an initialization. After applying the quick-fix this initialization can be removed again.

4. Refactoring real-life code

Listing 4.5: Before the refactoring

```
char magic[4] = "";
...
if (strcmp(
    magic,
    XBTF_MAGIC,
    sizeof(magic)) != 0){
    return false;}

```

Listing 4.6: After the refactoring

```
std::string magic = "";
magic.reserve(4);
...
if (magic.compare(0,
    sizeof(magic.c_str()),
    XBTF_MAGIC, 0,
    sizeof(magic.c_str())) != 0){
    return false;}

```

memcmp

“Memcmp” is a function that is often used inside If-statements. It is also frequently used as a return value or in an assignment to a variable.

The following example can be found inside `xbmc/guilib/AnimatedGif.cpp`. To successfully refactor it one needs to change the definition of the string into an initialization. After the refactoring has been done one can remove the initialization again.

Listing 4.7: Before the refactoring

```
char szSignature[6] = "";
...
if(memcmp(szSignature, "GIF", 2) !=
    0) {
    ...
}

```

Listing 4.8: After the refactoring

```
std::string szSignature = "";
szSignature.reserve(6);
...
if(szSignature.compare(0, 2, "GIF",
    0, 2) != 0) {
    ...
}

```

strcat

This function is typically used on its own in a separate statement. An occurrence that can be refactored with the CharWars plug-in could be found inside `lib/libmodplug/src/load_pat.cpp`:

Listing 4.9: Before the refactoring

```
static char timiditycfg[128] = "";
...
strcat(timiditycfg,
    "/timidity.cfg");

```

Listing 4.10: After the refactoring

```
static std::string timiditycfg = "";
...
timiditycfg +=
    "/timidity.cfg";

```

4. Refactoring real-life code

strncat

This function is used sparsely. It is used mostly as a single statement. Out of three occurrences that could be found inside the XBMC source code none of them could be refactored correctly.

strdup

This function is frequently used inside assignments and as return value. In the XBMC source code it is often used as return value which can't be handled correctly by the CharWars plug-in.

strcpy

With more than a thousand occurrences in the top 100 repositories the strcpy function is used primarily on its own in a separate statement.

The following example that can be found inside lib/libmodplug/src/load_pat.cpp shows how this function is refactored by the plug-in:

Listing 4.11: Before the refactoring

```
static char timiditycfg[128] = "";
...
strcpy(timiditycfg, p);
```

Listing 4.12: After the refactoring

```
static std::string timiditycfg = "";
...
timiditycfg = p;
```

strncpy

Like the strcpy function this function is also used mainly as a separate statement.

The following occurrence that could be successfully refactored is located inside tools/TexturePacker/SDL_anigif.cpp.

Listing 4.13: Before the refactoring

```
char version[4];
...
strncpy(version, (char*)buf+3, 3);
version[3] = '\0';
if((strcmp(version, "87a") != 0)
    && (strcmp(version, "89a") != 0)) {
```

Listing 4.14: After the refactoring

```
std::string version = "";
version.reserve(4);
...
version.replace(0, 3, (char*) (buf)
    + 3, 0, 3);
version[3] = '\0';
if((version != "87a")
    && (version != "89a")) {
```

4. Refactoring real-life code

memmove

Memmove is a function that is often used in separate statements. In the XBMC code the memmove function is mostly used with buffers that don't represent strings. These cases can't be handled by the CharWars plug-in.

memcpy

Also this function is used mostly as a separate statement. One occurrence that is used to copy C strings can be found in the file lib/lib-modplug/src/sndfile.cpp. See an example of the refactoring below:

Listing 4.15: Before the refactoring

```
char sztmp[40] = "";
memcpy(sztmp,
       m_szNames[nSample], 32);
```

Listing 4.16: After the refactoring

```
std::string sztmp = "";
sztmp.reserve(40);
sztmp.replace(0, 32,
             m_szNames[nSample], 0, 32);
```

strchr

The strchr function is typically used inside assignments or if statement conditions.

The following example that could successfully be refactored can be found inside the file xbmc/lib/timidity/timidity/m2m.cpp:

Listing 4.17: Before the refactoring

```
char program_str[20] = ""
...
if (strchr(program_str,
           '!'))
```

Listing 4.18: After the refactoring

```
std::string program_str = "";
program_str.reserve(20);
...
if (program_str.find_first_of('!')
    != std::string::npos)
```

strchr

The strchr function is also often used inside assignments.

An occurrence that shows the typical usage and could be refactored correctly is inside the following file: xbmc/linux/LinuxTimezone.cpp. The char pointer cleanup refactoring has not been performed because the variable "p" is afterwards modified with pointer operators which can't be handled by the CharWars plug-in.

4. Refactoring real-life code

Listing 4.19: Before the refactoring

```
char timezoneName[255];

timezoneName[rlrc] = '\0';
...
char* p = strrchr(timezoneName,
                  '/');
```

Listing 4.20: After the refactoring

```
std::string timezoneName = "";
timezoneName.reserve(255);
timezoneName[rlrc] = '\0';
...
char* p = strrchr(&*timezoneName.
                  begin(), '/');
```

strstr

The strstr function is frequently used inside if statement conditions and assignments.

To get a working example one needs to manually change an if statement that does a NULL check. The code is located inside `/xbmc/xbmc-c/cores/dvdplayer/DVDInputStreams/DVDInputStreamHTSP.cpp`

Listing 4.21: Before the refactoring

```
const char* method = "";
...
if (strstr(method,
           "channelAdd"))
    CHTSPSession::ParseChannelUpdate(
        msg, m_channels);
else if (strstr(method, "
channelUpdate"))
    CHTSPSession::ParseChannelUpdate(
        msg, m_channels);
else if (strstr(method, "
channelRemove"))
    CHTSPSession::ParseChannelRemove(
        msg, m_channels);
```

Listing 4.22: After the refactoring

```
std::string method = "";
...
if (method.find("channelAdd") !=
    std::string::npos)
    CHTSPSession::ParseChannelUpdate(
        msg, m_channels);
else if (method.find("channelUpdate")
        != std::string::npos)
    CHTSPSession::ParseChannelUpdate(
        msg, m_channels);
else if (method.find("channelRemove")
        != std::string::npos)
    CHTSPSession::ParseChannelRemove(
        msg, m_channels);
```

strpbrk

With forty occurrences in the top 100 C++ projects strpbrk is not used very frequently. The function is typically used inside assignments.

The following example from the file `xbmc/filesystem/iso9660.cpp` shows an assignment and a condition that could be refactored successfully with the plug-in:

4. Refactoring real-life code

Listing 4.23: Before the refactoring

```
char *pointer = ""
...
pointer = (char*)filename;
while(strpbrk(pointer, "\\\/")) {

    pointer = strpbrk(pointer, "\\\/")
        + 1;
}
```

Listing 4.24: After the refactoring

```
std::string pointer = "";
...
pointer = (char*)filename;
while (pointer.find_first_of("\\\/")
    != std::string::npos)
    std::string::size_type pointer_pos
        = pointer.find_first_of("\\\/");
    pointer = pointer_pos != std:::
        string::npos ? &pointer[
            pointer_pos] : nullptr + 1;
```

strcspn

The `strcspn` function is also used sparsely in the top 100 C++ projects.

There are two occurrences inside the code of XBMC. None of them could be refactored correctly because in both cases there are pointer operators that modify the content of the C string pointer.

strspn

`Strspn` is only used fourteen times in the top 100 projects, typically inside an assignment.

Only one occurrence of the function `strspn` could be found inside the XBMC code. Because the pointer variable is manually modified using pointer arithmetic, the plug-in was unable to handle this case.

memchr

With a bit more than hundred occurrences in the top 100 repositories `memchr` is used more often. It can mainly be found inside assignments.

None of the three occurrences in the XBMC project could be refactored, mainly because the function wasn't used to search inside a C string.

4. Refactoring real-life code

4.2.2. Second real-life test

In the first round of tests many occurrences could not be refactored because the string variables were defined at namespace or class level. Because of that the CharWars plug-in was unable to refactor them.

Therefore, we improved the plug-in to support these cases and created the statistics a second time. Again 150 occurrences have been tested and the amount of successfully refactored occurrences by the CharWars plug-in increased 17 percent. The result can be found in Table 4.3.

Table 4.3.: Refactoring statistics

Markers set	Markers tested	Solved	Unsolved
776	150	98 (65%)	52 (35%)

4.3. Where the plug-in needs manual corrections

This section describes how in some cases the plug-in doesn't have enough information to determine whether a variable is a C string or not. Sometimes it is then possible to do some manual adjustments that cause the plug-in to behave correctly. It also describes in which cases the plug-in may fail to get a correct result.

4.3.1. How to refactor C string definitions

To avoid producing code that doesn't work, only C strings that are defined and initialized in the same statement are marked by the checker. This way we can be sure, that the pointer isn't just a pointer to a single character.

With a small change one can also refactor a C string that is initialized later. First one needs to be sure that the pointer does actually point to a C string. Then the definition can temporarily be changed into an initialization with an empty string literal. After that, the plug-in

4. Refactoring real-life code

marks the string and the automated refactoring can be performed. Finally, the manual changes can be undone.

Listing 4.25: Original code

```
char *gender;
if (isMasculine()){
    gender = "masculine";
} else {
    gender = "feminine";
}
```

Listing 4.26: Code to refactor

```
char *gender = "";
if (isMasculine()){
    gender = "masculine";
} else {
    gender = "feminine";
}
```

4.3.2. How to refactor C string assignments

If a C string is initialized with a function call or another variable, it won't be marked because the assigned value could be NULL or a pointer to a character instead of a C string. If the programmer feels certain that the C string is always initialized with a valid string, the plug-in can still be used. To be able to refactor such variables one needs to do the following: First, add a statement that defines and initializes the variable with an empty string literal. Change the old definition into an assignment below the new definition. Now the code can be refactored with the plug-in. After the refactoring the temporary changes can be removed again.

Listing 4.27: Original code

```
char *name = person.getName();
std::cout << "Welcome " << name;
```

Listing 4.28: Code to refactor

```
char *name = ""
name = person.getName();
std::cout << "Welcome " << name;
```

4.3.3. How to refactor C string parameters

To be able to refactor C string parameters one also needs to make some manual changes. First, one has to make sure that the function is never called with a NULL argument. After that one needs to temporarily rename the parameter and add a local C string variable with the original parameter name. The refactoring is then performed on this new variable. After the refactoring, the new variable can be removed and the parameter can be turned into a `std::string` object with its original name.

4. Refactoring real-life code

Listing 4.29: Original code

```
void printString(char *s){  
    std::cout << s;  
}
```

Listing 4.30: Code to refactor

```
void printString(char *tmp_s){  
    char *s = "";  
    std::cout << s;  
}
```

4.3.4. Known issues

Problems that may occur while using this plug-in are described in this section.

Position of includes

The correct position of the includes that will be added during the refactoring can not be calculated correctly if the code contains if directives like “#if”, “#else” or “#endif”. The position will also not be calculated correctly if there are includes between the code.

In such cases it is recommended to add the includes manually before the refactoring is performed. The plug-in checks if the includes already exist and will not include them.

Global variables

Global variables that are defined as extern inside header files will also not be refactored correctly because the data type of the external definition also needs to be changed. This change has to be performed manually. It can be done before or after the refactoring.

Pointer operators

This plug-in will fail to correctly refactor C string pointers that are manipulated with pointer operators. In these cases a manual rewrite of the program logic is necessary.

Resource allocation

If a C string is allocated on the heap and is used across multiple blocks as a shared resource, the CharWars plug-in can't refactor it correctly. In this case the refactoring has to be performed manually.

4. Refactoring real-life code

C files

Files containing C code are automatically scanned by Codan. Therefore, these files could also contain some markers from the CharWars plug-in. Because `std::string` only works in C++ the refactoring doesn't work and these markers can't be resolved. In this case the markers can be ignored or some components of the plug-in can be deactivated.

NULL checks

While a C string can be a `nullptr` and it makes sense to compare it against `NULL` a `std::string` can not be a `nullptr`. Therefore, all `NULL` checks of the string will not be needed any more. The programmer may need to change some parts of the logic or use `std::optional` to achieve the same behaviour as the original program.

5. Conclusion

This chapter describes the results of the CharWars bachelor thesis. It also describes how this project can be continued and the plug-in can be extended and improved.

With 65 percent of successfully refactored C strings inside XBMC [xG14] many cases of the C string functions are covered by the plug-in. With some manual changes before or after triggering the refactoring even more C strings could be refactored. There are only a few cases where the code can't be refactored even after making some manual changes.

5.1. Achievements

The following achievements were made during the bachelor thesis:

- The C string functions have been analyzed and compared to corresponding `std::string` member functions.
- Refactorings for the C string functions have been implemented and continuously tested with unit tests.
- For special C string functions a second refactoring has been programmed to provide more flexibility and compatibility.
- A refactoring for a subset of the converting C functions (e.g., `atol()`) has been programmed.
- The plug-in has been tested with a real-life project and the results have been documented.

5. Conclusion

5.2. Future Work

The CharWars plug-in is an improvement over the existing Pointer-minator [Gon13] plug-in. It provides a lot more functionality and is well tested. However, there is still plenty of room for improvement. Here are some of the features that could be added to the plug-in in a future project:

- Refactoring of strings that are allocated on the heap
- Refactoring of string parameters
- Refactoring of string return values

A. User manual

This chapter describes how to de-/install the CharWars plug-in, how to use it and how some parts of it can be deactivated.

A.1. Installation

The CharWars plug-in requires the Eclipse CDT IDE (preferably the Kepler release or newer) and at least Java 1.6 installed on the system.

To install the plug-in first click on “Help” and select “Install New Software”.

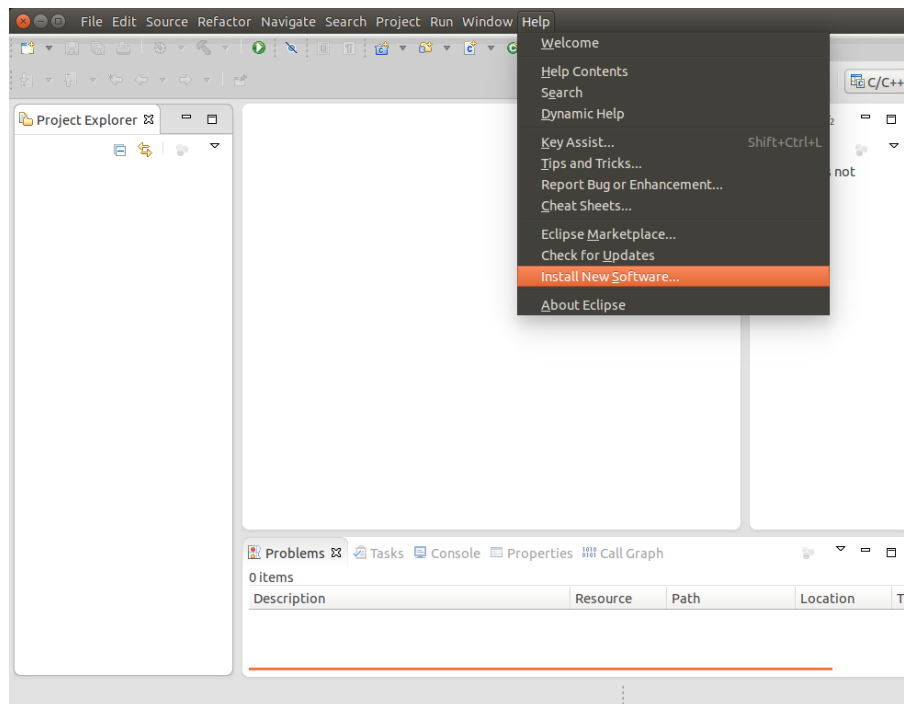


Figure A.1.: Install plug-in

A. User manual

Enter the plug-in url under “Work with:” and check the check-box that is shown next to the plug-in name.

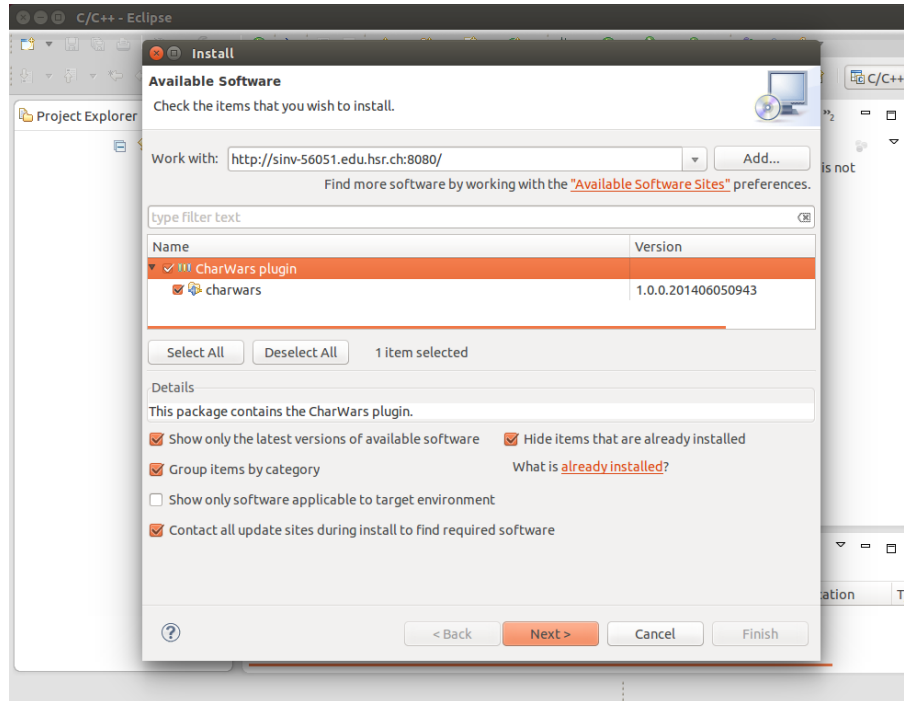


Figure A.2.: Install plug-in

Press next to go through the wizard and install the plug-in. At the end a prompt will ask you whether you want to restart Eclipse. Click “Yes”. After the restart you should be able to use the CharWars plug-in.

A.2. Usage and configuration

This section shows how the plug-in can be used and how parts of it can be deactivated.

A.2.1. Usage

The CharWars plug-in sets problem markers inside Eclipse. Markers can be selected with a left-click on the bug icon or with a corresponding short-cut (Ctrl+1 or Cmd+1, depending on your operating system) when the cursor is inside the marked code. This opens a new popup that shows the possible quick-fixes that can be applied.

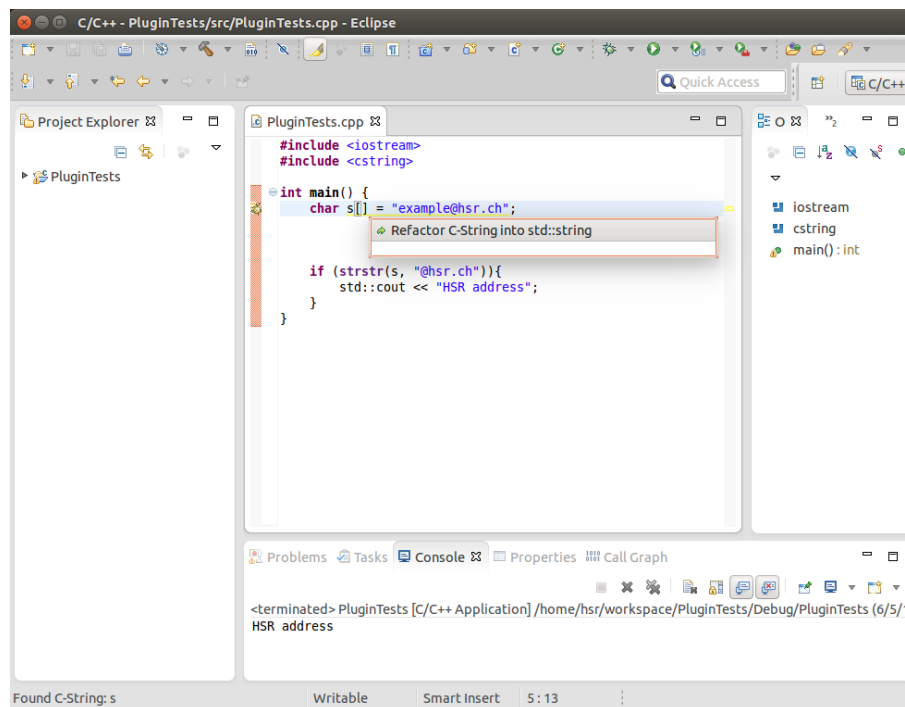


Figure A.3.: Resolving a problem marker

A. User manual

Pressing on the corresponding quick-fix will start the refactoring process of the CharWars plug-in. After the refactoring is done one can review the code and save the changes. Sometimes the code can still be improved by doing some manual changes. The changes can be reverted by pressing “Undo”.

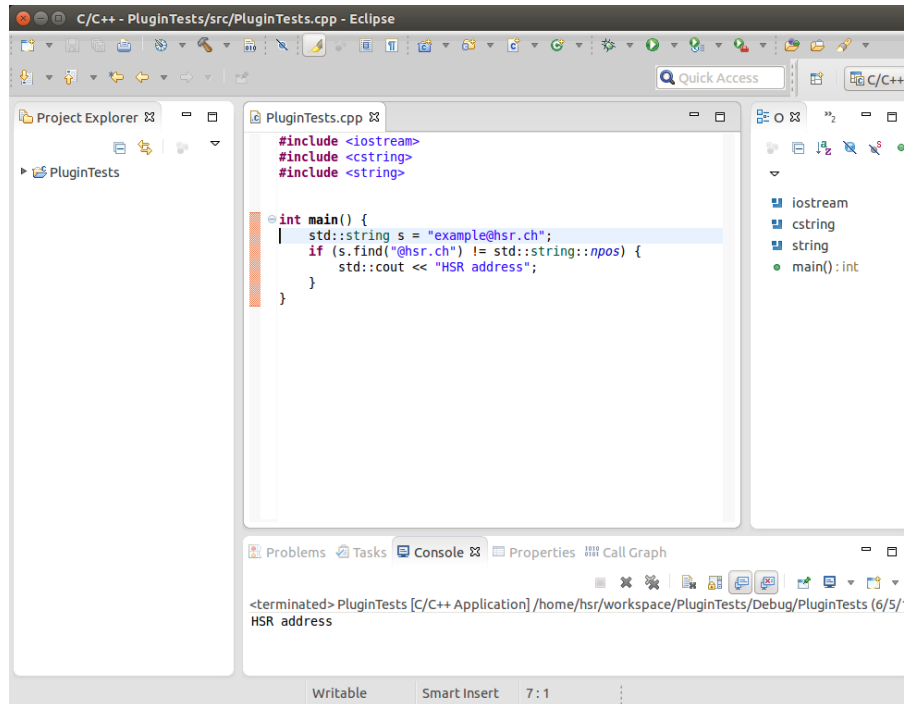


Figure A.4.: Resolving a problem marker

A. User manual

A.2.2. Configuration

The CharWars plug-in contains four checkers. One is used to set markers on C arrays, one for reference parameters and two for C strings. You can deactivate and reactivate these four markers individually. The following needs to be done to deactivate or reactivate a marker:

First you need to press on “Windows” and select “Preferences”.

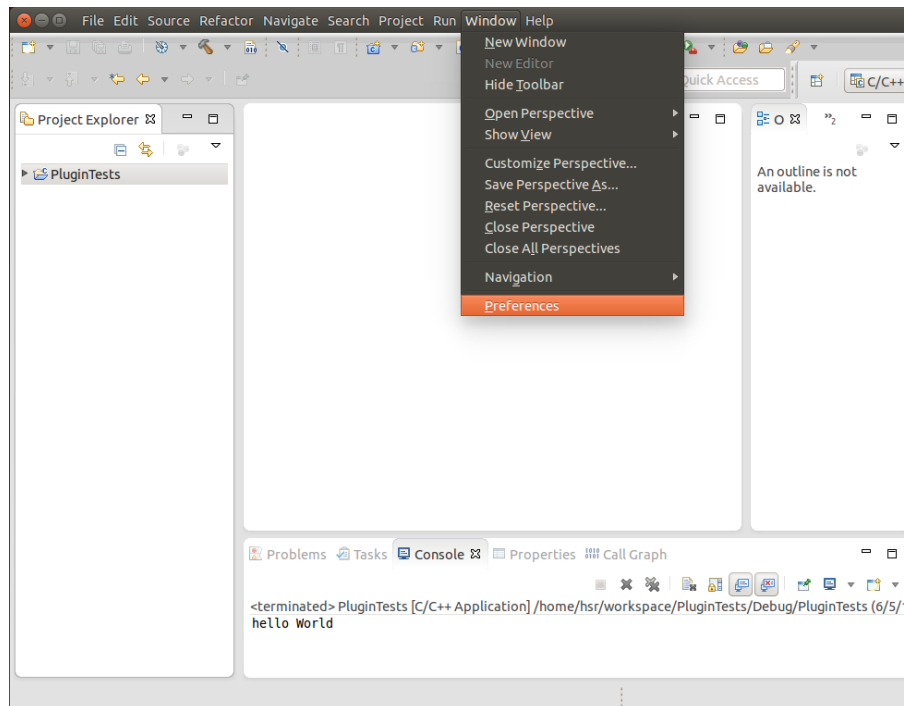


Figure A.5.: Deactivate marker

In the settings window open the section “C/C++” in the left panel. After that you need to press on “Code Analysis”.

This view shows a list with all markers that are set by plug-ins or CDT itself. All problems listed there can be deactivated and reactivated individually. The markers of the CharWars plug-in are activated by default. So there is no need to activate them when you use the plug-in for the first time.

A. User manual

The surrounded four problems that can be found in Figure A.6 are the ones that come from the CharWars plug-in. To deactivate one of these problems one just needs to uncheck the corresponding checkbox. To reactivate a deactivated problem one just needs to check the checkbox again. By clicking “Apply” and then “OK” the settings are saved.

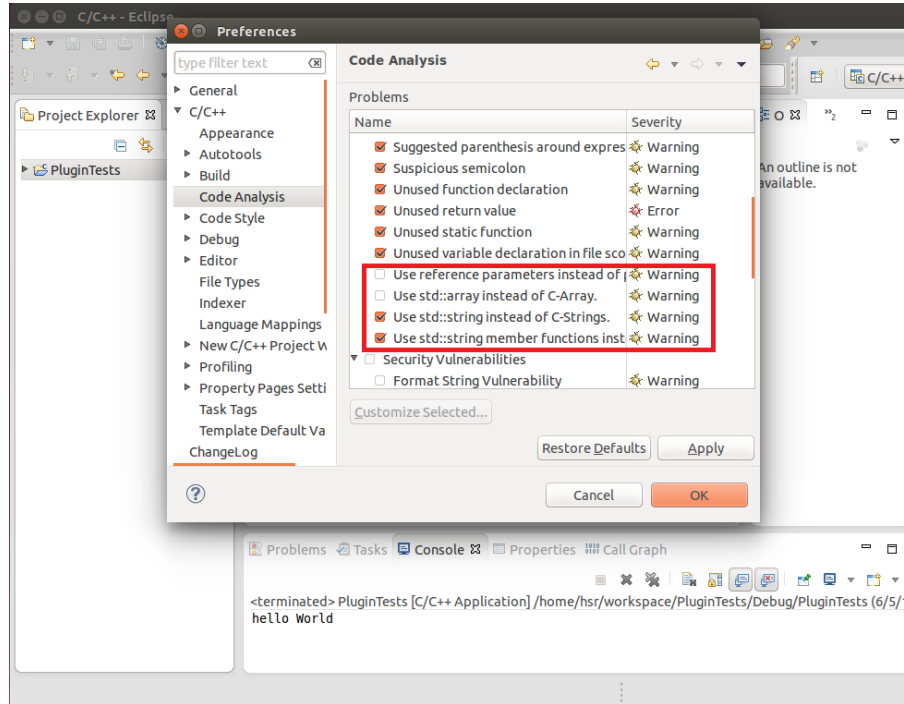


Figure A.6.: Deactivate marker

A. User manual

A.3. De-installation

To de-install the plug-in the following steps need to be performed:

First press on “Help” and select “About Eclipse”.

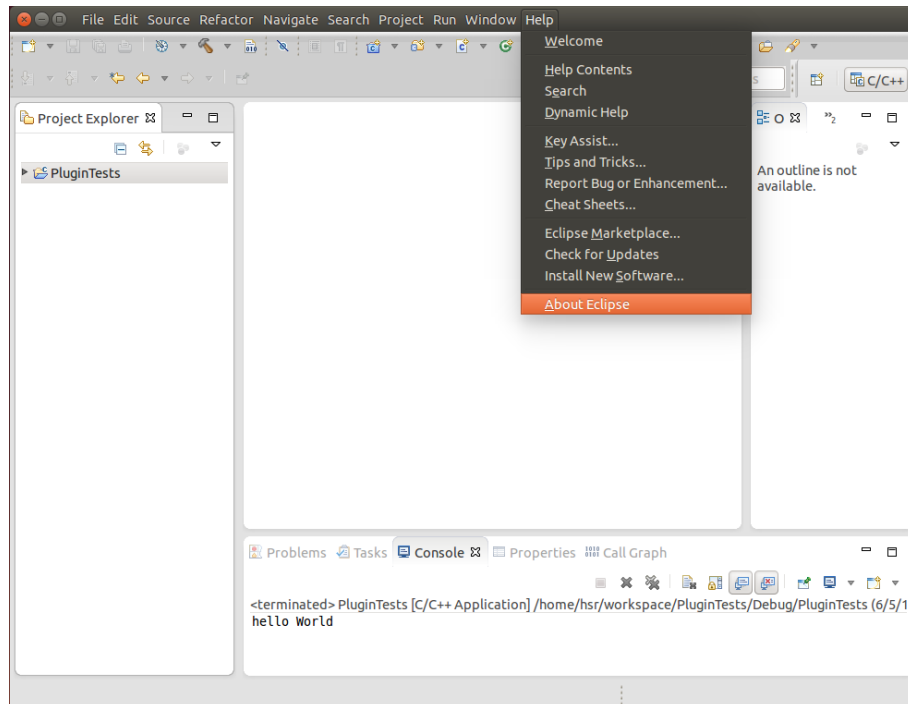


Figure A.7.: De-install plug-in

A. User manual

In the newly opened window press on “Installation Details” to open the details about the current Eclipse installation.

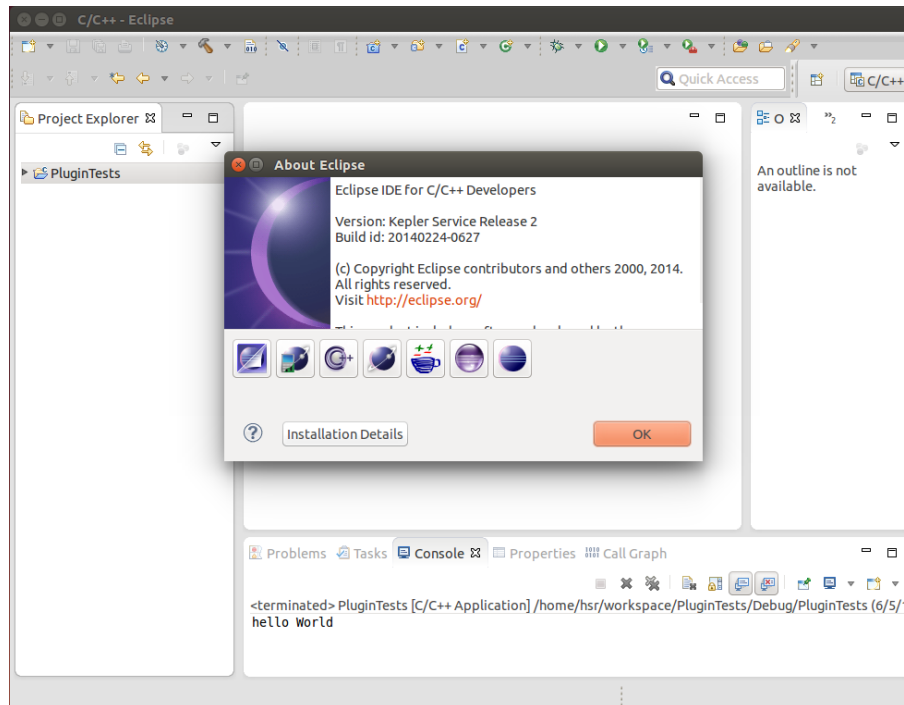


Figure A.8.: De-install plug-in

A. User manual

Under “Installed Software” in the “Installation Details” window all installed plug-ins are shown. Select the CharWars plug-in and then press the “Uninstall...” button. For more information see Figure A.9. Navigate with the “Next” button through the de-installation wizard and finish the de-installation.

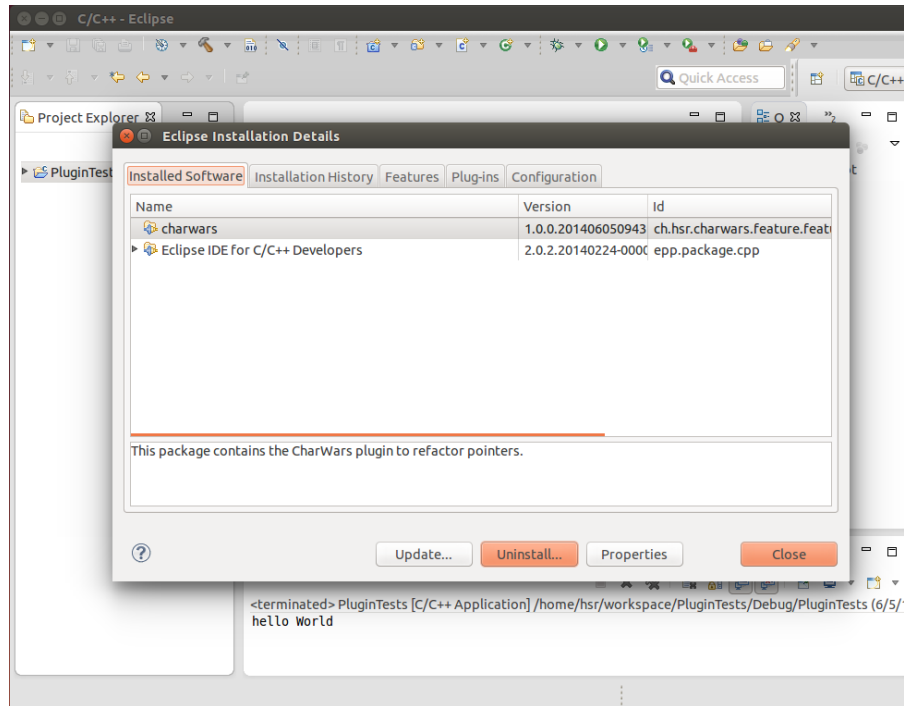


Figure A.9.: De-install plug-in

Bibliography

- [AST14] Class ASTRewrite. Class astrewrite, July 2014. <https://www.cct.lsu.edu/~rguidry/eclipse-doc36/org/eclipse/cdt/core/dom/rewrite/ASTRewrite.html>.
- [cdt14] cdttesting. ch.hsr.ifs.cdttesting, July 2014. <https://github.com/IFS-HSR/ch.hsr.ifs.cdttesting>.
- [fC14] Static Analysis for CDT. Static analysis for cdt, July 2014. <https://wiki.eclipse.org/CDT/designs/StaticAnalysis>.
- [Fel14] L. Felber. *Howto Develop CDT Refactorings*. 2014.
- [Fin14] FindBugs. Findbugs - find bugs in java programs, July 2014. <http://findbugs.sourceforge.net>.
- [Gam94] R. Helm R. Johnson J. Vlissides E. Gamma. *Design Patterns - Elements of Reusable Object-Oriented Software*. 1994.
- [Git14a] Git. Git, July 2014. <http://git-scm.com>.
- [Git14b] HSR Git. Scm manager, July 2014. <https://git.hsr.ch>.
- [Git14c] GitHub. Github, July 2014. <https://github.com/>.
- [Gon13] T. Suter F. Gonzalez. *Pointterminator*. 2013.
- [Jen14] Jenkins. Jenkins ci, July 2014. <http://jenkins-ci.org>.
- [oP14] Overview of Parsing. Overview of parsing, July 2014. http://wiki.eclipse.org/CDT/designs/Overview_of_Parsing.
- [Pro14a] Apache Maven Project. Maven - welcome to apache maven, July 2014. <http://maven.apache.org>.

Bibliography

- [Pro14b] The WebKit Open Source Project. Environmentutilities.cpp, March 2014. <https://github.com/WebKit/webkit/blob/e7207313fed4b7a2140c39f65d45e0f441731735/Source/WebKit2/Platform/unix/EnvironmentUtilities.cpp>.
- [Red14] Redmine. Overview - redmine, July 2014. <http://www.redmine.org>.
- [Spo14] Joel Spolsky. Back to basics, July 2014. <http://www.joelonsoftware.com/articles/fog0000000319.html>.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language*. 1997.
- [xG14] xbmc/xbmc GitHub. Xbmc main repository, May 2014. <https://github.com/xbmc/xbmc>.