Code Comprehension for the Move Semantics in C++

Attila Gyén[1], Dániel Kolozsvári[1] and Norbert Pataki[1,*]

[1]*Department of Programmming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, 1/C Pázmány Péter st., Budapest, H-1117, Hungary*

Abstract

The performance of C++ code can be improved with move semantics significantly. This approach is established by the C++11 standard. Applying these constructs, programmers can build faster and more memory-efficient code, however, there are not only pros in the move semantics, it has its own problems as well. The integrated development environments and their code comprehension capabilities are taken advantage of by the massive amount of programmers. These tools are typically based on static analysis which processes the source code with no execution. In this paper, we define scenarios when the programmers should pay attention to code because of the weird circumstances. We present an approach that can assist the programmers in validation of the move-related code. For this approach, we take advantage of the Clang compiler infrastructure and the Microsoft's web-based IDE, called Monaco. We present how our code comprehension approach helps the programmers.

Keywords

move semantics, C++, Clang, code comprehension, Monaco Editor

1. Introduction

One of the fundamental differences between classic and modern C++ belongs to the move semantics [1]. With the help of the related constructs, one can pass heap memory or other resources from an object or block to an other one in a very efficient way [2]. However, the resource in the original may become unavailable. C++'s move semantics may improve the execution time of algorithms since slow copying code snippets can be avoided, and the approach can improve the memory consumption, if the resource to move is heap memory [3]. Move semantics are designed to improve the performance of the classical copy semantics, but using the new approach can fall back to the former solution in special cases [4].

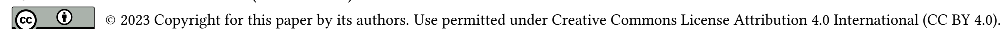
Methods of static code analysis take advantage and process of the source code without execution. Static analysis can be used for many purposes, for instance, finding bugs, refactor or rejuvenate the code [5]. These methods can be for used measuring software metrics, or visualize the code [6]. Approaches for better code comprehension are important to highlight specific details of the source code and assist the developers to understand the code base and maintain it in a more convenience way [7].

SQAMIA 2023: Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, September 10–13, 2023, Bratislava, Slovakia

*Corresponding author.

gyenattila@gmail.com (A. Gyén); kolozsvari.dl@gmail.com (D. Kolozsvári); patakino@elte.hu (N. Pataki)

0000-0002-7519-3367 (N. Pataki)



© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3588/p1.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

In this paper, we propose a code comprehension approach for the move semantics. The proposed approach takes advantage of the Clang compiler infrastructure for parsing the C++ code and finding move semantics-related parts and Microsoft's Monaco web-based IDE for the code comprehension because we successfully used these tools earlier [8].

The rest of this paper is organized as follows. In Section 2, the background of the move semantics is detailed. Section 3 presents the proposed approach regarding the Clang-based static analysis tool and the visualization in the Monaco editor. The evaluation of the method is discussed in Section 4. Finally, this paper concludes in Section 5.

2. Move Semantics in C++11

C++ is notorious for effective hardware utilization and fast execution. C++ compilers are able to generate efficient executable code that takes advantage of the hardware elements [9].

Classic C++ is based on the copy operations. By default, an object is passed or returned by value that means its copy constructor is called and the passed or returned object instance is a copy, not the same object. However, C++ objects can encapsulate resource management [10].

Next to the classical copy operations, C++11 introduces the move operations (e.g. the move constructor and the move assignment operator) which have weaker postconditions than the classical ones. They do not guarantee that copied object remains in the same state, it can be invalidated. This solution gives opportunity to the pass the ownership of object to another object which can be in a different block or function.

Listing 1: "Copy semantics vs move semantics"

```
std::string f( int id )
{
 std::string s = "Hello";
 // ...
 s.push_back( '!' );
 return s;
}

std::string&& g( int id )
{
 std::string s = "Hello";
 // ...
 s.push_back( '!' );
 return std::move( s );
}
```

Listing 1 presents two similar functions, both return a `std::string` objects. Function `f` uses copy constructor to return a `std::string`. In this case, the copy constructor allocates heap memory for the caller and the destructor deallocates the heap memory allocated in the function. Function `g` takes advantage of the move semantics and the caller utilizes the heap memory allocated in the callee. The object remains in an invalidated state, but it does not cause any problem because the execution of the function is over.

3. The Proposed Approach

While the move semantics introduced in modern C++ has a lot of benefits and great potential when it comes to optimizing the source code execution, over-, or misusing it can lead to unexpected behavior. In our paper, we aim to identify and address some of the potential faults when it comes to implementing move semantic-based operations.

3.1. Static Analysis Approach

3.1.1. Static Analyser Checkers

Clang-Tidy is a Clang-based static analyser tool providing hints and options to optimize the source code, make it more readable and less bug-prone. It implements static analyser checkers, covering the most common issues when it comes to using move semantics. However, these checkers can have their own limitations and also lack visualization. Therefore, we extend the move semantics-related functionality of the Clang-tidy and visualize of the retrieved code smells. For the latter, tools like CodeChecker or – in our case – Microsoft's Monaco web-based IDE can be used to enhance the analysis results.

In this paper, we focus on two specific move related issues: using an entity which has already been moved hence making it invalid, and calling move operations when doing so will not have any effect on how the program executes, although it would be expected to do so. Both of these issues are covered by already existing clang-tidy checkers: bugprone-use-after-move and performance-move-const-arg.

The purpose of the first is to warn for all the occurrences, when an object (local variables or function parameters) is used after it has already been moved, without reinitializing it as Listing 2 presents this scenario.

Listing 2: "Use after move"

```
std::string str = "Hello, world!\n";
std::vector<std::string> messages;
messages.emplace_back(std::move(str));
std::cout << str; // Warning
```

It can be seen in Listing 3, the checker is flow-sensitive but not path-sensitive, meaning that it will consider the fact if the `std::move` call to be examined is reachable or not (e.g. it appears in a dead block), but for example mutually exclusive branches are not considered (non path-sensitive).

Listing 3: "Use after move - false positive"

```
if (i == 1) {
 messages.emplace_back(std::move(str));
}
if (i == 2) {
 std::cout << str; // Warning (false positive)
}
```

Checker performance-move-const-arg emits a warning by default for the following scenarios:

- if `std::move()` is called with a constant argument,
- if `std::move()` is called with an argument of a trivially-copyable type,
- if the result of `std::move()` is passed as a const reference argument.

Listing 4: "Move with no effect"

```
const string s;
return std::move(s); // Warning: std::move of the const variable has no
    effect

int x;
return std::move(x); // Warning: std::move of the variable of a
    trivially-copyable type has no effect

void f(const string &s);
string s;
f(std::move(s)); // Warning: passing result of std::move as a const
    reference argument; no move will actually happen
```

3.1.2. Indirect Copy Fallbacks

While the expected behavior of this clang-tidy checker might be straightforward, there are scenarios which are currently not covered by this analyser, but we would want to examine with the help of our visualization tool. For this, we have enhanced the logic implemented: instead of focusing only on the "trivial" cases, we would like to cover implementations when the calling `std::move` indirectly would result in copying instead of moves. This would mean that the function call examined and warned for would not appear in our source code directly, but rather in a library we include. In the current iteration of the tool we prepare, we focus mainly on `std::move` calls where the fallback to copy would happen in the Standard Template Library (STL) library itself. In these scenarios, we would be interested in the location of the indirect `std::move` call in our own implementation instead of emitting messages for the STL library, since falling back to copy might be the expected behavior in some scenarios. Let us consider the following code snippet:

Listing 5: "Calling std::move on std::set will result in copy"

```
std::set<std::string> set_s{"string_1", "string_2"};
std::vector<std::string> vec_v{"string_3", "string_4"};
std::vector<std::string> vec_v2{"string_5", "string_6"};
std::move(set_s.begin(), set_s.end(), vec_v.begin()); // Warning should be
    emitted
std::move(vec_v.begin(), vec_v.end(), vec_v2.begin()); // No warning should
    be emitted
```

In this example, calling `std::move` would not fulfill the criteria listed earlier (having const arguments, etc.). These conditions would not be true when `std::move` is called indirectly, only for the `std::move` calls executed in the internal implementation of the library itself. In this case, since elements of the `std::set` would be considered as constants when the move operation would happen, a copy will be executed. To notify the developer it might not be the intended behavior, triggering the warning is only useful if it highlights the exact location where the copy-fallback is triggered indirectly – since this is the source code what the user if responsible for, and which should rely on the functionalities provided by the standard template library in the expected way.

To achieve this, we had to improve the business logic of the checker, while overcoming some limitations of the current AST-matching mechanism provided by the LLVM. There are three different matcher categories we can rely on: node, narrowing and traversal matchers. The first two matches for criterias concerning specific attributes or types of the nodes, while the third one allows traversal between the nodes, meaning that we can define our own conditions which the parent or child objects of the AST-node to be matched has to fulfill.

Example: The function call

Listing 6: "Example std::move call"

```
std::move(set_s.begin(), set_s.end(), vec_v.begin());
```

will result in the following (simplified) AST:

Listing 7: "Example AST"

```
`-CallExpr
  |-ImplicitCastExpr
  | `-DeclRefExpr // Function template: std::move
  |-CXXConstructExpr
  | `-MaterializeTemporaryExpr
  |  `-CXXMemberCallExpr
  |    `-MemberExpr // .begin
  |      `-ImplicitCastExpr
  |        `-DeclRefExpr // set_s
  |-CXXConstructExpr
  | `-MaterializeTemporaryExpr
  |  `-CXXMemberCallExpr
  |    `-MemberExpr // .end
  |      `-ImplicitCastExpr
  |        `-DeclRefExpr // set_s
  `-CXXConstructExpr
    `-MaterializeTemporaryExpr
     `-CXXMemberCallExpr
       `-MemberExpr // .begin
         `-DeclRefExpr // vec_v
```

If we want to have the desired indirect matching mechanism described earlier, it is not enough to traverse the AST by following the direct connections between the nodes: we want to identify

all the `std::move` calls, which indirectly (or directly) can lead to an `std::move` call, when a fallback to copy can happen. This means that when a `std::move` call happens, we have to examine the definition of the given `std::move` instance, which is defined as a template method implemented in the algorithm library (see the example above – instead of relying on child nodes, we have to analyse the definition of a method referenced by our call expression). To make the checker as robust as possible, we examine the `std::move` calls in a recursive manner, until we find the last method call which will be responsible for the actual move operation to be performed. When we have found the function call we looked for, we mark it, as well as the indirect method call triggering the whole procedure. The indirect method call should be the last one in the call chain, which is not part of the `std` namespace, since that is our own implementation and not part of the library we included. Now that we have marked both the direct and indirect calls, we will able to check the direct one for copy fallbacks, and we can emit the warning for the indirect call location.

For the example seen in Listing 5, the warning which can be seen on Listing 8 is generated and emitted.

Listing 8: "Now emitting a warning for copy fallbacks when moving std::set"

```
warning: std::move of the const expression has no effect; remove
    std::move() [performance-move-const-arg]
   std::move(set_s.begin(), set_s.end(), vec_v.begin());
   ^
```

Future work: checker could distinguish between other third-party libraries as well, therefore the warning message would not be triggered for the internal implementation of the library itself, but for the way how we use it in our own source code. Currently, this mechanism is implemented only for the `std` namespace provided by the STL library.

3.1.3. Input for visualization

For visualization purposes, we have to create the proper output format, which we will be able to process by using Monaco. For this, we have defined the following JSON format:

Listing 9: "Analysis result in JSON format"

```
{
  "<IDENTIFIER/NONE>|LINENUM_COLUMNSTART-COLUMNEND": {
    "moveNoEffect": [
      [
        {
          "move_location": "LINENUM_COLUMNSTART-COLUMNEND"
        },
        {
          "path": "<PATH TO THE WARNING LOCATION>"
        },
        {
          "reasoning": "WARNING MESSAGE CONTENT"
        }
```

```
   ]
  ]
 },
 "<IDENTIFIER/NONE>|LINENUM_COLUMNSTART-COLUMNEND": {
  "useAfterMove": [
   [
    {
     "use": "LINENUM_COLUMNSTART-COLUMNEND"
    },
    {
     "move": "LINENUM_COLUMNSTART-COLUMNEND"
    },
    {
     "path": "<PATH TO THE WARNING LOCATION>"
    },
    {
     "reasoning": "WARNING MESSAGE CONTENT"
    }
   ]
  ]
 },
 ...
}
```

The first JSON object represents a finding for the performance-move-const-arg checker. If it is possible, we determine the identifier the match applies for (meaning we have a named declaration instead of an expression), the line number, and position in the given line where the variable or expression triggering the warning is used. The tag “moveNoEffect” means the result is based on this checker, "move_location" refers to the move call which (indirectly) leads to a copy fallback: since we have distinguished between the indirect and direct move calls, it is possible that the "move_location" attribute points to our own source code, while the exact fallback location does not appear in the results.

Tag “path” refers to the file path which was analysed when the warning was triggered: this is useful since it is possible that a finding located in a compilation unit originates in a header file which was included in our source code. Attribute “reasoning” contains a briefly modified version of the warning message originally emitted by the checker.

The second JSON object is tagged with “useAfterMove” meaning it is generated by the bugprone-use-after-move checker. Here “use” refers to the source code location when we try to use the previously moved variable without reinitializing it, “move” points to the location where the `std::move` call happens, while “path” and “reasoning” attributes behave similarly to the previous example.

To make the results easily readable and processable for larger projects, the checkers will generate a JSON structure containing entries defined above for each compilation unit. The name of the JSON file generated will equal to the name of the source code file which is currently being analysed. To avoid duplications, a file-structure similar to the original project structure will be generated under a given root directory ("move_stats") containing all the JSON files generated

for our original C++ source files.

For example, when analysing the clang-tidy project itself, the following result structure will be generated:

Listing 10: "Example file structure"

```
move_stats
|__ home
   |__ koldaniel
      |__ llvm
         |__ llvm-project
            |__ clang-tools-extra
               |__ clang-tidy
               |  |__ ClangTidy.cpp.json
               |  |__ ClangTidyCheck.cpp.json
               |  |__ ClangTidyDiagnosticConsumer.cpp.json
               |  |__ ClangTidyModule.cpp.json
               |  |__ ClangTidyOptions.cpp.json
               |  |__ ClangTidyProfiling.cpp.json
               |  |__ GlobList.cpp.json
               |  |__ abseil
               |  |  |__ AbseilTidyModule.cpp.json
...
```

3.2. Visualization in the Monaco Editor

After our parser has finished its job and generated the right output, we need to parse that output and transform the information from it. The output generated by the parser is not in the correct shape, therefore we need to transform the data into the shape that will be processable for the Monaco Editor.

Listing 3.2 presents a snippet from the JSON generated by the parser.

```
"variableName|rowNr_colNrStart-colNrEnd": {
 "moveNoEffect": [
   [
    {"move_location": "rowNr_colNrStart-colNrEnd"},
    {"path": "pathToTheAppropriateHeaderOrCppFile"},
    {"reasoning": "theCorrespondingErrorOrWarningMessage"}
]]}
```

Every object in the JSON file starts with a key which is assembled by the variable name and the exact location in the file separated with a pipe character. Note that the pipe, underline and dash characters are necessary for the JavaScript parser to be able to split the string in a correct way. Then comes the effect's name. It can be `moveNoEffect` or `useAfterMove`. These two keys are hardcoded in the static parser. The designations speak for themselves. Every effect is an array of an array of objects. The inner arrays represent different usage of the given variable in the source code. Every inner array gives more information about how the variable was

handled. The key names are strict. Every inner array has the "path" and the "reasoning" keys. The former stores the file where the effect happened the latter gives descriptive information about what is the problem with the usage of the variable. The "move_location" refers to where the variable was moved. If the variable is used after a move the object would have two new keys. Namely the "use" and the "move".

The next step was to create a JavaScript parser that will be able to process this JSON file and transform it into a new shape. First, we need to create a container that will hold the raw source code and will display it like the Visual Studio Code editor. The source code comes from a file selected by the user. If a file was analyzed with the static parser the JSON will be placed right beside the file. When a source file is opened the JS parser will automatically open up the corresponding JSON file too and start to process it.

To create the VSCode-ish-like editor a new model must be instantiated from the MonacoEditor. Its input parameters are the source code passed in an array and a string that represents the programming language. In our case, the latter will be 'cpp'. This is necessary to add because the editor will highlight the code as it was opened in a real IDE or text editor. After the model is created, we are ready to add different markers or decorators as they are called. Delta decorators are responsible to color the selected background in the editor. They take an array of objects as an input parameter. Every object is a configuration of what the current decorator should do. The Listing 3.2 presents the schema in which `'className'` is the name of the CSS class in string format that should be applied on the selected range.

```
{
 range: new monaco.Range(rowNr, colStart, rowNr, colEnd),
 options: {
  isWholeLine: false,
  'className',
  hoverMessage: [
   value: 'messageComesHere'
  ]
 }
}
```

Figure 1 presents a visualized sample code.

Different colors represent different effects in the code. The green stands for variable declaration, blue indicates the usage of the `std::move()` without any negative effect in the program, the red presents a usage of a variable that was moved previously, and yellow indicates the usage of the `std::move()` that can cause an error in the program.

Figure 2 presents whether we hover the cursor on any of the highlighted text the appropriate message will show up. Figure 3 presents a warning message when a variable is used after it is moved.

We have added one more feature to show the possible error with messages in another way. Monaco Editor provides an option to display model markers in the editor instance. First, we need to assemble the list of the markers we want to add. One of the biggest advantages of these markers is that we can use different levels of severity in every marker. For example, we can show it as an error, warning, or as a simple information marker. And with the help of the up

```cpp
#include <memory>
#include <iostream>
#include <string>

std::string f(const std::string &s) {
  std::move(s);

  std::string s2;
  f(std::move(s2));
}

int main() {
  struct S {
    std::string str;
    int i;
  };
  S s = { "Hello, world!", 42 };
  S s_other = std::move(s);
  s.str = "Lorem ipsum";
  s.i = 99;
  s.i = 100;
  std::cout << s.str;
  S s_other_other = std::move(s_other);
  s_other.i = 42;

  int x;
  return std::move(x);
  return std::move(x);
}
```

Figure 1: Visualized sample C++ code

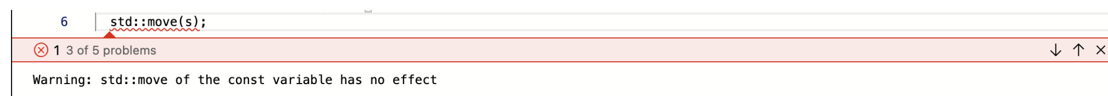


Figure 2: A highlighted warning

and down arrows on our keyboard, we can jump from one marker to another.

Markers can be added with the `setModelMarkers` method. The appropriate model and its markers should be passed as input arguments.

```
22
23    s_other used after it was moved  (s_other);
24    s_other.i = 42;
```

Figure 3: A highlighted warning

Table 1
Numbers of move-related findings in the source of Clang and Clang-Tidy

Finding	Numbers of Findings in Clang	Numbers of Findings in Clang-Tidy
Trivially-copyable	4870 (70.13%)	767 (94.23%)
const reference	1174 (16.91%)	32 (3.93%)
used after moved	740 (10.66%)	14 (1.72%)
const expression	156 (2.25%)	1 (0.12%)
const variable	1 (0.01%)	0 (0%)
Sum	6944 (100%)	814 (100%)

4. Evaluation

We evaluated our static analysis tool on two software artifacts. We analyse the source code of the Clang compiler and Clang-Tidy tool and searching for problems related to move semantics.

Our tool found five different kinds of misuse of the move constructs:

- moving a trivially-copyable variable
- passing the result of std::move() as a const reference argument
- variable usage after it is moved
- moving the const expression that has no effect
- moving the const variable that has no effect

These problems do not mean runtime bugs necessarily, but improvement of the referred code snippets is considerable.

Table 1 presents the numbers of different smell findings in Clang and Clang-Tidy. As one can see, the most typical problem is the moving of trivially-copyable variable that causes no problem at all. However, a rather high number of findings means our proposed visualization tool is considerable.

5. Conclusion

C++11 introduces the move semantics for improved performance. However, it is not a straightforward mechanism, its usage contains many pitfalls. In this paper, we propose a compound solution to avoid the pitfalls. First, we extended a Clang-Tidy static code analysis checker that processes the C++ code and detect the problematic or overcomplicated code snippets and prepares these issues for visualization. The visualization is executed in the Microsoft Monaco editor for an improved code comprehension. We evaluated our tool with open-source software

artifacts and realized that IDE visualization is considerable in order to reduce the move-related problems.

References

[1] Á. Baráth, Z. Porkoláb, Automatic checking of the usage of the C++ move semantics, Acta Cybernetica 22 (2015) 5–20. URL: https://cyber.bibl.u-szeged.hu/index.php/actcybern/article/view/3866. doi:10.14232/actacyb.22.1.2015.2.

[2] B. Stroustrup, Thriving in a crowded and changing world: C++ 2006–2020, Proc. ACM Program. Lang. 4 (2020). URL: https://doi.org/10.1145/3386320. doi:10.1145/3386320.

[3] Á. Baráth, Z. Porkoláb, Attribute-based checking of C++ move semantics, in: Z. Budimac, T. G. Grbac (Eds.), Proceedings of the 3rd Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, volume 1266, CEUR-WS.org, 2014, pp. 9–14.

[4] B. Stroustrup, The C++ Programming Language, 4th ed., Addison-Wesley, 2013.

[5] P. Pirkelbauer, D. Dechev, B. Stroustrup, Source code rejuvenation is not refactoring, in: J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, B. Rumpe (Eds.), SOFSEM 2010: Theory and Practice of Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 639–650. doi:10.1007/978-3-642-11266-9_53.

[6] B. Babati, G. Horváth, V. Májer, N. Pataki, Static analysis toolset with Clang, in: Proceedings of the 10th International Conference on Applied Informatics (ICAI 2017), 2017, pp. 23–29.

[7] E. Fülöp, A. Gyén, N. Pataki, Code comprehension for read-copy-update synchronization contexts in C code, in: S. Bourennane, P. Kubicek (Eds.), Geoinformatics and Data Analysis, Springer International Publishing, Cham, 2022, pp. 187–200.

[8] E. Fülöp, A. Gyén, N. Pataki, Monaco support for an improved exception specification in C++, IPSI Transactions on Internet Research 19 (2023) 24–31. doi:10.58245/ipsi.tir.2301.05.

[9] B. Babati, N. Pataki, Memory consumption of objects in C++, ICOOOLPS'22: Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems, 2022. URL: https://2022.ecoop.org/details?action-call-with-get-request-type=1&faa6b557e27743f2baa456a6316ac43eaction_17426506610f9506d5d198070672f9f7835cc429bc1=1&__ajax_runtime_request__=1&context=ecoop-2022&track=ICOOOLPS-2022-papers&urlKey=2&decoTitle=Memory-Consumption-of-Objects-in-C-, preprint.

[10] B. Stroustrup, Foundations of C++, in: H. Seidl (Ed.), Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 1–25.