Proceedings of the 10th International Conference on Applied Informatics
Eger, Hungary, January 30–February 1, 2017. pp. 271–278
doi: 10.14794/ICAI.10.2017.271

Symbol Clustering: Resolving ambiguous symbol references of large-scale C / C++ projects based on linkage information

Richárd Szalay[a], Zoltán Porkoláb[a], Dániel Krupp[b]

[a]Eötvös Loránd University, Faculty of Informatics,
Dept. of Programming Languages and Compilers
szalayrichard@inf.elte.hu, gsd@elte.hu

[b]Ericsson Ltd.
daniel.krupp@ericsson.com

Abstract

Name mangling – constructing linkage name by concatenating enclosing namespace and class names, function and template parameter information – is the main tool in C++ to make distinction between separate symbols declared with the same identifier, to resolve function overloading and others. The same mangled name in different translation units forming a single executable should refer to the same symbol. Not surprisinly, many development framework and code comprehension tool utilizes mangled name identity to implement actions like "go to definition" or "all references". However, in large projects, where multiple binaries are constructed, mangled names can be ambigous. In this paper we show, how relevant is this problem, and how the most important development tools are handling it. We describe our clustering–based algoritm to show how precison can be improved in symbol resolution. We implemented our algorithm as a working prototype to show the practical possibilities. Our solution can be utilized via the Language Server Protocol to improve the functionality of third-party developmnet tools.

Keywords: Mangled name, C++ programming language, Linking

MSC: 68N15 Programming languages

1. Introduction

Large scale software systems are scaling up to million lines of code. The source code of the Linux kernel is around 17 million lines of code. Telecom systems, developed by hundreds of designers over modre decades are is similar size. It is

Xerces [21]	18 547 (19, 0.1%)	63 167 (348, 0.5%)	14 933 (23%)	11 123 (18%)	37 111 (59%)
CodeCompass [7]	337 833 (143, 0.04%)	622 220 (473, 0.07%)	229 737 (37%)	127 864 (20%)	264 619 (43%)
LLVM [19]	1 793 978 (334, 0.02%)	4 545 086 (16 469, 0.4%)	1 213 123 (27%)	708 156 (16%)	2 623 807 (57%)
Linux [22]	70 579 (503, 0.7%)	411 691 (11 074 (2.7%)	33 746 (8%)	66 228 (16%)	311 717 (76%)
TSP [20]	410 946 (9 721, 2.3%)	2 091 578 (194 118 9.3%)	377 393 (18%)	321 964 (15%)	1 392 221 (67%)

Table 1: Number of *functions* discovered and found ambiguous in our tests projects (in parantheses the number of ambiguous names and nodes)

easy to see, that navigation in these large codebases are extremely challenging and requires tool support. Modern development frameworks [1, 2, 3, 4] and code comprehension tools [5, 6] provide advanced navigation features, like *go to definition* which jumps to the place of the definition or *all references* which iterates all places of reference of a given symbol. Unfortunately, we found that current development and comprehension tools are not exact in these functionalities and make mistakes in certain circumstances.

The main reason of the probem is that many current tools implemented the mentioned features based on matching *mangled names* (see Section 3) which allows to be ambiguous when multiple executables are produced to form the released product. The prevalence of the problem was tested on a group of open source projects. It appears that, statistically, a considerable ratio of symbols $(0.5 - 10\%)$ are affected – thus the problem is not negligible for its impact on code comprehension and development time when developers have to resolve symbol references using their internal and external knowledge instead of a proper tool solving the problem for them.

To solve the ambiguity problem, in this paper we describe a new method to group the mangled names into clusters, based on the *linking information*, i.e. how they used by the linker to form the resulting binaries. Mangled names belonging to the same cluster should refer to the same symbol; and they are distinct from symbols with identical mangled names classified to different clusters. This way, we were able to significantly reduce the possible ambiguity in symbol resolution.

We implemented the method as part of CodeCompass [7] – an open source software comprehension tool – where symbol correlation is used in a number of functionalities. We evaluated our method using CodeCompass to parse various open source C++ projects and measured different implementation variants on the run-time and storage expense. Our final solution has a minimal overhead on the discovery process, therefore it is applicable for various development and code comprehension tools.

This paper is organised as follows: In Section 2 we describe the compilation and linking process of C and C++ projects. Here we introduce the notion of

mangled names. The ambiguity of mangled names and the One Definition Rule is discussed in Section 3. Here, we also overview some of the current development and comprehension tools regarding the handling of the problem. Our solution based on symbol clustering is introduced in Section 4. Our paper concludes in Section 5,

2. Building C/C++ programs

Software products written in C or C++ language take multiple steps to be built. First of all, a translation unit, usually a source file is handled by the preprocessor which, amongst other things, includes header files containing more symbols (usually declarations and inlinable definitions) to the to-be-compiled code. The preprocessor can add and remove blocks of the source code (conditional compilation) and textually change tokens, inplacing different code parts (preprocessor macros). This preprocessed code, which is still a simple text file, is then *compiled* into an object file. Thus, at this point, multiple object files could be created from the same initial source file, albeit with possibly different contents based on build configuration parameteres, such as using different include paths for the headers, or defining different macros – the most trivial example would be enabling log emission from certain functions in a debug build.

A set of object files is then *linked* together by the linker to form a binary executable or a library. The linker takes object files and other libraries to form this output binary. Built libraries, which are results of a linking command, can also be used in the creation of other binaries. Thus, the potentially vast number of different object files could be used for multiple binaries, e.g. using 32-bit and 64-bit architectures or built for different system libraries (such as different versions of networking stacks or databases).

Various tools exist which allow the instrumentation of the pipeline in a way that the actual executed commands are available for other software to read and understand, such as the JSON compilation database format [8].

In the case of the C++ programming language, the same name can be used in multiple places of the program for identifying different objects. Typical function names, like `init`, `operator«`, or `operator==` are recurring between different classes, as well as some common attributes. The *overloading* is a language element to define separate functions with the same name but with different signature. Also, the *namespace* is another language feature to distinguish different objects with the same name.

In C++ the *mangled name* [9, 10] is used to distinguish between different symbols of the same name. The mangled name is constructed using (possibly multiple) namespace and class information by concatenation, but its exact form is compiler-dependent. Compilers use mangled names to enable *operator overloading* [11], i.e. they generate different mangled names for functions with the same name but different parameter lists. Linkers use these mangled names to resolve symbols [12].

In the case of the C language, there is no such thing as namespace, class, or function overloading [13]. Still, in certain cases, such as specific optimisations on

how a method is called, *name decoration* occurs [14]. In this paper we will refer to the name visible to the linker as *mangled name*, both in the context of C and C++ for the sake of simplicity. While mangled names must be unique for all translation units linked into a single executable, this does not stand for large-scale projects where multiple executables are typical.

3. Ambiguity of mangled names

In a software comprehension activity the most frequent questions users ask are "Where is this method called, or type referenced?" and "What does the declaration or definition of this looks like?" [15]. A software comprehension tool should be able to answer these questions as precisely as possible, as accuracy ensures more optimised usage of the developers' time spent working. Both questions lead to the fundamental problem of correctly resolving references to the definition and usages of a type, a function, or variable, and other language components.

According to the *One Definition Rule* (ODR) of C++ [16], only one definition of any variable, function, class type, enumeration type, or template is allowed in any one translation unit. When resolving references to ordinary C functions, static and non-virtual C++ member functions, type names or non-polymorphic variables, the unique definition within a single translation unit can be found based on static information. Specifically, the function definition of non-virtual functions or ordinary C functions can be looked up based on function signature, which – according to [17] – contains the name of the function, the enclosing namespace, class of which the function is member of (in the context of C++), the type of the parameters, template parameter list (in case of function templates in C++), *cv*- and *ref*-qualifiers, unique type names (qualified with namespace) and scope-correct variable names.

There can be, however, more than one definition belonging to the same unique-name or signature, but defined in different translation units that are not linked together. This is a typical scenario in large-scale programs consisting of multiple separate executables and build configurations, e.g. every executable having a *main* () function as entry point. Since the translation unit containing the reference and the set of translation units linked together is known for the linker, it is possible for the linker to look up the correct, unique definition for any given reference.

In contrast, for a software development or comprehension tool, while the user is browsing a source file, the linkage context (the set of translation units linked together) where the definitions should be resolved is usually unknown [18]. This leads to ambiguous type, function or variable references. In some cases this ambiguity can be resolved automatically, by taking into consideration the linkage information. Current development frameworks and grokking tools, unfortunately, are far from perfect when resolving symbols. In the following, we overview how some of the most important tools perform when we execute a *jump to definition* query.

One of the most advance C/C++ development tool, *Microsoft Visual Studio* [1] shows a disambiguation page when encountering ambigous mangled names. If the

entire *solution* is configured for a certain dependency and the user changes the internal settings of the solution, Visual Studio decides which symbol a "jump to definition" query jumps to. This fine-tuning on the users' end seemingly does not affect "get calls" / "get usage"-like queries, which still show results with every possible option present, including those which are clearly not valid in the solution's current state.

Another advanced development framework, *JetBrains CLion* [2] analyses and builds symbol information when a project is configured, and one project having multiple separately configured executable targets misleads the IDE. A certain file is designated as location where function $f()$ is defined – our understanding currently reveals that the file which has a larger name in lexicographical order. The ambiguity is present even when a certain executable is being debugged by the IDE with "Step Into" showing the proper implementation being executed while "Jump to Definition" opening an entirely distinct source file.

The open source *NetBeans* [3] development tool does not show a disambiguation page at all, seemingly jumping to a file first detected for a certain symbol after the last build – this can be overridden by manually setting certain files to "Exclude from Build" after which the file's symbols won't contribute to the set of potential results. The file to which NetBeans jumps for a definition varies between client restarts, seemingly in a non-deterministic fashion; the only exception is when the symbol is *defined* in the same file where it is *used*: in this case *all* queries jump to this location in particular.

The well-known *Eclipse* [4] tool written in Java properly prioritises symbols explicitly defined in the same source file, but if the definition is not found in the file where a query is issued a disambiguation page is shown. Putting different builds into entirely different projects with their own `Makefile` solves this issue, but search queries does not traverse project boundaries.

The code comprehension tool *Woboq* [5] shows the locations where a symbol is defined when viewing information about a particular symbol, but the problem of Section 3 is present. Jumping to definition by clicking on a usage location jumps the user to the definition that has been first (in the order of build commands) discovered by the `codebrowser_generator` tool of *Woboq*. It binds usages and definitions in the same source file together, but further "clustering" capabilities are not present.

4. Clusters

As we discussed in Section 2 the One Definition Rule in C++ ensures, that every succesfully built executable contains only one instance of a symbol. Thus we can form clusters of symbols starting from the final executables.

Symbols are propagated from one file to another via subsequent build actions, forming a chain. The intermediate files in this chain do not carry extra information in terms of symbol availability: if `a.o` and `b.o` is linked into `a.out`, the target will contain the symbols from every source. Thus these intermediate files can be omitted

without the loss of generality. The solution to the problem described in this paper requires the consideration of *ultimate clusters*, a set of clusters into which symbols are propagated, and from where they are not propagated any further – the leaf nodes of the graph formed from the aforementioned build relations.

Assuming that we know the build actions that were used in the compilation of a given project, we can calculate the ultimate clusters for each build action by using the following algorithm. This algorithm takes a single build action B' – usually a *compile* command of a translation unit/source file – and it calculates a set of build actions that represent the ultimate clusters for the symbols found in the input build action's sources.

Consider a set of symbols which has been appropriately pre-filtered by a "good-enough" filtering method, e.g. based on *mangled names* described in Section 3. For each of these symbols, we calculate the clusters in which a symbol is found, and if two symbols have at least one common cluster, i.e. they appeared in the same binary at least once, the relation between them is considered a *strong match*. A strong match definitely indicates that the compiler – in accordance with the language rules – did emplace a relation between the two symbols, e.g. a function call was linked to use the definition which was found in the same cluster.

On the contrary, if the two symbols never appear in a common cluster, the relation is considered a *weak match*. A weak match indicates that no *discovered* relationships enforce strength to the match, but the symbols could still be related to each other. A weak match could, e.g., indicate a function which is called from a dynamically loaded library.

We also define a third category called *indefinite match* for matches that are indefinite in their strength – no extra information could have been retrieved from calculating the clusters or the clusters do not exist at all. This could happen from the fact that certain translation units were not compiled or we have no build information (see Section 2) available. Certain matches will deliberately fall into the latter category due to optimisation reasons. The fact that a match does not have any strength indication proves not to hinder search efficiency and accuracy any further if a query is unambiguous based on previously available properties, such as mangled name equivalence.

5. Conclusion

In this paper we investigated the problems related to mangled name usage for symbol resolution. While most of the available development environments and comprehension tools utilize mangled names, in large industrial C/C++ projects, where the build process may create multiple binary outputs (libraries or executables) they do not uniquely identifies the object in question. This may lead imprecise or even faulty behaviour in case of development frameworks, code comprehension, or static analysis tools. We measured the relevance of the problem and found that in typical open source projects 0.5 – 10 percent of ambigous mangled names are not uncommon.

As ambiguity resolution should reason whether symbols in separate translation units with the same mangled name are identical or different, to solve the problem we have to utilize the essential information of the build process. We developed an algorithm and implemented an industrial prototype to demonstrate the correct symbol resolution. Using the Language Server Protocol our solution can be used by third party development tools to improve their precision in actions, like "go to definition" or "list all references" type queries.

References

[1] Microsoft Visual Studio, version 14.0.24720.00-Update1
`https://www.visualstudio.com/`, 08/15/2016.

[2] CLion: a cross-platform IDE for C and C++, version 2016.2.1
`https://www.jetbrains.com/clion/`, 08/15/2016.

[3] The Netbeans IDE, version 8.1
`https://netbeans.org/`, 08/15/2016.

[4] The Eclipse project, Eclipse CDT (C++ Development Tools), version *Neon* 4.6.0
`http://www.eclipse.org/home/`, 08/15/2016.

[5] Woboq code browser, version 2.0.1
`https://woboq.com/codebrowser.html`, 08/15/2016.

[6] OpenGrok home page, version 0.13-rc2
`https://opengrok.github.io/OpenGrok`, 08/15/2016.

[7] The CodeCompass project,
`https://github.com/Ericsson/CodeCompass`, 08/15/2016.

[8] JSON compilation database format specification,
`http://clang.llvm.org/docs/JSONCompilationDatabase.html`, 08/15/2016.

[9] Bjarne Stroustrup: *The C++ Programming Language, 4th ed.*
Addison-Wesley, ISBN 978-0321563842, 2013.

[10] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual.* Section 7.2.1c. ISBN 0-201-51459-1, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1990.

[11] Bjarne Stroustrup: *Design and Evolution of C++*, Addison-Wesley, ISBN 0-201-54330-3, 1994.

[12] The C++14 Standard, ISO International Standard, ISO/IEC 14882:2014 – Programming Language C++, §1.3 "Terms and definitions", 2014.

[13] The C11 Standard, ISO International Standard, ISO/IEC 9899:2011 – Programming Language C, 2011.

[14] Format of a C decorated name,
`https://msdn.microsoft.com/en-us/library/x7kb4e2f.aspx`, 08/15/2016.

[15] Jonathan Sillito, Gail C. Murphy, Kris De Volder: *Asking and Answering Questions during a Programming Change Task*, IEEE Transactions on Software Engineering, VOL. 34, NO. 4, July/August 2008.

[16] The C++14 Standard, ISO International Standard, ISO/IEC 14882:2014 – Programming Language C++, §3.2 “One Definition Rule”, 08/17/2016.

[17] The C++14 Standard, ISO International Standard, ISO/IEC 14882:2014 – Programming Language C++, §3.4 “Name lookup”, 08/17/2016.

[18] Rudolf, Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy: *Columbus Reverse Engineering Tool and Schema for C++*, In Proc. 18th Int'l Conf. Software Maintenance (ICSM 2002), pp. 172–181., Oct. 2002.

[19] Clang: a C language family frontend for LLVM,
`http://clang.llvm.org`, 08/15/2016.

[20] Ferraro-Esparza, Victor, Michael Gudmandsen, and Kristofer Olsson: *Ericsson Telecom Server Platform 4*, Ericsson Review 3 (2002) pp. 104–113., accessed 08/15/2016.

[21] Apache Xerces C++, part of Apache Xerces Project, git version `26f8004`
`http://xerces.apache.org`, 08/15/2016.

[22] LLVM/Linux: GNU/Linux kernel compatible for LLVM/Clang builds, git version `230b22d`
`http://llvm.linuxfoundation.org`, 08/15/2016