

TERM PROJECT, FALL TERM 2013

POINTERMINATOR

Rid C++ code of unnecessary pointers



IFS

INSTITUTE FOR
SOFTWARE

AUTHORS

Toni Suter & Fabian Gonzalez

SUPERVISOR

Prof. Peter Sommerlad



HSR

HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Term Project

Pointerminator

Rid C++ code of unnecessary pointers

Fabian Gonzalez, Toni Suter

Fall Term 2013

Supervised by Prof. Peter Sommerlad

Abstract

Pointers and Arrays as inherited from C are still in heavy use in C++ programs. They are used to represent strings, arrays, objects on the heap or they appear in function signatures to do call-by-reference. However, issues like resource responsibility, degradation of an array to a pointer losing its dimension or zero-termination of byte sequences lead to poor quality and potential security problems.

Modern C++ and its standard library provide a lot of functionality to avoid the use of raw pointers and arrays. If those concepts are applied correctly, they can lead to much better and more maintainable code. The goal of our term project is to write a plug-in for Eclipse CDT that allows a programmer to find and automatically refactor pieces of code, that use pointers in an unfavorable way.

We started with an analysis of the various roles pointers can have. Based on that analysis we decided that the plug-in should be able to refactor C strings, C arrays and pointer parameters. Then we implemented the plug-in and documented its architecture. Finally, we tested the plug-in in the code base of an existing C++ application called fish shell. 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 term project contains an analysis of the various ways pointers can be used in C++. Additionally, it describes the development of an Eclipse CDT plug-in that refactors and replaces pointers automatically to improve the quality of C++ code.

Motivation

In C and C++, pointer variables are used to refer to a specific location in memory. Pointers can be used to point to various things such as strings, arrays, functions or objects on the heap. While this makes them very powerful and flexible, extensive use of pointers leads to unreadable, inefficient and often unsafe code.

Modern C++ and its standard library provide a lot of functionality, that can be used instead of pointers. For example, objects of the classes `std::string` and `std::array` can be used instead of C strings and C arrays. Pointer parameters can often be replaced with reference parameters. However, programmers often don't take advantage of these solutions, because they don't know about the drawbacks of pointers or because they work with an existing code base, that already uses pointers heavily.

Goal

The main goal of this term project is to build a tool for programmers to improve existing C++ code by refactoring unnecessary pointers automatically. We first study the various use cases for pointers and define possible refactorings. To get the best results, we then analyse the different problems and edge cases for each refactoring. This analysis is very important, because the tool will only be used in practice, if it offers a certain degree of reliability.

In the implementation phase we develop the tool in the form of an Eclipse CDT plug-in written in Java. Finally, the plug-in is tested with an existing C++ code base. This helps us to find problems and optimize the refactorings.

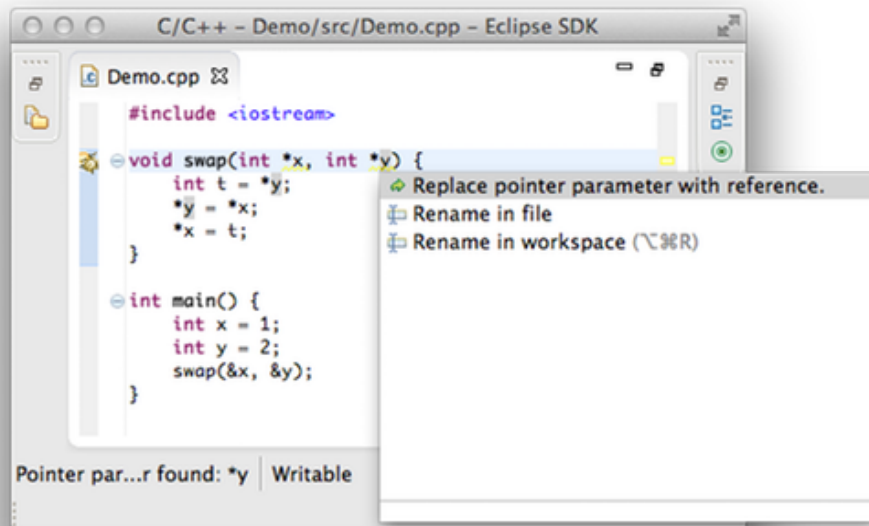
Results

The results of our term project can roughly be divided into three parts. First, we analysed the different use cases of pointers. Based on those use cases we decided to put the focus of our work on the following three areas:

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

In the second phase we implemented an Eclipse plug-in that allows a programmer to automatically refactor C++ code by replacing pointers with a more modern solution. The Pointerterminator 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 activates the automatic refactoring of the plug-in. The following page shows screen shots of the Pointerterminator plug-in in action:

Refactoring pointer parameters



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

```
#include <iostream>

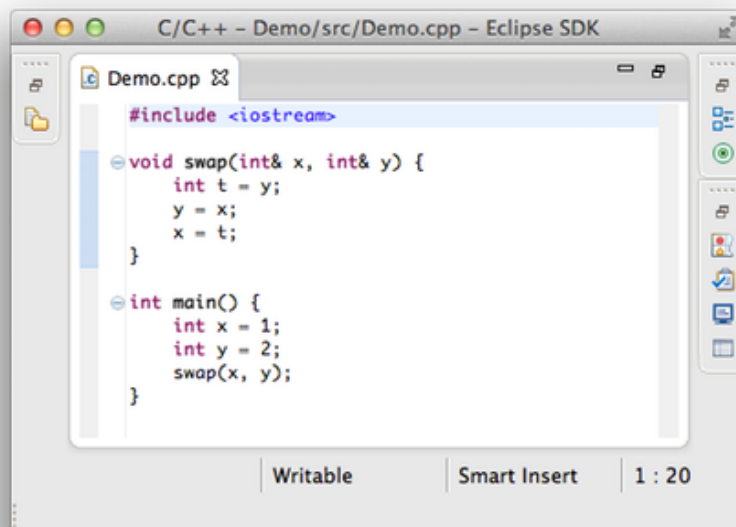
void swap(int *x, int *y) {
    int t = *y;
    *y = *x;
    *x = t;
}

int main() {
    int x = 1;
    int y = 2;
    swap(&x, &y);
}
```

The `*y` parameter in the `swap` function is selected, and a context menu is open with the following options:

- Replace pointer parameter with reference.
- Rename in file
- Rename in workspace (⌘R)

The status bar at the bottom indicates "Pointer par...r found: *y | Writable".



The screenshot shows the same Eclipse IDE window after refactoring. The code is now:

```
#include <iostream>

void swap(int& x, int& y) {
    int t = y;
    y = x;
    x = t;
}

int main() {
    int x = 1;
    int y = 2;
    swap(x, y);
}
```

The status bar at the bottom now shows "Writable | Smart Insert | 1 : 20".

Finally, to optimize the plug-in, we tested it with an existing open source C++ project called fish shell [7]. The following table shows the final statistic of the test results:

Refactoring	Markers set	Solved	Unsolved
std::string	18	9 (50%)	9 (50%)
std::array	154	114 (74%)	40 (26%)
pointer parameter	662	51 (42.5%)	69 (57.5%)

Further work

The Pointerterminator plug-in provides useful functionality but there is still room for improvement. Further optimization would be worthwhile. Also, there are other refactorings that could be added in addition to the existing ones such as:

- Refactor pointer return values
- Refactor function pointers
- Refactor pointers to objects on the heap
- Refactor pointers to dynamic strings

Contents

1	Task description	4
1.1	Problem	4
1.2	Solution	4
1.2.1	C strings	5
1.2.2	C arrays	6
1.2.3	Call-by-reference / Out-parameters	7
1.2.4	Pointers as return values	7
1.2.5	Pointers to dynamically allocated memory	7
1.2.6	Function pointers	8
1.3	Our goals	8
1.3.1	C strings to std::string	9
1.3.2	C arrays to std::array	9
1.3.3	Pointer as parameters	9
1.3.4	Additional refactorings	10
1.4	Time management	10
1.5	Final release	10
2	Analysis	11
2.1	The Pointerminator refactoring rules	11
2.2	Refactorings	12
2.2.1	Replace C string with std::string	13
2.2.2	Replace C array with std::array	16
2.2.3	Replace pointer parameter with reference	18
3	Implementation	22
3.1	Tools and technologies	22
3.1.1	Parser and Abstract Syntax Tree (AST)	22
3.1.2	Bindings	24
3.1.3	The index	25
3.2	Techniques and algorithms	26
3.2.1	The plug-in components	26
3.2.2	Traversing the AST	27

Contents

3.2.3	Modifying and Rewriting the AST	28
3.2.4	Testing	28
3.2.5	Searching across multiple files	30
3.2.6	Dealing with global variables	31
3.2.7	The <code>std::string</code> refactoring	31
3.2.8	The <code>std::array</code> refactoring	34
3.2.9	The pointer parameter refactoring	36
4	Test refactoring against real life code	37
4.1	Statistics	37
4.1.1	Statistics of week 11	38
4.1.2	Statistics of week 13	38
4.2	Examples for <code>std::string</code> refactoring	38
4.2.1	Standard <code>wchar_t*</code> variable	39
4.2.2	<code>char</code> string variable with size definition	39
4.3	Examples for <code>std::array</code> refactoring	40
4.3.1	Standard C array variables	40
4.3.2	C array with method calls	42
4.4	Examples for pointer parameter refactoring	43
4.4.1	Pointer parameter refactoring in single file	44
4.4.2	Pointer parameter refactoring in multiple files	45
5	Conclusion	46
5.1	Achievements	46
5.2	Future Work	46

1 Task description

This section contains the description of our project and our goals for it.

1.1 Problem

Raw pointers in C++ are one of C's legacies and can be used for various purposes. A pointer can point to a function, an array, a C string, a variable on the stack, dynamically allocated memory on the heap or NULL. Programmers often use pointers in function signatures to do call-by-reference, have other out-parameters in addition to the normal return value and to pass function pointers that point to callback functions.

Although pointers are a very powerful means to write software, the resulting code often becomes nearly unreadable. Additionally, the use of pointers can lead to some very unsafe code. For example, arrays can be accessed out of bounds or invalid pointers (e.g., a dangling pointer or a NULL pointer) can be dereferenced. Those bugs are not caught by the compiler but lead to undefined behaviour during runtime. The biggest issue when pointing to dynamically allocated memory is memory management. Careless programming can quickly lead to memory leaks or dangling pointers. Often it's also hard to quickly figure out who owns a certain resource and is therefore responsible for releasing it.

1.2 Solution

Modern C++ offers a lot of new classes and functionality which can be used to replace raw pointers. This section describes how code with raw pointers can be improved.

1 Task description

1.2.1 C strings

C strings should be changed to objects of the class `std::string`. This class has methods to do easy iteration, get the size and modify the string. That's a much cleaner and more object oriented way of dealing with strings. Listings 1.1 and 1.2 serve as an example.

Listing 1.1: Before refactoring

```
int main() {
    char str[] = "Hello";
    str[1] = 'a';
    std::cout << "Size = "
                << strlen(str)
                << std::endl;

    for(
        int i = 0;
        i < strlen(str);
        ++i
    ) {
        std::cout << str[i];
        std::cout << std::endl;
    }
}
```

Listing 1.2: After refactoring

```
int main() {
    std::string str = "Hello";
    str[1] = 'a';
    std::cout << "Size = "
                << str.size()
                << std::endl;

    for(
        auto it = str.cbegin();
        it != str.cend();
        ++it
    ) {
        std::cout << *it;
        std::cout << std::endl;
    }
}
```

1 Task description

1.2.2 C arrays

C arrays can be replaced with abstract data types such as `std::array`. These types provide better safety because they offer iterators and access methods, that throw exceptions when someone tries to access the array at an index that is out of bounds. When passed to a function, conventional arrays often require an additional parameter that specifies the length of the array, since this information is not stored in the array itself. Using templates, C++11 can deduce the size of a `std::array` at compile time. Listings 1.3 and 1.4 show a possible refactoring.

Listing 1.3: Before refactoring

```
void print_array(int arr [],
                int length) {

    std::cout << "[ ";
    for(
        int i = 0;
        i < length;
        ++i
    ) {
        std::cout << arr[i];
        std::cout << " ";
    }
    std::cout << "]" ;
    std::cout << std::endl;
}

int main() {
    int arr[] =
    {1, 2, 3, 4};
    arr[2] = 8;
    print_array(arr, 4);
}
```

Listing 1.4: After refactoring

```
template<std::size_t SIZE>
void print_array(std::array<
    int, SIZE> &arr){
    std::cout << "[ ";
    for(
        auto it = arr.cbegin();
        it != arr.cend();
        ++it
    ) {
        std::cout << *it;
        std::cout << " ";
    }
    std::cout << "]" ;
    std::cout << std::endl;
}

int main() {
    std::array<int, 4> arr =
    { {1, 2, 3, 4} };
    arr[2] = 8;
    print_array(arr);
}
```

1 Task description

1.2.3 Call-by-reference / Out-parameters

Instead of using raw pointers to do call-by-reference, C++ references should be considered. References provide more safety, since the address they point to, can not be directly manipulated or reassigned. It's also possible to define constant references, in order to pass large objects or data structures to a function without having to do an expensive copy. Listings 1.5 and 1.6 show a possible refactoring.

Listing 1.5: Before refactoring

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}

int main() {
    int a = 1;
    int b = 2;
    swap(&a, &b);
}
```

Listing 1.6: After refactoring

```
void swap(int &x, int &y) {
    int t = x;
    x = y;
    y = t;
}

int main() {
    int a = 1;
    int b = 2;
    swap(a, b);
}
```

1.2.4 Pointers as return values

Factory functions often return raw pointers to the newly allocated objects. The caller is therefore the owner of this memory and is responsible for deallocating it. In such cases raw pointers should be replaced by a smart pointer such as `std::unique_ptr`. This makes it more obvious who is the owner of the object.

1.2.5 Pointers to dynamically allocated memory

To avoid memory management issues, smart pointers such as `std::unique_ptr` or `std::shared_ptr` should be used. Smart pointers can take on the responsibility to release the memory once it is no longer used. They can also make the code more readable because it becomes clear which pointers owns a resource and which doesn't.

1 Task description

1.2.6 Function pointers

Function pointers can be used to pass callbacks to a function. However, it is better to use a template parameter instead, because this allows to not only pass normal functions, but also lambdas or `std::function` variables as arguments. Listings 1.7 and 1.8 show an example.

Listing 1.7: Before refactoring

```
void print(int n) {
    std::cout << n;
    std::cout << std::endl;
}

void load(void (*cb)(int)) {
    int i = ... //load number
    cb(i);
}

int main() {
    load(&print);
}
```

Listing 1.8: After refactoring

```
void print(int n) {
    std::cout << n;
    std::cout << std::endl;
}

template<typename F>
void load(F cb) {
    int i = ... //load number
    cb(i);
}

int main() {
    load(print);
}
```

1.3 Our goals

In our term project we will first look at common situations in which raw pointers are unnecessarily used in C++ code bases. We then try to define refactorings that improve the code by replacing the pointers with a more modern solution. After that we implement an Eclipse CDT plug-in that suggests an appropriate refactoring where necessary and lets the developer apply that refactoring automatically. In the final stage we test the plug-in with a well-known C++ open source project and try to optimize its heuristics and functions as much as possible. Since there is not enough time for us to do everything, we will concentrate on the following three areas:

- C strings
- C arrays
- Pointers as parameters (Call-by-reference, Out-parameters)

1 Task description

1.3.1 C strings to std::string

This refactoring will change C strings to objects of the class `std::string`.

Features

- Replace C string definition with `std::string` definition
- Replace calls to `strlen()` with a call to the `size()` member function

Conditions

- C string can not be dynamic (i.e., change its size)

1.3.2 C arrays to std::array

This refactoring will change C arrays to objects of the class `std::array`.

Features

- Replace array definition with `std::array` definition

Conditions

- Refactoring doesn't handle arrays defined as function parameters
- Arrays can not be dynamic (i.e., change its size)

1.3.3 Pointer as parameters

This refactoring will replace pointer parameters with reference parameters.

Features

- Replace pointer parameters with references in declaration and definition of the function
- Find calls of the function and adapt them accordingly
- Inside the function, replace the dereference-operators

Conditions

- Parameters can't be optional (i.e., can't be null)
- Pointer can't be repointed inside function body

1 Task description

1.3.4 Additional refactorings

If everything has been done and we have enough time, we will extend our plug-in to support more generic cases. Here are some additional refactorings that could be implemented:

- Replace dynamic strings with `std::string`
- Replace dynamic arrays with `std::vector`
- Replace function pointers with template parameter
- Replace pointers as return values (e.g., in factory function) with smart pointer

1.4 Time management

Our project started on the 16th of September, 2013. It will take 14 weeks and end on December the 20th, 2013 at 12:00 p.m. which is when the final release has to be submitted completely.

1.5 Final release

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

- 2 printed exemplars of the documentation
- Poster for presentation
- Management Summary and Abstract
- CD/DVD with update site that contains the plugin, project resources, documentation

2 Analysis

In this section, we analyze each refactoring and try to find problems and challenges that could complicate its development. Later, in the Implementation section, we will describe how those problems have been solved. The plug-in will only be useful if the programmers trust it enough, to actually apply the refactorings in a real code base. Therefore, we have to define rules according to which the refactorings will work and look at all the edge cases that can occur in day-to-day use.

2.1 The Pointerterminator refactoring rules

Refactoring means to improve the design of existing code without changing its behaviour. Therefore, the Pointerterminator plug-in will adhere to the following rules:

Lifetime

The lifetime of a variable should not be extended by a refactoring.

Visibility

The visibility of a variable should not be extended by a refactoring.

Memory location

Objects that are allocated on the heap may be moved to the stack by the refactoring if it makes sense in this particular context.

Constness

Generally, the constness of a variable should be preserved by the refactoring. If a variable is defined to be non-const before the refactoring but could also be defined const, multiple quick fixes may be proposed. For example:

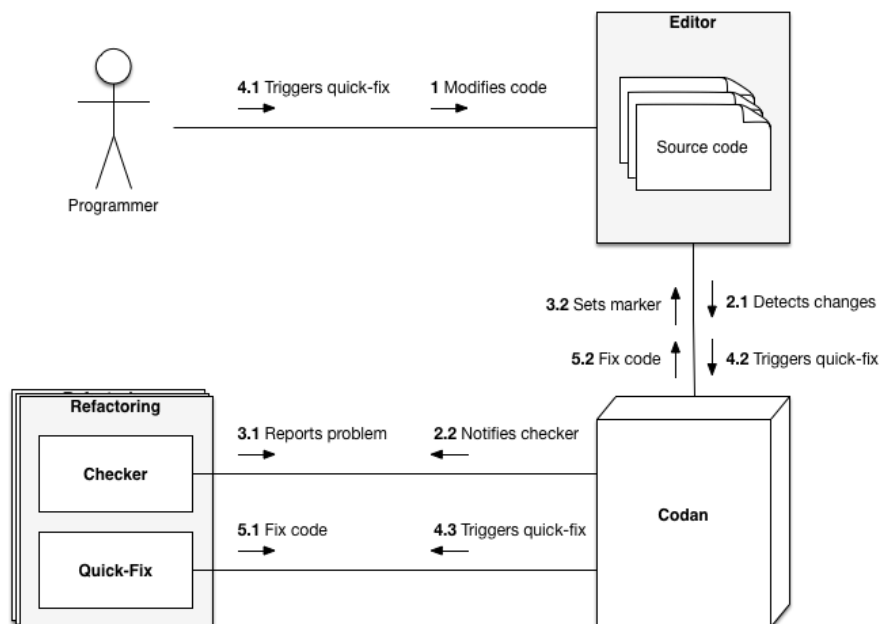
- Replace pointer parameter with reference
- Replace pointer parameter with const reference

2.2 Refactorings

To implement its functionality, the Pointerminator plug-in relies heavily on Codan[1]. 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.

As shown in the task description, the Pointerminator plug-in consists of 3 refactorings. Each refactoring, in turn, consists of a checker and a quick-fix. Figure 2.1 illustrates the typical refactoring cycle.

Figure 2.1: Refactoring cycle



1. The programmer modifies the source code.
2. Codan 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.

2 Analysis

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.

The following section describes possible problems and challenges for each refactoring:

2.2.1 Replace C string with std::string

In C, strings are just char pointers to a '\0'-terminated array of characters. Those strings are often used in C++ programs even though they are inefficient and unsafe. For example, if someone wants to find out the length of a string in C, the person normally calls the `strlen()`-function. Since the string does not store its own size, this function then has to count the characters until it reaches '\0'. That is inefficient ($O(n)$). The string refactoring of the Pointerterminator plug-in can automatically replace a C string with an object of the more efficient and safer class `std::string`. Objects of this class store the size of the string ($O(1)$).

Problem 1: Recognizing C strings

In order to set the markers, the checker of the string refactoring has to be able to identify C strings. This can be difficult under certain circumstances. Consider the possibilities in listing 2.1.

Listing 2.1: Possible cases

```
1: char *x; //we don't know
2: char *str1 = "Hello"; //set a marker
3: char str2[] = "World"; //set a marker
4: char a = 'a'; //no marker
5: char *ptr = &a; //we don't know
6: char arr[] = {'a', 'b'}; //no marker
```

Case 1:

It could be that `x` eventually will point to a C string. However, `x` could also be pointed to a single character or to an array of characters. To be sure, what `x` is used for, further static analysis would be necessary.

2 Analysis

Case 2:

In this case, a marker can be set, because the char pointer points to a C string.

Case 3:

Here a marker can be set as well, because a ‘\0’-terminated array of characters is effectively a C string.

Case 4:

The variable “a” is a single character (not a pointer). It can’t be refactored with the string refactoring of the Pointerterminator plug-in.

Case 5:

The char pointer “ptr” points to the variable “a”. In order to find out whether this variable is a C string, further static analysis would be necessary.

Case 6:

In this case, “arr” is an array of characters. Because it is not ‘\0’-terminated, this is not the same as a C string. Therefore, no marker can be set.

Problem 2: Multiple declarations in a single statement

In C++ it is possible to declare multiple variables in a single statement. This can cause some complications. For example, consider the following code in listing 2.2.

Listing 2.2: Multiple variable declarations

```
char *x = "Hello", y[] = "World", z = 42;
```

While x and y are both strings, z is just a single char. This means that the checker should only mark single variable names and not complete statements, because there could be additional variable declarations in the same statement, that can not be refactored.

2 Analysis

Problem 3: Variable shadowing

After a C string definition has been replaced with a `std::string` definition, the string refactoring quick-fix needs to find all subsequent occurrences of the string variable in order to do additional modifications if necessary. One approach could be to search for variables with the same name. However, consider the code in listing 2.3.

Listing 2.3: Variable shadowing

```
int main() {
    char str[] = "Hello!";
    std::cout << str << std::endl;    // -> Hello!

    for(int i = 0; i < 5; ++i) {
        std::cout << str << std::endl;    // -> Hello!
        char str[] = "Goodbye!";
        std::cout << str << std::endl;    // -> Goodbye!
    }
    std::cout << str << std::endl;    // -> Hello!
}
```

If the refactoring was applied to the “Hello!”-string, this approach would cause problems, due to the “Goodbye!”-string in the nested for-loop. Because both string variables have the same name, the “Goodbye!”-string “hides” the outer “Hello!”-string. This is called variable shadowing [2]. Therefore, a simple comparison of the variable names is not enough to guarantee that only occurrences of the same variable are found by the string refactoring.

Problem 4: Include handling

The use of the `std::string` datatype requires that the string header be included with a preprocessor directive. If the string header is not already included, the string refactoring has to take care of that too.

Problem 5: Memory management

When a char array is initialized with a string literal, it is possible to specify an array size that is bigger than the string size. To ensure that the refactored program behaves exactly the same, the string refactoring has to reserve the same amount of memory for the `std::string`. This

2 Analysis

can be done with a call to the `reserve()`-member function. Listings 2.4 and 2.5 show an example.

Listing 2.4: Before refactoring

```
int main() {
    char s[42] = "Hello";
}
```

Listing 2.5: After refactoring

```
int main() {
    std::string s = "Hello";
    s.reserve(42);
}
```

2.2.2 Replace C array with `std::array`

When an array is defined in C, the required space is allocated in memory and initialized with the contents of the array. The array variable itself is just a pointer to the first element in the array. The size of the array is not stored anywhere. This means that the programmer needs to keep track of the size of the array. Such arrays are often used in C++ programs but they can lead to inefficient or unsafe code. For example, if an array is passed to a function as a parameter, there needs to be a second parameter, that specifies the size of the array. The array refactoring of the Pointerminator plug-in can automatically replace a C array with an object of the more efficient and safer class `std::array`.

Problem 1: Recognizing C arrays

In order to set the markers, the checker of the array refactoring has to be able to identify C arrays. Therefore, it has to differentiate between the following cases shown in listing 2.6:

Listing 2.6: Possible cases

```
1: int numbers[] = {1, 2, 3, 4, 5}; //set a marker
2: char arr[] = {'a', 'b'}; //set a marker
3: char str[] = "Hello!"; //no marker
```

Case 1:

In this case, a marker can be set, because the array “numbers” is initialized with an array of integers.

2 Analysis

Case 2:

The array “arr” is initialized with an array of characters, which is not the same as a string. Therefore, a marker can be set.

Case 3:

The char array “str” is initialized with a string literal. In this case, the marker should be set by the checker of the string refactoring.

Problem 2: Multidimensional arrays

The array refactoring should also be able to handle multidimensional arrays. For that, the quick-fix needs to create a `std::array` of `std::arrays`. The nested `std::array` specifies the type of the elements in the multidimensional array. Listings 2.7 and 2.8 show an example refactoring.

Listing 2.7: Before refactoring

```
int b[2][2] =  
    {{1, 2}, {3, 4}};
```

Listing 2.8: After refactoring

```
std::array<  
std::array<int, 2>, 2> b =  
    {{1, 2}, {3, 4}};
```

Problem 3: Arrays as arguments

After a C array definition has been replaced with a `std::array` definition, the array refactoring quick-fix needs to find all subsequent occurrences of the array variable in order to do additional modifications if necessary. Often, no modification is required. For example, array subscript expressions such as “arr[5] = 2;” don’t have to be altered, because the `std::array` class implements the corresponding operator overloads. But if an array is passed to a function as an argument, the array refactoring quick-fix needs to adapt the argument by calling the `std::array::data` member function. This is shown in listings 2.9 and 2.10.

2 Analysis

Listing 2.9: Before refactoring

```
int main() {
    int a[] =
    {1, 2, 3};
    a[0] = 4;
    printArray(a, 3);
}
```

Listing 2.10: After refactoring

```
int main() {
    std::array<int, 3> a =
    {{1, 2, 3}};
    a[0] = 4;
    printArray(a.data(), 3);
}
```

A better solution would be to adapt the signature of the “printArray()” function so that it only takes one `std::array` parameter. However, this is outside the scope of this project.

Problem 4: Include handling

The use of the `std::array` datatype requires that the array header be included with a preprocessor directive. If the array header is not already included, the array refactoring has to take care of that too.

2.2.3 Replace pointer parameter with reference

In C and C++, pointer parameters are often used to either do call-by-reference or to pass a large object or data structure to a function without copying it. In both cases it is usually better to use C++ references instead. References provide better safety, since the address they point to can not be directly manipulated or reassigned. Additionally, a reference always points to something (i.e., is never NULL). The pointer parameter refactoring of the Pointerminator plug-in can automatically replace a pointer parameter with a reference parameter.

2 Analysis

Problem 1: Recognizing pointer parameters

In order to set the markers, the checker of the pointer parameter refactoring has to be able to find pointer parameters. However, not all pointer parameters can be changed to reference parameters. Consider the different possibilities in listing 2.11.

Listing 2.11: Possible cases

```
1: void double1(int *x) { //set a marker
    *x = *x * 2;
}
2: void double2(int *x) { //no marker
    if(x) {
        *x = *x * 2;
    }
}
3: void print1(char *str) { //no marker
    std::cout << str << std::endl;
}
4: void print2(int *arr, int len) { //no marker
    for(int i = 0; i < len; ++i) {
        std::cout << arr[i] << std::endl;
    }
}
```

Case 1:

In this case, the pointer parameter can safely be converted into a reference parameter.

Case 2:

The if-statement in the function body checks, whether x is not NULL. This implies that the function can be called with a NULL-pointer. Since references can not be NULL, this parameter can not be converted.

Case 3:

In this case, the function takes a C string parameter. Therefore, it would be more appropriate to apply the string refactoring here.

2 Analysis

Case 4:

This function takes a pointer to an array and a second parameter that specifies the length of the array. Instead of converting the array pointer to a reference, it would be more suitable to replace both parameters with single `std::array` parameter.

Problem 2: Dealing with overloaded functions

In C++, functions can be overloaded. This means that there can be multiple functions with the same name as long as they have a different function signature. Listing 2.12 shows an example.

Listing 2.12: Overloaded functions

```
void print(int i) {
    std::cout << "Integer: " << i << std::endl;
}

void print(double d) {
    std::cout << "Double: " << d << std::endl;
}

void print(char* c) {
    std::cout << "String: " << c << std::endl;
}

int main() {
    print(10);
    print(10.10);
    print("ten");
}
```

When a programmer triggers the pointer parameter refactoring, the pointer parameter quick-fix has to adapt all occurrences of the function (i.e., function definition, function declarations, function calls). Because the function could be overloaded, the quick-fix can not just look for occurrences of a function with the same name. Further analysis is required to differentiate between the various overloaded functions.

2 Analysis

Problem 3: Adapting all occurrences of the function

Normally, a C++ project consists of multiple header and cpp files. This means that the occurrences of a function often will be spread across multiple files. Therefore, the pointer parameter refactoring has to search all files of the project which could have negative impacts on the performance of the refactoring. Listing 2.13 and 2.14 show how a complete refactoring could look like.

Listing 2.13: Before refactoring

```
//mul.h
class Mul {
public:
    Mul();
    void triple(int *x);
};

//mul.cpp
#include "mul.h"

Mul::Mul() {}

void Mul::triple(int* x) {
    *x = *x * 3;
}

//main.cpp
#include "mul.h"

int main() {
    int x = 5;
    int y = 10;
    int *z = &y;

    Mul *m = new Mul();
    m->triple(&x);
    m->triple(z);
}
```

Listing 2.14: After refactoring

```
//mul.h
class Mul {
public:
    Mul();
    void triple(int &x);
};

//mul.cpp
#include "mul.h"

Mul::Mul() {}

void Mul::triple(int& x) {
    x = x * 3;
}

//main.cpp
#include "mul.h"

int main() {
    int x = 5;
    int y = 10;
    int *z = &y;

    Mul *m = new Mul();
    m->triple(x);
    m->triple(*z);
}
```

3 Implementation

In this section we describe how we created our plug-in, how it works and how we handled the problems that we described in the Analysis section.

3.1 Tools and technologies

The following subsections explain some tools and technologies that were used to create and test the Pointerterminator plug-in.

3.1.1 Parser and Abstract Syntax Tree (AST)

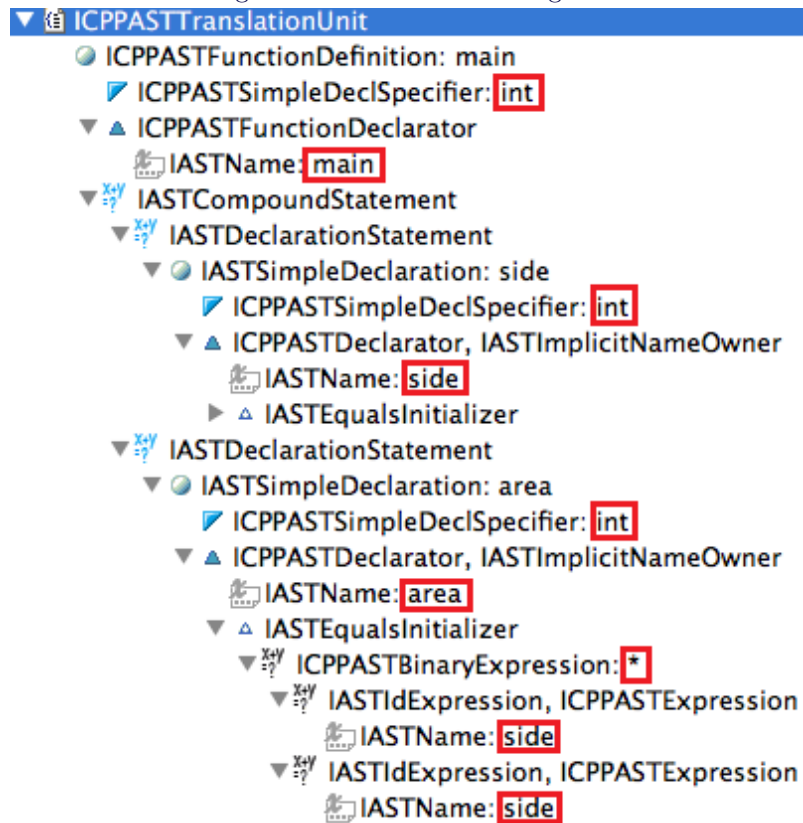
When a cpp-file is opened in an Eclipse CDT editor, the parser creates a tree-representation of the code, which is called the Abstract Syntax Tree (AST). The AST consists of nodes that all implement the IASTNode interface. Each node has one parent node and an array of child nodes. The AST can be used by static analysis tools such as the Pointerterminator 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 3.1 show an example of what the AST looks like for a short program.

Listing 3.1: AST example

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

3 Implementation

Figure 3.1: AST tree of listing 3.1



3 Implementation

3.1.2 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.

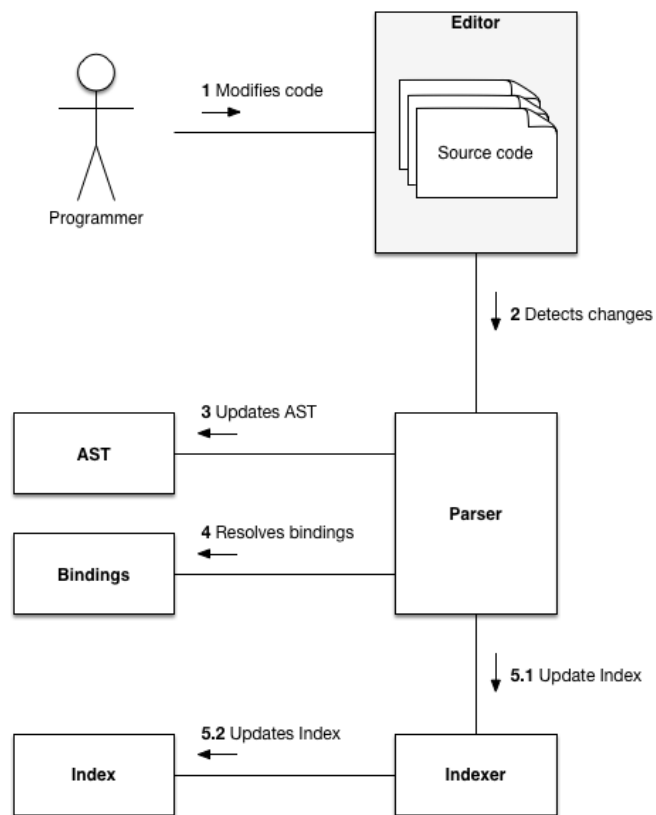
3 Implementation

3.1.3 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 3.2 shows how everything fits together [3].

Figure 3.2: How everything fits together



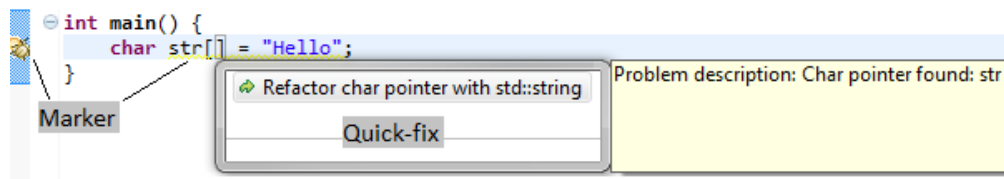
3.2 Techniques and algorithms

The following subsections explain some general techniques as well as specific algorithms that were used to solve the problems described in the section “Analysis”.

3.2.1 The plug-in components

The Pointerminator 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 3.3: 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 Pointerminator 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 Implementation

3.2.2 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** [4]. 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 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;
        }
    }
}
```

3.2.3 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 [5].

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();
}
```

3.2.4 Testing

The Codan framework contains testing infrastructure classes (e.g., `CheckerTestCase`, `QuickFixTestCase`) that provide the basic tools to test a plug-in that does static analysis with Codan. Testing checker classes is pretty easy and involves the following steps:

- Get some test code that the checker should check.
- Load the test code and run Codan.
- Check that there is an error (i.e., a marker set by the checker) on a certain line.

To make the unit tests more readable it is possible to write the test code as a comment on top of the unit test and then dynamically load

3 Implementation

it with the helper function `getAboveComment()`. Listing 3.4 shows a unit test for the string refactoring checker:

Listing 3.4: A unit test for a checker

```
//int main() {
//  const char *str = "Hi";
//}
public void testStringChecker() {
    String testCode = getAboveComment();
    loadCodeAndRunCpp(testCode);
    checkErrorLine(2);
}
```

Testing quick-fix classes is bit more complicated and involves the following steps:

- Get some test code that contains a problem to be fixed by the quick-fix.
- Load the test code.
- Run the quick-fix and get the resulting code.
- Compare the resulting code with the expected code.

Because the resulting code string that is returned from the method `runQuickFixOneFile()` does not always have a consistent format, it is difficult to compare the entire string. Therefore, there's a method `assertContainedIn()` that checks whether a certain code snippet exists in the result.

Listing 3.5 shows a unit test for the string refactoring quick-fix:

Listing 3.5: A unit test for a quick-fix

```
//int main() {
//  const char *str = "Hi";
//}
public void testStringQuickfix() throws Exception {
    String testCode = getAboveComment();
    loadcode(testCode);
    String r = runQuickFixOneFile();
    assertContainedIn("std::string str = \"Hi\";", r);
}
```

3 Implementation

We had problems with running the tests on the build server. Sometimes the refactoring just doesn't run during the test what led to a failing test. Even when we didn't change the code some tests just failed randomly. This seems to happen due to race conditions in the Codan testing infrastructure.

3.2.5 Searching across multiple files

Sometimes it is necessary to be able to search across multiple files. For example, for the pointer parameter refactoring the Pointerterminator plug-in needs to find all calls of the function in order to adjust the corresponding argument. Since each file has its own AST, the visitor-approach described above is not well-suited. A better solution is to search for occurrences of the function using the index provided by Eclipse CDT. Listing 3.6 contains an example.

Listing 3.6: Using the index to search across files

```
IBinding b = declarator.getName().resolveBinding();
IIndexName[] in;
in = index.findNames(b, IIndex.FIND_ALL_OCCURRENCES);

for(IIndexName n : in) {
    ITranslationUnit tu;
    ASTRewrite rewrite;
    IASTTranslationUnit ast;
    IASTNode node;

    tu=CxxAstUtils.getTranslationUnitFromIndexName(n);
    ast = tu.getAST(
        index,
        ITranslationUnit.AST_SKIP_INDEXED_HEADERS
    );
    rewrite = ASTRewrite.create(ast);

    int o = n.getNodeOffset();
    int l = n.getNodeLength();
    node = ast.getNodeSelector(null).findNode(o, l);

    //modify and rewrite AST
}
```

3 Implementation

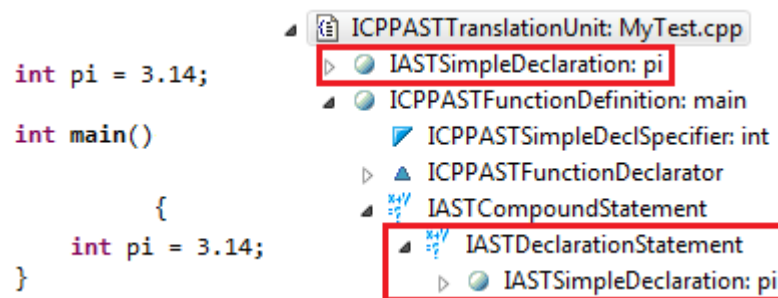
Since the index uses bindings to search for the given IASTName, a few other potential problems are solved as well. For example, overloaded functions won't be found by the index, because they have a different binding object.

3.2.6 Dealing with global variables

The string and the array refactoring both have 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 3.4 for an example.

Figure 3.4: AST structure - Global vs. local variable



3.2.7 The std::string refactoring

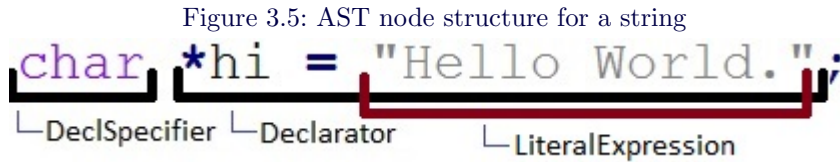
The following subsection describes the algorithms that were used to solve the problems related to the std::string refactoring.

Problem 1: Recognizing C strings

Figure 3.5 shows the AST node structure of a C string definition. The names for the nodes (DeclSpecifier, Declarator, etc.) are defined in

3 Implementation

the C++ standard [6] and Codan uses those names in the interfaces that are implemented by the nodes in the AST.



To determine whether a variable is a string or not, the `std::string` refactoring has to inspect the AST node structure and check if all of the following conditions are met:

1. DeclSpecifier contains `char`, `wchar_t`, `char32_t` or `char16_t`.
2. Declarator contains `*` or `[]` sign.
3. The right side of the assignment is a `LiteralExpression`.

Problem 2: Multiple declaration handling

As described in the problem description, multiple variables can be declared within a single declaration statement. Therefore, the checker of the string refactoring should mark individual declarator nodes instead of the whole declaration statement node. This allows the programmer to refactor variables individually. Also, it is important to maintain the order in which the variables are declared, because there could be dependencies between them. To do that, the refactoring needs to take apart the declaration statement and put each declarator in its own statement. Listings 3.7 and 3.8 show an example.

Listing 3.7: Before refactoring

```
int main() {
    char c = 'a',
          *str = "hello",
          i = 12;
}
```

Listing 3.8: After refactoring

```
int main() {
    char c = 'a';
    std::string str = "hello";
    char i = 12;
}
```

If the declarator is longer than one line there are some graphical glitches, so the whole line will be marked. This is because the size of the marker is not correctly calculated by Codan.

3 Implementation

Problem 3: Variable shadowing

The variable shadowing problem can be solved by using the index to search for subsequent occurrences of a variable. Because each variable has its own distinct binding object, the index only finds occurrences of that exact variable. The sections “3.1.2 Bindings” and “3.2.5 Searching across multiple files” contain more information on this topic.

Problem 4: Include handling

The `std::string` class is defined in the `string` header of the C++ standard library. This means, that the string refactoring of the Pointerminator plug-in has to include the `string` header automatically if it isn't already included. However, preprocessor statements can not be added to the AST with an `ASTRewrite`. Therefore, the algorithm needs to add the include statement as plain text to the document. The structure of the preprocessor statements in the AST is flat so to place the new include correctly it is needed to create a tree out of the statement. As this require future analysis of the code it's not implemented. There are cases where the include statement is misplaced and need to be manually placed correctly. Listing 3.9 shows how this can be done in code.

Listing 3.9: How to add an `#include` statement

```
String includeText = "\n#include <string>\n";
//location after the last include statement
int lastIncludeStatement = ...;

try {
    IDocument doc = getDocument();
    doc.replace(lastIncludeStatement, 0, includeText);
}
catch(BadLocationException e) {
    e.printStackTrace();
}
```

Problem 5: Memory management

To ensure that the refactored program behaves the same as before, the `std::string` refactoring needs to reserve the same amount of memory for the string. This will be done with the `reserve()` member function (See

3 Implementation

Analysis). It is called directly after the declaration of the `std::string` variable.

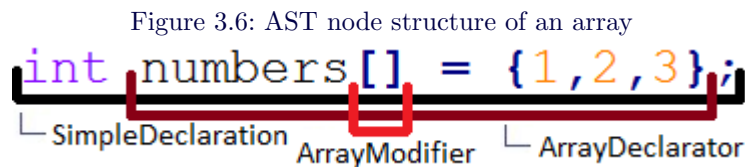
If the variable is global, the `reserve()` member function can not be called, because no code can run in global scope.

3.2.8 The `std::array` refactoring

The following subsection describes the algorithms that were used to solve the problems related to the `std::array` refactoring.

Problem 1: Recognizing C arrays

Figure 3.6 shows the AST node structure of a C array definition. The names for the nodes (`SimpleDeclaration`, `ArrayDeclarator`, etc.) are defined in the C++ standard [6].



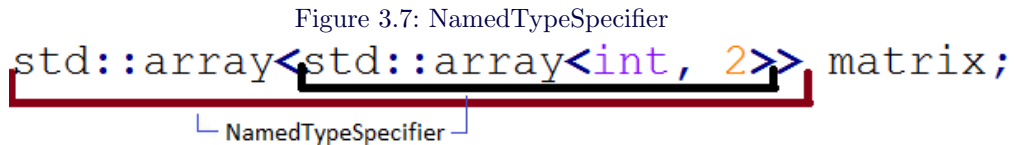
To determine whether a variable is an array or not, the `std::array` refactoring has to inspect the AST node structure and check if all of the following conditions are met:

1. `SimpleDeclaration` contains an `ArrayDeclarator`.
2. `SimpleDeclaration` doesn't contains only one `LiteralExpression` that represents a string.

3 Implementation

Problem 2: Multidimensional arrays

Figure 3.7 shows the AST node structure of a multidimensional `std::array`. The names for the nodes (`NamedTypeSpecifier`, etc.) are defined in the C++ standard [6].



The data type of the array elements is defined in the innermost `NamedTypeSpecifier`. The factory class “`ICPPNodeFactory`” can’t be used to create a more specific object than an `ICPPASTQualified Name` that represents the whole data type of the variable. It takes as argument an `IASTName` that can be created out of a char array. So we needed to create a string that represents the whole data type and convert it to a char array to create the correct `IASTName` that can then be used for the creation of the `ICPPASTQualified Name`.

Problem 3: Arrays as arguments

By default, occurrences of the array are replaced with a call to the `std::array::data` member function. This ensures that existing function signatures, that have array parameters don’t have to be changed. There are a few exceptions though. For example, an `ArraySubscript-Expression` such as “`arr[5] = 2;`” should not be modified, because the class `std::array` overloads this operator.

Problem 4: Include handling

If the array header is not already included the array refactoring will add it. More details about the realization of this feature can be found in the section “Problem 4: Include handling”.

3.2.9 The pointer parameter refactoring

The following subsection describes the algorithms that were used to solve the problems related to the pointer parameter refactoring.

Problem 1: Recognizing pointer parameters

As described in the Analysis section, pointer parameters can not always be refactored to reference parameters. If all of the following conditions are met, the refactoring is possible:

1. The Declarator of the parameter contains a * sign.
2. The DeclSpecifier of the parameter does not contain char, wchar_t, char32_t or char16_t.
3. There is no NULL-check of the parameter in the function body.

This does not exclude array parameters as shown in the problem description. However, a lot of additional static analysis would be required to decide whether the parameter is really an array or just a normal pointer parameter. Therefore, it is outside the scope of this project.

Problem 2: Dealing with overloaded functions

The problem of dealing with overloaded functions can be solved elegantly by using the index to search for occurrences of the same function using its binding object. Because each overloaded function has its own binding object, the search results will only contain function calls and function declarations of exactly that function. This was described above in the section “3.2.5 Searching across multiple files”.

Problem 3: Adapting all occurrences of the function

Similarly, the index finds all occurrences, no matter in which file they appear. This solves “Problem 3” as well. It is important to note, that each file has its own AST and therefore needs its own ASTRewrite object in order to modify the file. Section “3.2.5 Searching across multiple files” contains more information on this topic.

4 Test refactoring against real life code

In this section we used an existing C++ application called fish shell [7] to test our Pointerminator plug-in. We took a snapshot of the application's source code (in November 2013) from Github [8] and tried to apply as many of the refactorings as possible. More information about fish shell can be found under fishshell.com.

4.1 Statistics

In week eleven we tested the Pointerminator plug-in and created a statistic of the results. After that we fixed some bugs and implemented some of the special cases that caused the refactorings to fail. Finally, we tested it again and created another statistic at the end of the term project in week thirteen.

4.1.1 Statistics of week 11

The plug-in added 18 `std::string` markers, 154 `std::array` markers and about 660 pointer parameter markers to the fish shell code base. The plug-in was able to successfully refactor half of the `std::string` markers, about 2/3 of the `std::array` markers. 29 of the unresolved `std::array` markers are struct arrays. Because the plug-in found a lot of pointer parameters we only tested the first 120 markers. 51 pointer parameters could be refactored successfully. The others led to compilation errors. Table 4.1 shows the results:

Table 4.1: Refactoring statistics

Refactoring	Markers set	Solved	Unsolved
<code>std::string</code>	18	9 (50%)	9 (50%)
<code>std::array</code>	154	111 (72%)	43 (27%)
pointer parameter	662	51 (42.5%)	69 (57.5%)

4.1.2 Statistics of week 13

Most changes we have done were for the `std::array` refactoring. While for the other two refactorings the successful and unsuccessful count hasn't changed, the amount of successful refactored arrays has improved. Table 4.2 shows the results:

Table 4.2: Refactoring statistics

Refactoring	Markers set	Solved	Unsolved
<code>std::string</code>	18	9 (50%)	9 (50%)
<code>std::array</code>	154	114 (74%)	40 (26%)
pointer parameter	662	51 (42.5%)	69 (57.5%)

4.2 Examples for `std::string` refactoring

Here we show some code examples that were found in the fish shell code base that can be refactored with the string refactoring of the

4 Test refactoring against real life code

Pointterminator plug-in. More information about this refactoring can be found in section “Replace C string with std::string”.

4.2.1 Standard `wchar_t*` variable

In the files `builtin_set_color.cpp` and `builtin_set.cpp` the checker of the string refactoring found some standard `wchar_t` variables. These variables can be replaced with `std::wstring` variables which leads to more efficient and safer code. Listing 4.1 shows this example:

Listing 4.1: Before refactoring

```
const wchar_t *short_options = L"b:hvocu";
int c = wgetopt_long(argc, argv,
    short_options, long_options, 0);
```

Listing 4.2: After refactoring

```
const std::wstring short_options = L"b:hvocu";
int c = wgetopt_long(argc, argv,
    short_options.c_str(), long_options, 0);
```

4.2.2 char string variable with size definition

In another file the checker found a char buffer variable that contains a string whose length is shorter than the actual buffer size. The string refactoring was able to successfully refactor this buffer into a `std::string` variable and reserve the right amount of memory with a call to the `std::string::reserve` member function. Listings 4.3 and 4.4 show the code before and after the refactoring was applied:

Listing 4.3: Before refactoring

```
char buff[128] = "\x1b[";
```

Listing 4.4: After refactoring

```
std::string buff = "\x1b[";
buff.reserve(128);
```

4.3 Examples for `std::array` refactoring

Here we show some code examples that were found in the fish shell code base that can be refactored with the array refactoring of the Pointerterminator plug-in. More information about this refactoring can be found in section “Replace C array with `std::array`”.

4.3.1 Standard C array variables

2/3 of all C arrays could be refactored with the `std::array` refactoring. For example, the C array that can be found in the file `iothread.cpp` was refactored as shown in Listings 4.5 and 4.6:

Listing 4.5: Before refactoring

```
int pipes[2] = {0, 0};
```

Listing 4.6: After refactoring

```
std::array<int, 2> pipes = { { 0, 0 } };
```

The array refactoring can also handle arrays whose modifier doesn't contain a size expression. It will calculate the required size by itself. Listings 4.7 and 4.8 show how the refactoring of such a variable was applied in the file `common.cpp`:

Listing 4.7: Before refactoring

```
const wchar_t *sz_name[] = { L"kB", L"MB", L"GB",  
L"TB", L"PB", L"EB", L"ZB", L"YB", 0 };
```

Listing 4.8: After refactoring

```
std::array<const wchar_t*, 9> sz_name = { { L"kB",  
L"MB", L"GB", L"TB", L"PB", L"EB", L"ZB",  
L"YB", 0 } };
```

The `InitializerList` can also contain method calls that return the correct data type. Listings 4.9 and 4.10 are also from the file `common.cpp`:

4 Test refactoring against real life code

Listing 4.9: Before refactoring

```
char tmp[3] = { '.', },
extract_most_significant_digit(&remainder), 0 };
```

Listing 4.10: After refactoring

```
std::array<char, 3> tmp = { { '.', },
extract_most_significant_digit(&remainder), 0 } };
```

Macros inside the definition of a C array are also handled correctly by the array refactoring. Listings 4.11 and 4.12 show such an example from the file fish_pager.cpp:

Listing 4.11: Before refactoring

```
int pref_width[PAGER_MAX_COLS];
```

Listing 4.12: After refactoring

```
std::array<int, PAGER_MAX_COLS> pref_width;
```

Also, static arrays can be refactored with the array refactoring. The static modifier will be placed correctly before the datatype array. Listings 4.13 and 4.14 show an example which can be found in the env.cpp file:

Listing 4.13: Before refactoring

```
static const wchar_t * const
locale_variable[] = {
L"LANG", L"LC_ALL", L"LC_COLLATE", L"LC_CTYPE", L"
LC_MESSAGES", L"LC_MONETARY", L"LC_NUMERIC", L"
LC_TIME", NULL};
```

Listing 4.14: After refactoring

```
static std::array<const wchar_t* const, 9>
locale_variable = { {
L"LANG", L"LC_ALL", L"LC_COLLATE", L"LC_CTYPE", L"
LC_MESSAGES", L"LC_MONETARY", L"LC_NUMERIC", L"
LC_TIME", NULL} };
```

4.3.2 C array with method calls

C arrays can also be refactored when they are passed as arguments in a function or method call. The example from listings 4.15 and 4.16 can be found in the file color.cpp:

Listing 4.15: Before refactoring

```
static unsigned char term8_color_for_rgb(const unsigned
char rgb[3]) {
const uint32_t kColors[] = { 0x000000, //Black
0xFF0000, //Red
0x00FF00, //Green
0xFFFF00, //Yellow
0x0000FF, //Blue
0xFF00FF, //Magenta
0x00FFFF, //Cyan
0xFFFFFFFF, };
return convert_color(rgb, kColors, sizeof
kColors / sizeof *kColors);}
```

Listing 4.16: After refactoring

```
static unsigned char term8_color_for_rgb(const unsigned
char rgb[3]) {
std::array<const uint32_t, 8> kColors = { {0x000000,
0xFF0000,
0x00FF00,
0xFFFF00,
0x0000FF,
0xFF00FF,
0x00FFFF,
0xFFFFFFFF } };
return convert_color(1, rgb, kColors.data(), kColors.
size());}
```

The array refactoring replaced the calculation of the array size with a call to the built-in member function `std::array::size`. In another step, the function signature of the `convert_color()` function could be adapted, so that it takes a `std::array` parameter instead of a C array. It would then also be possible to get rid of the size parameter, because the `convert_color()` function could simply call the `std::array::size` member

4 Test refactoring against real life code

function of the `std::array` parameter. However, this is outside of the scope of this refactoring.

The listings 4.17 and 4.18 show another array refactoring example:

Listing 4.17: Before refactoring

```
wchar_t static_buff[256];
...
while (status < 0) {
    if (size == 0) {
        buff = static_buff;
        size = sizeof static_buff;
    } else {
        ...
        buff = (wchar_t *) realloc(
            (buff == static_buff ? NULL : buff), size);
        ...
    }
}
```

Listing 4.18: After refactoring

```
std::array<wchar_t, 256> static_buff;
...
while (status < 0) {
    if (size == 0) {
        buff = static_buff.data();
        size = sizeof static_buff.data();
    } else {
        ...
        buff = (wchar_t *) realloc(
            (buff == static_buff.data() ? NULL : buff), size);
        ...
    }
}
```

4.4 Examples for pointer parameter refactoring

Here we show some code examples that were found in the fish shell code base that can be refactored with the pointer parameter refactoring of the `Pointerminator` plug-in. More information about this refactoring can be found in section “Replace pointer parameter with reference”.

4 Test refactoring against real life code

4.4.1 Pointer parameter refactoring in single file

Below in listings 4.19 and 4.20 is an example where the method taking a pointer parameter is only used in a single file. So this method will only be called from the builtin_test.cpp file. The pointer parameter refactoring replaces the parameter and all calls correctly:

Listing 4.19: Before refactoring

```
static bool parse_number(const wstring &arg, long long *
    out) {
    const wchar_t *str = arg.c_str();
    wchar_t *endptr = NULL;
    *out = wcstoll(str, &endptr, 10);
    return endptr && *endptr == L'\0';
}
...
return parse_number(left, &left_num) && parse_number
    (right, &right_num) && left_num == right_num;
```

Listing 4.20: After refactoring

```
static bool parse_number(const wstring &arg, long long &
    out){
    const wchar_t *str = arg.c_str();
    wchar_t *endptr = NULL;
    out = wcstoll(str, &endptr, 10);
    return endptr && *endptr == L'\0';
}
...
return parse_number(left, left_num) && parse_number
    (right, right_num) && left_num == right_num;
```

4 Test refactoring against real life code

4.4.2 Pointer parameter refactoring in multiple files

The pointer parameter refactoring can also replace parameters if the method is used in multiple files and/or declared in a header file. The code shown in listings 4.21 and 4.22 can be refactored by the plug-in:

Listing 4.21: Before refactoring

```
FILE: env_universal_common.cpp
    void connection_destroy(connection_t *c) {
    if (c->fd >= 0) {
        if (close(c->fd)) {wpperror(L"close");}}}}

FILE: env_universal_common.h
    void connection_destroy(connection_t *c);

FILE: env_universal.cpp
    connection_destroy(&env_universal_server);

FILE: fishd.cpp
    connection_destroy(&c);
    connection_destroy(&*iter);
```

Listing 4.22: After refactoring

```
FILE: env_universal_common.cpp
    void connection_destroy(connection_t& c){
    if (c.fid >= 0){
        if (close(c.fid)){wpperror(L"close");}}}}

FILE: env_universal_common.h
    void connection_destroy(connection_t &c);

FILE: env_universal.cpp
    connection_destroy(env_universal_server);

FILE: fishd.cpp
    connection_destroy(c);
    connection_destroy(*iter);
```

5 Conclusion

This chapter is about the results of our term project and also describes how this project can be continued.

5.1 Achievements

During our term project we made the following achievements:

- We analysed the different use cases for pointers in C and C++ and compared them to more modern solutions that could be used.
- A solution was implemented in the form of an Eclipse CDT plug-in and its functionality was continuously tested with a set of unit tests.
- The plug-in was tested with a real C++ code base and the results were documented accordingly.

5.2 Future Work

Here are some improvements that could be done to the the existing Pointerterminator plug-in in the future:

- The string refactoring could be extended, so that it also supports dynamically allocated strings on the heap.
- Improve array refactoring to support array parameters.
- Use index instead of visitors for string and array refactorings to add support for member variables.
- Find a way to get more reliable unit tests for the plug-in.

Bibliography

- [1] Static Analysis for CDT, <https://wiki.eclipse.org/CDT/designs/StaticAnalysis>, 08.12.2013
- [2] Variable shadowing, http://en.wikipedia.org/wiki/Variable_shadowing, 14.12.2013
- [3] Overview of Parsing, http://wiki.eclipse.org/CDT/designs/Overview_of_Parsing, 08.12.2013
- [4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*
- [5] Class ASTRewrite, <https://www.cct.lsu.edu/~rguidry/eclipse-doc36/org/eclipse/cdt/core/dom/rewrite/ASTRewrite.html>, 08.12.2013
- [6] ISO/IEC 14882:2011.
- [7] fish shell, <http://fishshell.com>, 12.12.2013
- [8] fish shell on GitHub, <https://github.com/fish-shell/fish-shell>, 12.12.2013
- [9] Redmine, <http://www.redmine.org>, 13.12.2013
- [10] Apache Maven Project, <http://maven.apache.org>, 13.12.2013
- [11] Jenkins, <http://jenkins-ci.org>, 13.12.2013
- [12] FindBugs, <http://findbugs.sourceforge.net>, 13.12.2013
- [13] HSR Git, <https://git.hsr.ch>, 13.12.2013
- [14] Git, <http://git-scm.com>, 13.12.2013
- [15] ch.hsr.ifs.cdttesting, <https://github.com/IFS-HSR/ch.hsr.ifs.cdttesting>, 13.12.2013